Forensic Analysis of Windows User space Applications through Heap allocations.

Michael Cohen
Google Inc.,
Brandschenkestrasse 110, Zurich, Switzerland.
Email: scudette@google.com.

Abstract—**Memory analysis is now used routinely for incident response and forensic applications. Current memory analysis techniques are very effective in finding kernel artifacts of significance to the forensic investigator. However, the analysis of user space applications has not received enough attention so far. We identify the lack of pagefile support in analysis and acquisition as a major hurdle in the analysis of user space applications. We present a set of patches to the Rekall Memory Forensic platform that enable the analysis of pagefiles on all operating systems. We then continue by studying the process heaps, and in particular the Windows userspace heap allocator. We present a set of plugins to enumerate heap allocations and discover internal references. We demonstrate that using the heap allocations as a guide, it is easier to reverse engineer user space private data structure simply by observation. Finally, we apply the heap analysis technique to study the allocations made by the windows DNS client cache.**

I. Introduction

Memory analysis has gained popularity in recent years as a powerful and effective method for obtaining forensically relevant information from running computer systems.

The widespread use and availability of powerful memory analysis frameworks such as Rekall [1] or Volatility [2] has enabled this trend. Forensically relevant information, such as IP addresses, recent connections and running processes can assist in rapid triaging of forensic acquisition and analysis. For closed source operating systems, such as Microsoft Windows, much effort has been devoted into reverse engineering core parts of the operating systems in order to identify important artifacts in memory, such as important kernel structures.

However, full reverse engineering is generally time consuming and requires highly specialized skills. As a result, the analysis of application memory has been slower to develop, since applications change more frequently and there are more forensically interesting applications. While some of the more important user space applications have received some interest (e.g. the Service Control Manager [3], or the Windows Command Shell [4]), there remain many user space applications which have not been analyzed sufficiently.

This paper focuses specifically on the analysis of user space applications running under the windows operating system. We consider some of the challenges encountered in this scenario and in particular examine ways to make the reversing of the application easier through deep inspection of the user mode heap.

By enumerating all user space heap allocations one gains a view of the application memory with sufficient context to be able to recognize repeating patterns of object reuse by the application.

Our contributions in this paper fall into a number of areas. First we have written specialized code to be able to acquire and utilize the pagefile in the analysis of the Windows operating system. We find that the analysis of the pagefile is crucial to recovering sufficient data from userspace applications. Secondly we wrote a set of plugins that enumerates all heap allocations in a Windows application, and automatically identify cross allocation references.

Using these plugins it is possible to rapidly understand the inter-relationship between allocations made by an application, and thus write a parser for the data of relevance without access to the original source code. We demonstrate this technique by analyzing a windows application of forensic significance.

A. What is the user space heap?

Modern operating systems make a distinction between "user space" and "kernel space". Kernel code runs at Ring 0, while user mode code runs at Ring 3. The kernel provides general services to user mode code via a well defined system call interface [5]. One of the services a kernel may provide is memory management for the process. A process can request the kernel to augment its address space with new memory or that an external file is to be mapped into it's own address space. The kernel keeps data structures that describe the address space layout for a particular process, termed the VAD tree (Virtual Address Descriptor Tree) in order to rapidly manage the process's page tables [6].

The process may request the kernel to allocate new memory to its address space by use of the *VirtualAllocEx()* system call. This function will cause the kernel to add an additional Virtual Address Descriptor to the VAD tree and manipulate the process's page tables to allow the new memory to be accessible. The kernel is only able to extend a process's address space in multiples of whole pages.

For many applications, however, memory must be allocated in much smaller sizes from a few bytes to large allocations. For this reason, most applications use an internal heap implemented typically by a library. The heap exposes an API where the application can allocate small, arbitrary sized memory blocks (We shall term these *allocations* in this work). For example, the ANSII C *malloc()/free()* API, or the C++ *new()* APIs can be used to allocate arbitrarily sized allocations efficiently. The library itself requests page sized allocations from the kernel to service these smaller requests, carving up these larger areas into internally managed chunks.

There are many heap implementations. Below we list some of the more common implementations:

1) Doug Lee's dlmalloc [7] is one of the oldest implementations of a general purpose memory allocator. This has been adapted in ptmalloc2 which add multithreading support [8]. Ptmalloc2 is commonly used in many Unix/Linux distributions, but it is also available in windows. For example windows programs compiled under mingw or cygwin may use ptmalloc2.
2) The tcmalloc allocator is a high performance allocator written by Google and released as an open source alternative [9]. This allocator is typically used with the Chrome browser and other Google originated products.
3) The Microsoft Visual Studio heap implementation is the default implementation linked into binaries built using the Microsoft Visual Studio Development environment. All windows applications receive a single Heap created by *ntdll.dll*.

In practice different heap implementations strike a balance between the sometimes competing needs for security, performance and memory efficiency. Ultimately the heap simply requests large contiguous regions of memory from the kernel, and subdivides them into smaller sized chunks.

It is important to note that the specific heap implementation used does not depend on the operating system. It is not the case that all windows programs will use the Microsoft heap implementation. For example, the Chrome browser uses tcmalloc even on Windows. Nevertheless, the Microsoft allocator is the most commonly used by Windows applications and therefore we focus on it in this work.

The analysis in this work was performed on Windows 7 AMD64 with Service Pack 1 running inside a Virtual Box emulator with 1Gb RAM.

B. Why analyse the heap allocator?

From the kernel's point of view the application simply has large memory regions allocated to it in it's address space. Current memory analysis tools are able to dump or view this kernel's view of memory, or even search the memory for certain signatures. However, this coarse view of a process's memory is not aligned with how the process itself views its own memory, and therefore lacks the context required to properly interpret the data.

For example, a process might allocate different objects (e.g. C++ objects or C structs) on the heap, and treat those as distinct entities. If we simply examine process memmory as a large contiguous amorphous stream of bytes we lose the context of the individaul allocations and must detect the beginning and end of each object, perhaps by developing validity tests or signatures [4]. This increases the likelihood of errors and false positives.

By having a detailed understanding of the heap implementation, the forensic analyst is able to subdivide process memory into allocation sized units. Every time the process calls the 'malloc()' or 'new()' function, the returned memory can be viewed as a distinct allocation. Since typically applications allocate memory consistently for their required use, this gives the analyst a strong hint as to the purpose of the allocation. Section IV details some examples where heap analysis can be utilized to recover data from user space applications.

II. The Microsoft Heap Allocator

In this paper we focus on the Microsoft Heap allocator which is the most widely used alloctor in Windows. The allocator has been studied extensively by the information security community, and a number of detailed accounts have been published (most notably [10], [11], [12]).

The main interest in the allocator is from a security perspective. That is, how the allocator can be exploited as a result of a heap overflow bug. While in order to successfully and reliably exploit the heap allocator, deep understanding of the allocator operations is required, the forensic analyst is merely interested in enumerating heap allocations.

Heap implementations maintain data structures to allow fast and efficient operation of the heap (e.g. lookaside lists, fine grained thread locks) but precise knowledge of these mechanisms is not required in order to simply enumerate the heap allocations. Most heaps maintain high level structures that allow simple enumeration. Even though the literature contains very detailed explanation of the heap operation, we include a simple overview in this paper to describe the basic method for enumerating heap allocations.

Some existing forensic frameworks do support limited analysis of heap allocations. For example the Volatility memory analysis framework supports extracting edited text from the notepad applications [2], it does not however, support the full heap analysis - only the backend allocator. Additionally the existing implementation does not support the heap entry encoding. Therefore it can not be used for recovering small allocations which are typically handled by the low fragmentation heap, or work with newer windows versions then XP. Similarly [13] provides an overview of the basic heap operation but it does not utilize any front end heap implementations, making it impossible to locate very small allocations.

To our knowledge our implementation of front end heap analysis is unique in a memory forensic tool. It, of course, relies heavily on the detailed descriptions presented by the seminal work in [12].

The following description applies to Windows 7 SP1 heap implementation. Older implementations use slightly different details. Note that the actual heaps used in a process depend on the way the process has been built. Specifically with Microsoft Visual Studio, it is possible to make a debug build in which case the process will create debugging heaps with a different layout then the regular production heaps. These debugging heaps contain larger heap metadata structs and the allocation layouts are different. Most production software however, is not built with the debugging configuration. Note also that on windows if a process is running under WoW64 emulation it will have a 32 bit heap implementation, even when running on a 64 bit operating system.

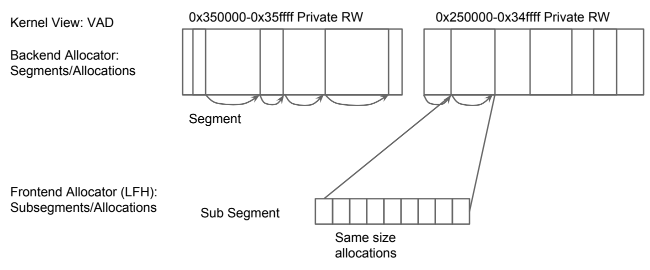

Kernel View: VAD
0x350000-0x35ffff Private RW
0x250000-0x34ffff Private RW
Backend Allocator: Segments/Allocations
Segment
Frontend Allocator (LFH): Subsegments/Allocations
Sub Segment
Same size allocations

Fig. 1. A typical heap.

A process may contain multiple heaps, with new heaps created via the *CreateHeap()* API. When a process is created, it is given the first initial heap (created by *ntdll.dll*). The process's heaps are listed in the Process Environment Block (*_EPROCESS.Peb.ProcessHeaps*) array which contains pointers to a *_HEAP* header. Sometimes different components of the same process will build different heaps for different purposes. As we see in subsequent sections (Section IV), we can use this property to identify related data structures, since all data from the same component will be grouped in a single logical heap.

The Windows heap allocator is divided into two parts [14]:

1) The backend allocator is responsible for requesting large contiguous memory blocks from the kernel using *VirtualAllocEx()*. These blocks (called Segments) are divided into smaller chunks to service small allocations. Note that very large allocations are serviced directly with VirtualAllocEx() by the backend.
2) The frontend allocator is used to improve performance for small allocations. For current windows versions the only front end implementation available is the Low Fragmentation Heap (LFH) implementation (and that is the only one we consider in this work). A LFH is only created when the implementation detects the application will benefit from it, hence not all heaps contain a front end [12].

The LFH allocates larger blocks from the backend allocator (termed Subsegments) and carves these into identically sized allocations. If an allocation request can be served from an existing Subsegment, the Front End Heap serves it from this Subsegment. Allocations with each LFH Subsegment are not coalesced hence this reduces heap fragmentation.

Figure 1 illustrates an example heap for a simple application. The Kernel is only aware of two VAD descriptors for large contiguous regions of private process memory. The heap's backend creates these VAD regions (called Segments) using *VirtualAllocEx()* calls.

The backend then divides the segments into single allocations. Each allocation is preceeded with a *_HEAP_ENTRY* data struct header (which is 16 bytes). The header contains the size of the allocation, and therefore the beginning of the next allocation can be calculated. It is therefore possible to follow the allocation from the start of the segment to the end in order to enumerate all allocations in this segment. When the heap can not fit an allocation into an existing Segment, it requests a new Segment to be allocated using another VirtualAllocEx() call.

The *_HEAP_ENTRY* data structure is encoded in practice, in order to make it difficult for heap overflow exploits. Before parsing the struct it must be decoded by XORing it with the constant given at the *_HEAP.Encoding* field. I.e. the Heap Encoding is different and random for each heap in the process.

The backend allocations can be enumerated using this simple algorithm (On Windows 7):

1) The pointer *_EPROCESS.Peb.ProcessHeaps* points at an array of pointers to *_HEAP* structures.
2) For each *_HEAP* structure, enumerate the segments by following the *_HEAP.SegmentListEntry* list.
3) for each *_HEAP_SEGMENT* struct, *_HEAP_SEGMENT.FirstEntry* is the first *_HEAP_ENTRY* header.
4) Decode the *_HEAP_ENTRY* header by XOR with the *_HEAP.Encoding* field.
5) Find the next entry by adding the allocation size *_HEAP_ENTRY.Size* multiplied by the heap allocation granularity to this entry.

III. The Windows Pagefile

When we initially attempted to enumerate process heaps, memory forensic tools were unable to process the pagefile or prototype PTEs correctly. We found that many pages in the heaps were invalid making it difficult to follow the heap allocations and perform user space analysis. Clearly as illustrated in Figure 1, if pages are unreadable part way through a heap segment it is impossible to follow the allocation list completely to the end of the heap. Thus heap enumeration becomes incomplete.

Many of the existing plugins are designed to analyze kernel memory. Although technically the windows kernel is also pageable, in practice many of the current plugins which analyze kernel structures are not particularly affected by paging, even though the effects of paging on userspace analysis are quite profound.

The windows kernel can allocate memory from one of the paged or non-paged pools for its own kernel data structures. We used the Rekall *pool_tracker* plugin [1] to examine from which pool different kernel allocations were made in practice. To our surprise we found that most significant kernel structures are allocated in Nonpaged pools (Allocations such as processes, threads, TCP connections, VAD entries and many more). Some kernel allocations did come from Paged pool (and would benefit from our pagefile support), such as the Registry structures (Pool tags starting with CM) and NTFS structures, but on the whole, most plugins are using structures from non-paged pools.

This explains why the pagefile was not sorely missed when analyzing kernel structures until now. However, when analyzing userspace memory, the page file plays a much greater role.

The literature documents many researchers identifying the pagefile as an important source of information. For example [15] notes a vast increase in the number of available pages when page translation considers pages in Transition and Prototype states. References are found in the literature as early as [16] to implementations of pagefile support in memory analysis tools, but these claims were never followed through with published code.

The windows page file was studied in the past by [15] and some heuristics were developed by observation. This early analysis was confined to 32 bit versions of Windows XP, and does not include the analysis of file mappings (Section objects). To our knowledge this analysis has not been not implemented in any current open source tools, before we developed our patches. [3] claims that currently the Volatility memory forensic framework does not support analysis of page files.

[14] explains how PTEs can be parsed in order to recover memory from the page file. We have developed a set of patches to the Rekall memory forensic tool to enable the acquisition and analysis of page-files. Detailed information about these patches are published elsewhere [17] however, in this section we outline a short summary of the changes as they pertain specifically to the analysis of userspace applications.

A. Virtual Address Translation

The Windows operating system utilizes the CPU's protected mode. In this mode all memory accesses made by the CPU are done in *Virtual Addresses*. The hardware's Memory Management Unit (MMU) uses specially configured *Page Tables* in order to translate a virtual address into a physical memory address (where data can be fetched) [14]. Each process has its own set of page tables which are used to reference its own unique *Process Address Space*.

The details of this translation process are described elsewhere [18], but for our purposes we simply point out that the resolution process retrieves a *Page Table Entry (PTE)* from the page tables for each virtual page accessed. This PTE contains information encoded in various bitfields about where the physical page corresponding with the virtual address may be located in the physical memory.

Note that the address translation process occurs automatically by the MMU hardware. However, if the PTE has it's least significant bit (bit 0) unset then the PTE is considered *Invalid* by the hardware, which generates a *Page Fault* interrupt for the kernel to resolve the page.

A fundamental capability of memory analysis frameworks is to emulate this address translation process in order to present an abstraction for the process virtual address (So process memory can be examined). Earlier frameworks would strictly perform the translation made by the MMU but if the Valid bit was 0 would consider that the page is invalid and not available. This is unfortunately insufficient since in practice many pages in process memory are invalid (from a hardware point of view), but can still be resolved using the OS page-fault handler (See Figure 4).

We have extended the Rekall memory forensic framework with an OS specific Virtual Address Translation facility which considers the special cases where the Valid flag is unset in a PTE [17]. In these cases the other bits of the PTE are able to be used freely by the operating system, and therefore are OS specific by nature.

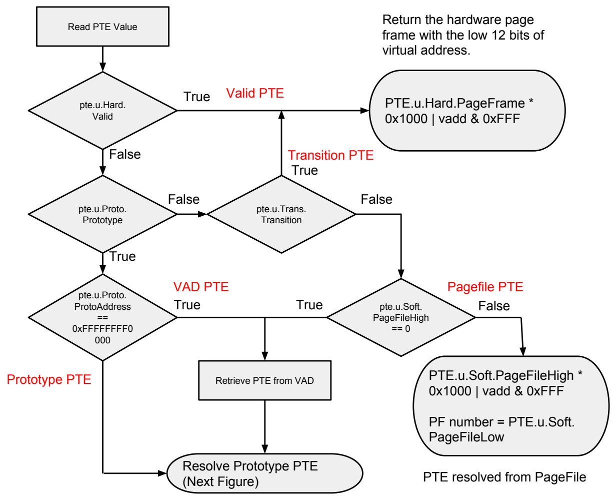

Read PTE Value
Return the hardware page frame with the low 12 bits of virtual address.
pte.u.Hard.Valid
True
Valid PTE
PTE.u.Hard.PageFrame * 0x1000 | vadd & 0xFFF
False
Transition PTE
True
pte.u.Proto.Prototype
False
pte.u.Trans.Transition
False
True
pte.u.Proto.ProtoAddress == 0xFFFFFFFF0000
VAD PTE
True
True
pte.u.Soft.PageFileHigh == 0
Pagefile PTE
False
Prototype PTE
Retrieve PTE from VAD
PTE.u.Soft.PageFileHigh * 0x1000 | vadd & 0xFFF
PF number = PTE.u.Soft.PageFileLow
Resolve Prototype PTE (Next Figure)
PTE resolved from PageFile

Fig. 2. Hardware PTE resolution algorithm.

B. OS specific address translation

The hardware generates a page fault while translating a page, by calling an interrupt into the OS page fault handler, passing in the faulting PTE. At this point the page fault handler uses a number of flags to determine which state the PTE is in. Windows uses the *_MMPT* struct to describe the PTE which is actually a union of all the possible states the PTE can be in (The correct member of the union is chosen based on the flags).

Figure 2 shows the algorithm used for resolution of the PTE passed into the page fault handler (Sometimes termed the Hardware PTE). Our Rekall patches implement this algorithm in order to emulate the page-fault handler and deduce the correct physical page to use.

In the first stage the PTE might represent one of the following states:

1) Valid PTE: If Bit 0 is set the PTE refers to a page in physical memory.
2) If the ProtoType bit is unset and the Transition bit is set the page is in the Transition state. Its content is still valid and therefore we can directly read the data from the image.
3) If both the ProtoType bit and the Transition bit are unset, the PTE refers to a Software PTE (i.e. the data exists in the pagefile). The offset into the pagefile can be calculated from the *PageFileHigh* field. Except if the offset into the pagefile is 0, in this case, the VAD must be consulted and the ProtoType PTE recovered from the VAD and analyzed through the second stage algorithm.
4) If the ProtoType bit is set, and the ProtoAddress field is 0xFFFFFFFF0000 then the ProtoType PTE must be fetched from the VAD entry corresponding with the relevant address. The ProtoType PTE is then found by using the *_MMVAD.FirstPrototypePte*

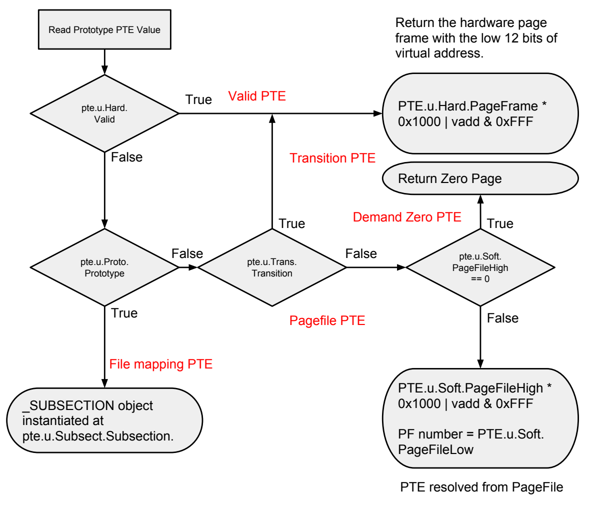

Read Prototype PTE Value
pte.u.Hard.Valid
True
Valid PTE
False
Return the hardware page frame with the low 12 bits of virtual address.
PTE.u.Hard.PageFrame * 0x1000 | vadd & 0xFFF
Transition PTE
Return Zero Page
Demand Zero PTE
True
True
pte.u.Proto.Prototype
False
pte.u.Trans.Transition
False
pte.u.Soft.PageFileHigh == 0
True
Pagefile PTE
False
File mapping PTE
_SUBSECTION object instantiated at pte.u.Subsect.Subsection.
PTE.u.Soft.PageFileHigh * 0x1000 | vadd & 0xFFF
PF number = PTE.u.Soft.PageFileLow
PTE resolved from PageFile

Fig. 3. Prototype PTE resolution algorithm.

pointer. The retrieved PTE is then processed further as a ProtoType PTE in the second stage.

If a ProtoType PTE was retrieved in the first stage, it is analyzed again to try to resolve the page. ProtoType PTEs are allocated from system pool and are not part of the hardware page tables (i.e. The MMU never reads a ProtoType PTE directly), but they share the same kernel structs and add some additional PTE states. Figure 3 illustrates the algorithm for resolving a prototype PTE. One major difference from the previous algorithms is the case where the prototype bit is set in a ProtoType PTE. The PTE then represents a Subsection Object (i.e. a File Mapping). When the page fault handler encounters a Subsection PTE it simply reads the data from the file into a new physical page. However, in a memory analysis framework we are unable to resolve this page without access to the corresponding disk image.

Additionally if the ProtoType PTE is a Software PTE (i.e. refers to the pagefile) but the offset into the pagefile is 0, the PTE is considered a “Demand Zero PTE”. In this case the kernel will re-purpose a zeroed page and remap it into memory on demand.

In order to evaluate the importance of the page file in the analysis of usermode applications, we wrote a Rekall plugin called *vadmap*, that enumerates every process PTE (from the VAD tree) and determines it’s state. As an example, we have used a memory image of a Windows 7 system and enumerated the heaps of the *svchost.exe* process (This process contains forensically significant information which will be analyzed in a later section).

If we only consider heap owned pages, there were 3286 pages in total, of which 457 were valid and 144 were in Transition (i.e. 601 pages were immediately usable). However there were 431 pages which only exist in the pagefile and 2254 Zero pages (I.e. unused by the heaps). Hence, in this case, the pagefile actually contains over 40% of the usable pages in the process heaps. Clearly the exact ratio of pagefile pages to valid pages depends greatly on external factors, such as available memory, process activity and the time since the particular page was used. However, this example demonstrates that the page file is extremely important to the analysis of user-space applications.

It is therefore crucial that the pagefile be acquired at the same time as the memory. Unfortunately, the pagefile is locked for shared access when the system is running, thus it can not ordinarily be opened for reading. Despite this, some memory acquisition tools can still acquire the pagefile. For example judging by the debugging output, [19] searches for the handle for the open pagefile in the System process (Process ID 4) and then reuses that handle to read the pagefile.

Alternatively, it is possible to acquire the pagefile using the *fcat* tool from the *Sleuthkit* suite of forensic applications [20]. This tool parses the NTFS Master File Table (MFT) and directly extracts the disk clusters that the pagefile resides on [21]. This approach bypasses the regular kernel file lock restrictions and enables reading the locked file.

C. Smear and pagefile acquisition.

While experimenting with the pagefile processing patches we found an interesting effect of smear during acquisition. Smear in memory acquisition is not a new issue [22], [23], but is typically measured in terms of total pages changed from an “atomic” snapshot (for example obtained though virtual machine introspection [24], [25]).

However, we found that not all page changes are the same. In fact we identified a new type of smear dubbed “Pagetable smear”, which depends on the order of acquisition. In order to study the effects of this smear we wrote a program which allocates 800mb of memory (called *swapper.exe*), marking each page with a unique sequence number. We ran the program and observed that from the total 800mb allocated, the program’s working set was 347mb prior to acquisition (i.e. 347mb were present in memory or in the Valid state).

We then acquired memory using the WinPmem 1.6.2 tool[1]. Afterwards we copied the pagefile from the system using the *fls* tool. Within seconds of the acquisition starting, the working set of *swapper.exe* was trimmed to 188mb - possibly due to the IO operations in writing the image to disk.

We then used our patches for OS specific page translation to read the known pages from our test program. To our surprise we found, among the known marked pages, some corrupted pages - even some which appear to contain kernel data (which should never appear in a userspace program).

After further analysis we realized that the order of acquisition matters - as the page tables are usually stored at low memory addresses, in this case the kernel’s DTB was at 0x187000 (About 1.6Mb into the image), the page tables will be copied first into the image. The PTEs for the process might still indicate that some pages are in a Valid (or Transition) state. However, by the time the acquisition tool gets to copy those physical pages into the disk, the kernel’s working set trimmer might decide to trim the process’s working set so the PTEs will be in the Software PTE or ProtoType PTE states, while the actual pages are re-purposed.

This type of smear is very dangerous since the Memory analysis framework has no idea that the pages are in any way invalid. It would simply go through its normal page translation algorithm and use the pages indicated by the original PTE, even though these pages are now no longer part of the process working set. Thus an out of date pagetable page has much larger consequences to analysis than simply an out of date kernel or process memory page.

Similar effects have been noted (but not experimentally verified) by [26]. However, in that work the effects were attributed to smear with respect to kernel swap data structures. We have determined that this is not necessarily related to swap specifically, or to kernel data structures, since the same effects can be observed for prototype pages and memory mapped pages. In fact any difference between the acquisition times of the PTEs and the pages they refer to can be classified as "pagetable smear".

Further research is required to determine the optimal order of acquisition. Perhaps page tables should be acquired after the rest of main memory. Perhaps they should they be acquired twice - before and after, and the differences noted.

IV. Userspace Heap Analysis

The following is an example of applying heap analysis to reversing the windows DNS Client Resolver Cache. Although we examine a single example here, the technique has been successfully applied to a number of other user space applications.

Windows implements a common caching resolver for DNS client queries. Every program which queries for a DNS resolution will cause an entry to be added to this cache for a period of time. Being able to inspect the cache is valuable from a forensic perspective since DNS activity on the machine is a strong signal for the presence of network connections (e.g. for malware Command and Control connections).

An early implementation of a plugin to extract the DNS cache was published on the Volatility issue tracker [27], however the plugin only supported some versions of windows XP 32 bit and used scanning techniques to traverse the heap (and hence does not work on windows 7).

The file *dnsrslvr.dll* describes itself as "DNS Caching Resolver Service" in its PE Version strings, and is therefore responsible for implementing the resolver cache. It is also a service hence it is running in *svchost.exe*.

We now apply our userspace heap analysis plugins to learn more about this service. The first plugin we use is *inspect_heap* to learn about the allocations in all heaps.

Figure 4 shows the output of the *inspect_heap* plugin. The *svchost.exe* process hosts many services, each implemented as a dll at the same time. Therefore there are quite a number of heaps present. The above figure only shows an extract from the full output. In it we notice clearly domain names such as *ssl.bing.com*.

Figure 5 demonstrates the analysis process. We use the *show_allocation* plugin to locate the allocation which contains an address of interest. The plugin dumps the data within this allocation. The plugin also iterates over each 8 byte integer and attempts to resolve it back to a known allocation. The overall effect is that we can identify pointers to other allocations.

```
************************************************
0xfa8002b4f6c0 svchost.exe  1148
Heap 4: 0x11d0000 (BACKEND)
Backend Info:

Segment            End          Length   Data
------------- -------------- -------- ----
. 0x11d0040        0x1250000 524224
.. 0x11d0a80       0x11d12f0 2144     00 00 00 00 00 00 00 00  ........
                                      00 01 00 00 00 00 00 00  ........
.. 0x11d12f0       0x11d1500 512      00 13 1d 01 00 00 00 00  ........
                                      00 13 1d 01 00 00 00 00  ........
.. 0x11e27f0       0x11e2820 32       73 00 73 00 6c 00 2e 00  s.s.l...
                                      67 00 73 00 74 00 61 00  g.s.t.a.
.. 0x11e2820       0x11e2860 48       61 00 2d 00 30 00 30 00  a.-.0.0.
                                      30 00 31 00 2e 00 61 00  0.1...a.
# This looks like a DNS name.
.. 0x11e2860       0x11e28c0 80       73 00 73 00 6c 00 2d 00  s.s.l.-.
                                      62 00 69 00 6e 00 67 00  b.i.n.g.
.. 0x11e28c0       0x11e28f0 32       40 78 24 01 00 00 00 00  @x$.....
                                      20 51 24 01 00 00 00 00  .Q$.....
.. 0x11e28f0       0x11e2930 48       40 29 1e 01 00 00 00 00  @)......
                                      80 28 87 02 00 00 00 00  .(......

Heap 12: 0x5100000 (LOW_FRAG)
Backend Info:

Segment            End          Length   Data
------------- -------------- -------- ----
. 0x5100040        0x5110000 65472
.. 0x5100a80       0x51012e0 2128     00 00 00 00 00 00 00 00  ........
                                      00 01 00 00 00 00 00 00  ........
.. 0x5107fc0       0x5108000 48       d0 df 36 05 00 00 00 00  ..6.....
                                      f8 00 10 05 00 00 00 00  ........
.. 0x5108000       0x5110000 32752    00 00 00 00 00 00 00 00  ........
                                      00 00 00 00 00 00 00 00  ........
. 0x5360040        0x5460000 1048512
.. 0x5360070       0x536c4d0 50256    40 2a ef 02 00 00 00 00  @*......
                                      ff ff ff ff 00 00 00 00  ........
.. 0x536c4d0       0x536c540 96       00 00 00 00 00 00 00 00  ........
                                      10 c5 36 05 00 00 00 00  ..6.....
.. 0x536e000       0x5460000 991216   00 00 00 00 00 00 00 00  ........
                                      00 00 00 00 00 00 00 00  ........
Low Fragmentation Front End Information:
    Entry      Alloc Length Data
-------------- ----   ----  ----
     0x536c760 96        64 00 00 00 00 00 00 00 00  ........
                            a0 c7 36 05 00 00 00 00  ..6.....
                            00 00 00 00 03 00 00 00  ........
                            a0 3b 1e 01 00 00 00 00  .;......
                            00 00 00 00 00 00 00 00  ........
                            00 00 00 00 00 00 00 00  ........
                            74 00 2e 00 75 00 72 00  t...u.r.
                            73 00 2e 00 6d 00 69 00  s...m.i.
     0x536c7c0 96        64 00 00 00 00 00 00 00 00  ........
                            00 c8 36 05 00 00 00 00  ..6.....
                            00 00 00 00 03 00 00 00  ........
                            c0 51 1e 01 00 00 00 00  .Q......
                            00 00 00 00 00 00 00 00  ........
                            00 00 00 00 00 00 00 00  ........
                            63 00 72 00 6c 00 2e 00  c.r.l...
                            74 00 68 00 61 00 77 00  t.h.a.w.
```

Fig. 4. Enumerating the heap of the *svchost.exe* process which is running the *dnsrslvr.dll* DNS Client Resolver Cache. The process has 12 heaps in this case but only the relevant heaps to the following discussion are shown.

Additionally we use the *show_referrer_alloc* plugin to search for all references to a memory address from heap allocations. Matching allocations are then displayed.

Typically, userspace applications maintain data structures using pointers to other data structures in a consistent way. Given our knowledge of allocations, we are able to identify similar objects and quickly identify the patterns which hint at the data structures involved.

So to summarise the analysis process outlined in Figure 5:

1) We identify a domain name in an allocation at 0x11e2860, using the *show_allocation* plugin we see that this is a single string (*ssl-bing-com.a-0001.a-msedge.net*).
2) Now we ask "which data structures refer to this string?" Using the *show_referrer_alloc* plugin we see an allocation at 0x11e2b30. We shall name the data structure *DNS_RECORD*. The string reference is at offset 0x08 from the start of the struct.
3) We then ask, "which data structure points to the *DNS_RECORD*?" The allocation at 0x1244fc0 looks

remarkably like a *DNS_RECORD* too, hence we identify a singly linked list at offset 0x00 of this struct. In fact each *DNS_RECORD* is storing an additional piece of information about the record. By observation we can see the type of the struct is stored at offset 0x20 of this struct - Type 0x05 refers to a CNAME record and type 0x1 refers to an A record.

4) By repeating the *show_referrer_alloc* plugin we eventually get to a struct which does not look like a *DNS_RECORD* at offset 0x5101f60. In fact we can see it is located in a different heap altogether. The struct seems to contains a reference to the string “ssl.bing.com” at offset 0x08.

5) Following the referrer of this struct seems to be a large allocation with sparse pointers to similar such structs mixed with NULL pointers. This looks like a hash table, therefore we name the previous struct *DNS_HASHTABLE_ENTRY*.

Figure 7 illustrates the relationships between all the allocations diagrammatically. We used this to write a new plugin: First locate the relevant heap from a reference inside *dnsrslvr.dll*, then find the hashtable (which is the largest single allocation). The members are then followed and printed in order.

```
[1] output.elf.E01 23:59:04> show_allocation 0x11e2860
Address      0x11e2870 is 0 bytes into allocation of
 size 88 (     0x11e2870 -      0x11e28c8)
    Offset                   Data                                  Comment
-------------- ---------------------------------- --------------------------
     0x11e2860 2e 00 63 00 31 00 30 00  ..c.1.0. _HEAP_ENTRY
     0x11e2868 36 16 3e 50 18 57 00 1e  6.>P.W.. _HEAP_ENTRY
     0x11e2870 73 00 73 00 6c 00 2d 00  s.s.l.-.
     0x11e2878 62 00 69 00 6e 00 67 00  b.i.n.g.
     0x11e2880 2d 00 63 00 6f 00 6d 00  -.c.o.m.
     0x11e2888 2e 00 61 00 2d 00 30 00  ..a.-.0.
     0x11e2890 30 00 30 00 31 00 2e 00  0.0.1...
     0x11e2898 61 00 2d 00 6d 00 73 00  a.-.m.s.
     0x11e28a0 65 00 64 00 67 00 65 00  e.d.g.e.
     0x11e28a8 2e 00 6e 00 65 00 74 00  ..n.e.t.
     0x11e28b0 00 00 6f 00 6f 00 6b 00  ..o.o.k.
     0x11e28b8 2e 00 63 00 6f 00 6d 00  ..c.o.m.

# Lets check who refers to this allocation. We will call the referring struct
#     DNS_RECORD.
[1] output.elf.E01 23:59:11> show_referrer_alloc 0x11e2870
Address      0x11e2b38 is 8 bytes into allocation of
  size 72 (     0x11e2b30 -      0x11e2b78)
    Offset                   Data                                  Comment
-------------- ---------------------------------- --------------------------
     0x11e2b20 00 00 00 00 00 00 00 00  ........ _HEAP_ENTRY
     0x11e2b28 35 16 3e 53 1a 57 00 28  5.>S.W.( _HEAP_ENTRY
     0x11e2b30 c0 21 87 02 00 00 00 00  .!...... 0x28721c0(56@0x28721c0)
     0x11e2b38 70 28 1e 01 00 00 00 00  p(...... 0x11e2870(88@0x11e2870)
     0x11e2b40 05 00 08 00 09 30 00 00  .....0..
     0x11e2b48 73 17 04 00 01 00 00 00  s.......
     0x11e2b50 30 28 1e 01 00 00 00 00  0(...... 0x11e2830(56@0x11e2830)
```

Fig. 5. Analysis of allocations in *svchost.exe* heaps.

V. Conclusions and Future Work

The present work examined a novel method of user space program analysis using the enumerations of heap allocations. We present patches for the Rekall memory forensic tool that enable the analysis of the pagefile in an operating system specific way. We find that the pagefile is crucial for the analysis of user space programs, more so than kernel data structures.

We present a set of plugins which enumerate program heap allocations and identify inter-relations between these allocations. The plugins can be used to rapidly reverse engineer many user space programs by simply examining their in-memory data structures and identifying some of the more

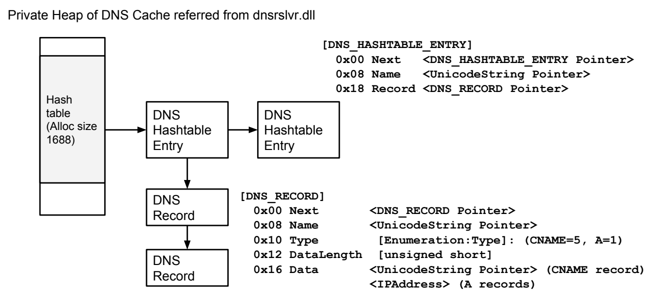
```
# Who refers to the DNS_RECORD? This looks a lot like another DNS_RECORD.
[1] output.elf.E01 23:59:33> show_referrer_alloc 0x11e2b30
Address      0x1244fc0 is 0 bytes into allocation of
  size 56 (     0x1244fc0 -      0x1244ff8)
    Offset                   Data                                  Comment
-------------- ---------------------------------- --------------------------
     0x1244fb0 6f 00 67 00 6c 00 65 00  o.g.l.e. _HEAP_ENTRY
     0x1244fb8 34 16 3e 52 1e 57 00 18  4.>R.W.. _HEAP_ENTRY
     0x1244fc0 30 2b 1e 01 00 00 00 00  0+...... 0x11e2b30(72@0x11e2b30)
     0x1244fc8 30 54 24 01 00 00 00 00  0T$..... 0x1245430(40@0x1245430)
     0x1244fd0 05 00 08 00 09 30 03 00  .....0..
     0x1244fd8 73 17 04 00 01 00 00 00  s.......
     0x1244fe0 d0 2a 1e 01 00 00 00 00  .*...... 0x11e2ad0(88@0x11e2ad0)

# Who refers to that one? This does not look like a DNS_RECORD at all.
[1] output.elf.E01 23:59:48> show_referrer_alloc 0x1244fc0
Address      0x5101f78 is 24 bytes into allocation of
  size 88 (     0x5101f60 -      0x5101fb8)
    Offset                   Data                                  Comment
-------------- ---------------------------------- --------------------------
     0x5101f58 13 71 d2 2f 89 53 00 16  .q./.S.. _HEAP_ENTRY
     0x5101f60 00 00 00 00 00 00 00 00  ........
     0x5101f68 90 1f 10 05 00 00 00 00  ........ +0x40(0x5101f90)
     0x5101f70 00 00 00 00 03 00 00 00  ........
     0x5101f78 c0 4f 24 01 00 00 00 00  .O$..... 0x1244fc0(56@0x1244fc0)
     0x5101f80 00 00 00 00 00 00 00 00  ........
     0x5101f88 00 00 00 00 00 00 00 00  ........
     0x5101f90 73 00 73 00 6c 00 2e 00  s.s.l...
     0x5101f98 62 00 69 00 6e 00 67 00  b.i.n.g.
     0x5101fa0 2e 00 63 00 6f 00 6d 00  ..c.o.m.

# The referrer of this record seems to be a hash table (following all non zero
#     pointers yields similar records to the above DNS_HASHTABLE_RECORD.
[1] output.elf.E01 00:00:00> show_referrer_alloc 0x5101f60
Address      0x5101860 is 1392 bytes into allocation of
  size 1688 (     0x51012f0 -      0x5101988)
    Offset                   Data                                  Comment
-------------- ---------------------------------- --------------------------
     0x5101840 00 00 00 00 00 00 00 00  ........
     0x5101848 00 00 00 00 00 00 00 00  ........
     0x5101850 00 00 00 00 00 00 00 00  ........
     0x5101858 00 00 00 00 00 00 00 00  ........
     0x5101860 60 1f 10 05 00 00 00 00  `....... 0x5101f60(88@0x5101f60)
     0x5101868 00 00 00 00 00 00 00 00  ........
     0x5101870 00 00 00 00 00 00 00 00  ........
     0x5101878 00 00 00 00 00 00 00 00  ........
     0x5101880 e0 dc 36 05 00 00 00 00  ..6..... 0x536dce0(72@0x536dce0)
     0x5101888 00 00 00 00 00 00 00 00  ........

[1] output.elf.E01 00:00:24> show_allocation 0x536c9b0
Address      0x536c9b0 is 0 bytes into allocation of
  size 88 (     0x536c9b0 -      0x536ca08)
    Offset                   Data                                  Comment
-------------- ---------------------------------- --------------------------
     0x536c9a0 63 00 6f 00 6d 00 00 00  c.o.m... _HEAP_ENTRY
     0x536c9a8 57 df 1f 73 6b 00 00 88  W..sk... _HEAP_ENTRY
     0x536c9b0 00 00 00 00 00 00 00 00  ........
     0x536c9b8 e0 c9 36 05 00 00 00 00  ..6..... +0x40(0x536c9e0)
     0x536c9c0 00 00 00 00 03 00 00 00  ........
     0x536c9c8 d0 5d 24 01 00 00 00 00  .]$..... 0x1245dd0(56@0x1245dd0)
     0x536c9d0 00 00 00 00 00 00 00 00  ........
     0x536c9d8 00 00 00 00 00 00 00 00  ........
     0x536c9e0 63 00 6c 00 69 00 65 00  c.l.i.e.
     0x536c9e8 6e 00 74 00 73 00 33 00  n.t.s.3.
     0x536c9f0 2e 00 67 00 6f 00 6f 00  ..g.o.o.
     0x536c9f8 67 00 6c 00 65 00 2e 00  g.l.e...
     0x536ca00 63 00 6f 00 6d 00 00 00  c.o.m...
```

Fig. 6. Analysis of allocations in *svchost.exe* heaps.

common data structures. This type of analysis can be utilized to quickly extract forensically significant artifacts from memory images.

We then apply this analysis to study the internals of the windows DNS client cache, producing a plugin to enumerate all cached entries - information which is forensically significant for the detection of malware communication channels.

We also document a new type of memory acquisition smear called “Pagetable Smear”. This occurs mainly due to the fact that many pages may change state during the acquisition process, between the time the page table itself was imaged, and the page referenced by the page-table was imaged. This type of smear might result in completely incorrect pages being considered part of the process’s working set. Further research is required to establish the optimal acquisition order to minimize this kind of smear.


Private Heap of DNS Cache referred from dnsrslvr.dll
Hash table (Alloc size 1688)
DNS Hashtable Entry
DNS Hashtable Entry
DNS Record
DNS Record

```
[DNS_HASHTABLE_ENTRY]
  0x00 Next   <DNS_HASHTABLE_ENTRY Pointer>
  0x08 Name   <UnicodeString Pointer>
  0x18 Record <DNS_RECORD Pointer>
```

```
[DNS_RECORD]
  0x00 Next       <DNS_RECORD Pointer>
  0x08 Name       <UnicodeString Pointer>
  0x10 Type        [Enumeration:Type]: (CNAME=5, A=1)
  0x12 DataLength  [unsigned short]
  0x16 Data       <UnicodeString Pointer> (CNAME record)
                  <IPAddress> (A records)
```

```
[1] output.elf.E01 11:28:41> dns_cache
     Name                             Record       Type    Data
------------------------------  -------------- ------- ----
  dl.google.com                 0x00000536c710 HTABLE
. dl.google.com                 0x000002872b30 CNAME   dl.l.google.com
. dl.l.google.com               0x0000011e2a30 A       173.194.44.5
. dl.l.google.com               0x0000011e3880 A       173.194.44.3
  www.facebook.com              0x000005101f00 HTABLE
. www.facebook.com              0x0000011e3030 CNAME   star.c10r.facebook.com
  ssl.bing.com                  0x000005101f60 HTABLE
. ssl.bing.com                  0x000001244fc0 CNAME   ssl-bing-com.a-0001.a-
                                                       msedge.net
. ssl-bing-com.a-0001.a-msedg   0x0000011e2b30 CNAME   a-0001.a-msedge.net
  e.net
. a-0001.a-msedge.net           0x0000028721c0 A       204.79.197.200
```

Fig. 7. Final DNS Cache data diagram deduced by examining the allocations in Figure 5. Below is a sample of the output of the *dns_cache* plugin. The plugin indicates the address of the hash table entry and then follows each of the *DNS_RECORD* entries.

REFERENCES

[1] "Rekall memory forensic framework," http://www.rekall-forensic.com/, 2015, [Online; accessed 09-Feb-2015].

[2] "Volatility," https://github.com/volatilityfoundation/volatility, 2015, [Online; accessed 09-Feb-2015].

[3] M. H. Ligh, A. Case, J. Levy, and A. Walters, *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*, 1st ed. Wiley Publishing, 2014.

[4] R. M. Stevens and E. Casey, "Extracting windows command line details from physical memory," *digital investigation*, vol. 7, pp. S57–S63, 2010.

[5] A. S. Tanenbaum, "Modern operating systems," 2009.

[6] B. Dolan-Gavitt, "The vad tree: A process-eye view of physical memory," *digital investigation*, vol. 4, pp. 62–64, 2007.

[7] D. Lea, "A Memory Allocator," http://g.oswego.edu/dl/html/malloc.html, 2000, [Online; accessed 09-Feb-2015].

[8] N. Douglas, "ptmalloc2 homepage," http://www.nedprod.com/programs/Win32/ptmalloc2/, 2013, [Online; accessed 09-Feb-2015].

[9] S. Ghemawat and P. Menage, "TCMalloc : Thread-Caching Malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2015, [Online; accessed 09-Feb-2015].

[10] C. Valasek and T. Mandt, "Windows 8 heap internals," *Black Hat USA*, 2012.

[11] J. McDonald and C. Valasek, "Practical windows xp/2003 heap exploitation," *Black Hat USA*, 2009.

[12] C. Valasek, "Understanding the low fragmentation heap," 2010. [Online]. Available: http://illmatics.com/Understanding_the_LFH.pdf

[13] A. White, "Identifying the unknown in user space memory," Ph.D. dissertation, QUT, 2013. [Online]. Available: http://eprints.qut.edu.au/64181/1/Andrew_White_Thesis.pdf

[14] M. Russinovich, D. Solomon, and A. Ionescu, *Windows Internals*, ser. Developer Reference. Pearson Education, 2012, no. pt. 1. [Online]. Available: https://books.google.ch/books?id=w65CAwAAQBAJ

[15] J. D. Kornblum, "Using every part of the buffalo in windows memory analysis," *Digital Investigation*, vol. 4, no. 1, pp. 24–29, 2007.

[16] N. L. Petroni, A. Walters, T. Fraser, and W. A. Arbaugh, "Fatkit: A framework for the extraction and analysis of digital forensic data from volatile system memory," *Digital Investigation*, vol. 3, no. 4, pp. 197–210, 2006.

[17] M. Cohen, "Windows Virtual Address Translation and the Pagefile." http://rekall-forensic.blogspot.ch/2014/10/windows-virtual-address-translation-and.html, 2014, [Online; accessed 09-Feb-2015].

[18] Intel, *Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A*. Intel Corporation, 2015, vol. 3A, ch. Chapter 3.

[19] GMG Systems, Inc., "KnTTools with KnTList," KnTToolswithKnTList, 2015, [Online; accessed 09-Feb-2015].

[20] B. Carrier, "The Sleuth Kit," http://www.sleuthkit.org/sleuthkit/desc.php, 2015, [Online; accessed 09-Feb-2015].

[21] Brian Carrier, *File system forensic analysis*. Addison-Wesley Reading, 2005, vol. 3.

[22] S. Vömel and F. C. Freiling, "Correctness, atomicity, and integrity: defining criteria for forensically-sound memory acquisition," *Digital Investigation*, vol. 9, no. 2, pp. 125–137, 2012.

[23] S. Vömel and J. Stüttgen, "An evaluation platform for forensic memory acquisition software," *Digital Investigation*, vol. 10, pp. S30–S40, 2013.

[24] B. Schatz, "Bodysnatcher: Towards reliable volatile memory acquisition by software," *digital investigation*, vol. 4, pp. 126–134, 2007.

[25] A. Savoldi and P. Gubian, "Blurriness in live forensics: An introduction," in *Advances in Information Security and Its Application*. Springer, 2009, pp. 119–126.

[26] G. G. Richard and A. Case, "In lieu of swap: Analyzing compressed ram in mac os x and linux," *Digital Investigation*, vol. 11, pp. S3–S12, 2014.

[27] phatbuck...@gmail.com, "Issue 124:add plugin to dump dns resolver cache," https://code.google.com/p/volatility/issues/detail?id=124#c12, 2013, [Online; accessed 09-Feb-2015].