available at www.sciencedirect.com

ScienceDirect

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

Digital Investigation

Locating ×86 paging structures in memory images

Karla Saur[*], *Julian B. Grizzard*

The Johns Hopkins University, Applied Physics Laboratory, 11100 Johns Hopkins Rd, Laurel, MD 20723-6099, USA

ARTICLE INFO

Article history:
Received 16 November 2009
Received in revised form
26 March 2010
Accepted 13 August 2010

Keywords:
Digital memory forensics
×86
Process identification
Linux
Microsoft Windows
Paging
Virtualization

ABSTRACT

Digital memory forensics consists of analyzing various components of a memory image from a compromised host. A memory image consists of data and processes that were running on the system at the time the image was created. Previously running processes are one of the key items in memory images to identify, including potentially hidden processes. Each process has its own paging structures that define its address space, so locating the paging structures can potentially lead to finding all of the processes that were running. In this paper, we describe an algorithm to locate paging structures in a memory image of an ×86 platform running either Linux or Windows XP. The algorithm can be used to find paging structures for potential processes that were hidden by rootkits or other malware. Furthermore, if the system was running an ×86 virtual machine, the algorithm can locate paging structures associated with both the host kernel and the guest kernel processes. Our algorithm relies more on the constructs of the ×86 hardware and less on the operating system running on top of the hardware. This means that the algorithm works for many different operating systems with only minor tweaking.

1. Introduction

Volatile memory in a compromised system may contain critical evidence relevant to an attack that will be lost if the machine is powered off. To prevent this loss, an analyst can take a snapshot of the memory before shutting the system down. This snapshot will likely include both benign user data as well as malicious attacker data. For example, the attacker may have left a process running that contains details of how the attacker broke in and what he or she did after obtaining access. One important problem to solve in digital memory forensics is how to locate all of the processes that were running at the time the snapshot was taken, including malicious processes. With paging enabled, each of these processes must run in an address space described by a set of paging structures. Therefore, as a method to help locate all of the processes in a memory image, this paper focuses on finding paging structures that can lead to identifying processes.

To locate paging structures in a memory image, the analyst must understand how to interpret the raw bytes in the image. The semantics of a compromised memory image depends on the layers in the previously running system (hardware, operating system, and applications) and the malicious code injected into the system. One approach to locate paging structures is to use the details of the operating system kernel implementation. For example, because the kernel maintains all of the paging structures, a program could inspect the kernel data structures to locate all of the paging structures. This could also be used to enumerate the processes, which is what we are eventually interested in discovering. However, since the memory image is from a compromised host, there is a risk that even the kernel and its data structures have been compromised. Therefore, a more robust approach for locating

[*] *Corresponding author.*
E-mail addresses: ksaur@umd.edu (K. Saur), julian.grizzard@jhuapl.edu (J.B. Grizzard).

1742-2876/$ – see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2010.08.002

paging structures is to use specific details of the software–hardware interaction that even malicious code must maintain.

This paper describes an algorithm that uses the latter approach. We describe an algorithm for finding paging structures in a memory image by searching for data structures that are used by the hardware to support the processes. Specifically, we focus on the paging structures in the ×86 architecture. However, we do use some of the characteristics of the kernel implementations to optimize the search.

The ×86 hardware supports a mapping from what is known as virtual memory addresses to physical memory addresses (addresses on the physical RAM chips) so that each process running on the system can have its own virtual address space. When a machine instruction references a memory location, the hardware automatically translates the virtual address to a physical address using the previously configured paging structures in the physical memory. This process is called paging and must be enabled by the operating system. An important assumption to point out is that we assume that all code running on the system was setup to use paging, which is true in the majority of common operating systems. This limitation is discussed in Section 9.

Each process must have its own set of paging structures and those structures define the physical memory that is used by that process, enabling our algorithm to potentially locate all of the processes that were running on the system by finding all of the paging structures. In fact, the algorithm can potentially help identify some previously running processes if the operating system did not clean up the paging structures. In our experiments, our algorithm is able to find the paging structures associated with all running processes on both Linux platforms and Windows XP platforms.

In this paper, we first provide background information on ×86 paging. Next, we describe our algorithm for process identification in memory images of a Linux system and then explain how our algorithm can be generalized to different operating systems. We show the results of verifying our algorithm with clean memory images and then show the results of applying our algorithm to find potential rogue processes and to separate a virtual machine's paging structures from a host's paging structures. Finally, we discuss important limitations, related work, future work, and end by highlighting conclusions from our results.

2. Background on ×86 paging

On ×86 computers, the most common method of managing processes is to use the paging mechanism provided by the hardware. The paging mechanism provides a mapping from the virtual address space to the physical address space. For example, the operating system could set up a process so that a virtual address of 0xC0000000 maps to a physical address of 0x0003A000. Paging enables a number of benefits for processes to include: separate address spaces, shared mappings of memory, isolation, a virtual view of memory, swapping to disk, and so on. Commodity ×86 operating systems almost always enable paging for process management.

There are different types of paging structures that are legitimate depending on whether the ×86 system uses normal 32-bit paging or Physical Address Extension (PAE) paging (Intel Corporation, 2007). These structures are as follows:

- *Page Directory Pointer Table (PAE only)* – a 32-byte structure containing up to 4 pointers to page directories
- *Page Directory* – a 4 KB structure containing up to 1024 entries pointing to either a page table or a physical page (2 MB physical page if PAE or 4 MB physical page if non-PAE)
- *Page Table* – a 4 KB structure containing up to 1024 pointers to 4 KB physical pages
- *4 KB page, 2 MB page (PAE only), or 4 MB page (non-PAE only)* – translated physical page
- *CR3 Register* – stores the base physical address of the page directory pointer table (PAE only) or the page directory (non-PAE only) for the current process.

In 32-bit paging, the mapping is defined by the CR3 register value, a page directory, and the corresponding page tables which then point to physical pages of the process' data as seen in Fig. 1. For paging that uses 4 KB pages, a virtual address is divided into three ranges: high bits, mid bits, and low bits. The high bits provide an index into the page directory. The mid bits provide an index into the corresponding page table, and the low bits provide an offset into the corresponding page. For 4 KB pages, the page directories, page tables, and pages are all 4 KB in size and must be aligned on 4 KB boundaries. The physical address of the current page directory provides the root of a mapping and is stored in the CR3 register during execution of a process. The physical addresses of the page tables are stored in the page directory, and the physical addresses of the pages are stored in the page tables. For 4 MB pages, there are no page tables, so instead, the page directories point to 4 MB pages. Each process has its own CR3, which enables each process to have a different virtual to physical memory mapping.

Physical Address Extension (PAE) mode maps 32-bit virtual addresses to 36-bit physical addresses, enabling up to 2^{36} bits (64 GB) of RAM. PAE requires an additional layer of abstraction called the page directory pointer table. The address of the page directory pointer table is stored in the CR3 register during execution. The page directory pointer table has 4 entries which point to the process' 4 page directories. Each page directory pointer table is 32 bytes in length (4 × 8-byte entries) and must be 32-byte aligned within physical memory. PAE uses 2 MB large pages instead of 4 MB page pages.

From a forensics perspective, it is interesting to find the paging structures in a memory image because each set of paging structures will correspond to a process that was either currently running or previously running for a given memory image. In 32-bit systems without PAE-mode enabled, the

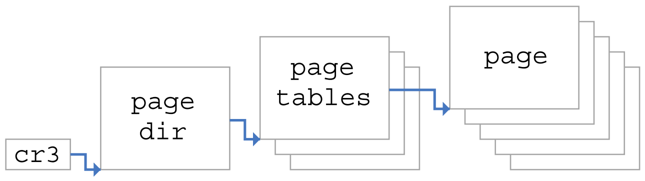


Fig. 1 – ×86 linear to physical address translation.

important structures to locate are the page directories and page tables. In 32-bit systems with PAE-mode enabled, the important structures are the page directory pointer tables, the page directories, and the page tables. Although we focus on 32-bit systems in this work, our algorithm generalizes to other architectures including 64-bit ×86, virtual page tables provided by recent ×86 hardware virtualization support, and even non-×86 platforms.

3. Related work

Linux memory analysis was part of the focus of the Digital Forensics Research Workshop (DFRWS, 2008) Forensics Challenge in 2008, in which participants analyzed a Linux memory image among other pieces of evidence from a Linux system. Two of the challenge submissions made use of the tool PyFlag to automate the analysis of the provided Linux memory dump.

Presented at an earlier DFRWS Conference was FATKit, a versatile memory forensics toolkit that included the capability to parse out Linux process lists using Linux kernel structures (Petroni et al., 2006). At Black Hat, Burdach (2006) presented a Linux memory analysis tool using Linux internal structures. In our Linux process list construction, we focus on the hardware structures rather than the Linux kernel structures.

Kornblum (2007) makes use of page directory entry flags, applying Windows XP's utilization of the "available bits" 10–11 to search additional pages that are flagged as "invalid", gaining additional information about these otherwise overlooked pages in memory. Like this paper, we focus on paging-related flags; however we utilize only the flags as they are explained in the Intel Corporation (2007) IA-32 manual, ignoring bits 9–11, which are left available by Intel for operating system specific use.

One of the earliest tools to scan for virtual address spaces in memory images was a script called pmodump that was developed as part of the Truman project (Stewart, 2010). The pmodump script is based on the observation that page directories on Microsoft Windows platforms contain a self-referencing physical address pointer at a known offset in the page. The tool is fast because it only checks the value of one double word for each page to determine whether or not that page is a page directory. The main advantage of this algorithm is speed. However, one of the main disadvantages is that the algorithm is specific to the operating system.

Schuster (2006) creates a set of scanning rules based on the structure of Microsoft Windows processes and threads to locate processes and threads. The process structure information includes a pointer to the paging structures for that process, which can be used to map out the virtual address space for that process. Schuster (2007b,c) later considers searching for processes in the reverse direction by first finding paging structures. He develops a method to find page directories by testing to see if the upper portion of a potential page directory matches an expected pattern. His method has some false positives, but it is not clear what is causing the false positives (Schuster, 2007a).

Wu (2007) focuses on finding page directories for Windows XP SP2 with PAE enabled. Wu notes that the fourth entry in the page directory pointer table points to itself, so that the paging structures can be mapped into the virtual address space. Therefore, the test to determine if a given page is a page directory pointer table is to see if the fourth entry contains the physical address of that page.

Previous work has focused on developing patterns to find paging structures based on observations about specific operating systems. The pmodump tool's method and Wu's method for locating paging structures are fast because they only inspect a small amount of data per page. Although these algorithms are fast, there is less data that needs to be modified in order to confuse their algorithm. Schuester has a more robust method because he has developed a signature for the upper portion of the address space, but his method is also operating system specific. In this work, we are interested in developing a method to find paging structures that is less operating system dependent. Instead, our method focuses on general ×86 constraints on paging structures as much as possible.

4. Linux algorithm to locate processes

Following the Intel Corporation (2007) 64 and IA-32 Architecture Software Developer's Manual, we made predictions about the flag values of the page directory entries belonging to the kernel. Our plan was to use these flag values as signatures for valid page directories, as all valid page directories will have the same kernel mappings. For example, we expected the kernel pages to be flagged at the supervisor privilege level and for the read/write flag to be set to allow reading from and writing to the kernel pages, and we expected the page base address bits of the page directory entry to be within the range of physical memory if the present flag indicated the page was present. After making predictions for all of the register values, we then refined and verified our predictions by inspecting the kernel mapping's flags in known valid page directories using a JTAG debugger (Arium, 2010) to view the actual flag values. We then used these flag values as a signature in our model of a page directory.

Our basic algorithm iterates through the entire memory image using a simple scanner. We search the entire memory image one page directory-sized block (4 KB) at a time, treating each block as a potential page directory. The page of memory is rejected as a valid page directory if it has a characteristic that is not consistent with our model of a valid page directory. Because PAE mode uses a page directory pointer table, we must use a slightly different algorithm for PAE-enabled images. These differences are described in Section 5.2. We will first describe our algorithm as applied to Linux in non-PAE mode, and then show how changing parameters can easily expand the algorithm to other platforms. Our algorithm uses two passes to rule out possible page directories as described below.

4.1. *First pass of algorithm*

First we search the potential page directory for kernel mappings. If the physical memory size is less than 896 MB, the Linux kernel maps all of physical memory into the virtual address space starting at 0xC0000000, the fourth gigabyte of the kernel linear address space. Memory above 896 MB cannot be mapped entirely into the kernel linear address space (Bovet and Cesati, 2006), but this does not affect our algorithm.

In the Linux kernel, kernel mapping flags correspond to a pattern where the least significant bits are equal to “0x1E3”. The physical memory is mapped in as 4 MB pages with the flags as shown in Fig. 2. We chose to focus on the hardware flags because it is a quick and consistent way to identify the kernel mappings which correspond to the location of potential processes in physical memory. Although all page directories in a memory image contain pointers to the same kernel and will therefore have the same base addresses, the kernel location and address will change for different platforms while the flag values remain the same. The page directory entry flags for non-PAE Linux are as follows:

- Flag 0, the “present” flag, is always 1 because the kernel will always be mapped in and present.
- Flag 1, the “read/write” flag, is set to 1 to allow the kernel to read and write.
- Flag 2, the “user/super” flag, is always 0 (super), because only the kernel should have access.
- Flag 3, the “write-through” flag, is 0 to allow write-back caching.
- Flag 4, the “cache disable” flag, is 0 to allow the page table to be cached.
- Flag 5, the “accessed” flag, is 1 showing that the page or page table has been accessed.
- Flag 6, the “dirty/clean” flag, is 1 showing that the page has been written to.
- Flag 7, the “page size” flag is 1 in Linux systems, as Linux uses large pages to map in the kernel. (Large pages are 4 MB in non-PAE mode and 2 MB in PAE mode.) Windows XP uses both 4 KB and large pages, but the majority of page directory entries in Windows XP kernel space are 4 KB.
- Flag 8, the “global” flag, is 1 in all Linux systems and is unpredictable in Windows XP systems.

All valid page directories in Linux will contain a number of entries whose least significant bits are equal to “0x1E3”. The number of these entries is slightly less than 1/4 times the physical memory size in megabytes because the kernel uses 4 MB pages for its mappings. The flags remain the same across PAE enabled and non-PAE-enabled Linux systems. This has been successfully tested on many Linux distributions across varying memory sizes, as shown in Table 2.

Page directories can hold up to 1024 (0–1023) entries. For each potential page directory, we count the number of entries that match the kernel mapping flag pattern located in the top 1/4 of the page directory (page directory entries 768–1023), which correspond to the kernel’s virtual address space. We then compare the counted number to the threshold of expected entries, which we set to slightly less than 1/4 times the physical memory size as shown in Table 1. If there are not enough kernel mappings as determined by flag pattern count, then we disregard the potential page directory and move on to the next 4 KB block of memory.

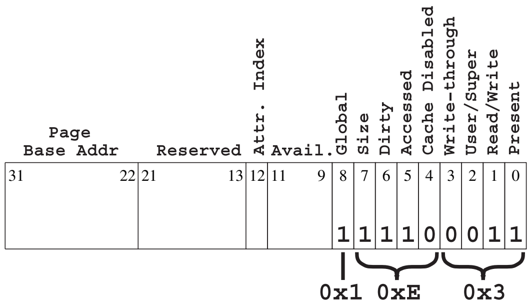


Fig. 2 – Page directory entry values for a 4 MB Page (Linux kernel mapping).

In this initial pass, we also count the number of populated entries in the page directory. If a non-PAE potential page directory has no entries (and is therefore entirely blank), or 1024 entries (and is therefore entirely garbage, because valid page directories contain some blank entries), we discard this potential page directory. This allows us to quickly rule out potential page directory memory segments without further processing.

4.2. *Second pass of algorithm*

Once the kernel mappings have been detected, we further inspect the potential page directory by looking at the userspace (0–767) entries. If the “present” bit of an entry (bit 0 in Fig. 2) is set to 0, we continue to the next entry and ignore the rest of the current entry, as the page it points to has been swapped out of physical memory. If bit 0 is set to 1, indicating a present entry, we further analyze the page directory entry, this time checking that the page base address or the page table base address is in a valid address range within physical memory. If the potential page directory passes this final check, we mark it as a valid page directory. This valid page directory represents another potential process detected by our algorithm, and it is added to our list of potential processes that the analyst can inspect.

5. Generalized algorithm

We have explained how paging structures can be located in a memory image using the kernel mappings in Linux non-PAE systems. As the introduction states, this algorithm can be applied to any ×86 system implementing paging by making a few minor modifications. These modifications are made by adjusting flag values and threshold count as shown in Table 1. Future work will investigate flag values and threshold count for other platforms such as Mac OS X, newer Windows versions, and 64-bit operating systems. We will now show how our algorithm can be applied to Microsoft Windows XP and PAE-enabled Linux and Windows XP.

5.1. *Algorithm applied to Microsoft Windows XP*

In a Windows XP system, the algorithm is similar with an adjustment to the “0x1E3” flag pattern. Instead of using only large pages to map in the kernel like Linux, Windows XP uses both 4 KB and large pages to map in the kernel (Russinovich and Solomon, 2005). However, the majority of kernel mapping entries are 4 KB, and for simplicity, we change the “page size” bit (Flag 7) from 1 to 0 in our matching pattern.

Also, the “global” flag (Flag 8) varies in Windows XP. According to the Intel Corporation (2007) manual, Flag 8 is the “global” flag in non-PAE and 2 MB PAE entries; in 4 KB PAE entries, there is no “global” flag, and Flag 8 is always set to 0.

Table 1 – Algorithm parameters for different operating systems.

Platform	Flag values	Flag threshold count	Flag location[a]
Linux (Non-PAE)	0x1E3	$\approx \max(\frac{1}{4} * (\text{mem\_size MB}), \frac{1}{4} * 896)$	pg_dir entries 768–1023
Linux (PAE)	0x1E3	$\approx \max(\frac{1}{2} * (\text{mem\_size MB}), \frac{1}{2} * 896)$	pg_dir 3
Windows (Non-PAE)	0x63	(350–500)	pg_dir entries 512–1023
Windows (PAE)	0x63	(350–500)	pg_dir 2 and 3

a Zero indexed (page directories entries from 0 to 1023, PAE page directories from 0 to 3).

However, observations show that Windows XP PAE 4 KB page table entries may be either 1 or 0.

Because of these variations, we ignore Flag 8, and only invert Flag 6. From this change, the least significant bits we search for in Windows XP systems are “0x63”. The Windows XP default settings map in the kernel above 0x80000000, using around 400 × 4 KB page directory entries. Therefore, we search for “0x63” flag entries in the top half of the page directory.

Because Windows XP uses a large number of 4 KB pages to map in the kernel, the “0x63” entries in the page directory are pointers to page tables (rather than directly pointing to 4 MB kernel pages), therefore the number of mappings does not vary greatly with the amount of memory because we are not directly considering the *pages*, only the page directory entries, which are pointers to page *tables*. Note that the number 400 is an observed approximate and further research is necessary to carefully bound this number. These variations are shown in Table 1.

5.2. *Algorithm applied to 36-bit PAE*

In PAE mode, the CR3 value points to a page directory pointer table, which points to four page directories, adding an additional level of paging structure. The above parsing algorithm is similar for PAE mode, except that we now consider each block of memory as a potential page directory pointer table with pointers to four corresponding page directories. According to the Intel Corporation (2007) specification, these page directory pointer tables must be 32-byte aligned, rather than page-aligned like the page directories, so we adjust our scanning increment accordingly. In Windows XP, the last two page directories contain the kernel mappings, corresponding with 0x80000000 and higher in the virtual address space. In Linux, the last page directory contains the kernel mappings, corresponding with 0xC0000000 and higher. Because the Linux kernel maps into 2 MB pages in PAE mode rather than 4 MB pages, there are around twice as many “0x1E3” values, or around half of the memory size, as opposed to one-quarter the memory size in non-PAE images. For example, the kernel maps in to 255 entries (“0x1E3” entries per page directory) in a 512 MB Ubuntu Server 8.04 (PAE) memory image, but only 127 entries in a 512 MB Ubuntu Desktop 8.04 (non-PAE) image. Table 1 explains the different parameters we input to our algorithm allowing for the different modes. Note that the flag threshold count is approximate.

6. Verifying algorithm with clean images

In order to determine if the paging structures we found correlate with processes on the system, we compared our list to process lists reported by known clean operating systems.

6.1. *Linux verification*

We walked the Linux kernel task structures to verify our findings. We tested on a variety of system configurations as shown in Table 2. In non-infected memory images, our algorithm did not miss any processes listed in the task structure lists. We also detected several additional process spaces. Many of the unlisted process spaces we found were terminated processes, which we verified using multiple image captures over a period of time. In many cases we are able to distinguish terminated processes from potentially malicious processes, as will be described in Section 7.

6.2. *Windows XP verification*

To verify our algorithm on Windows XP images, we acquired memory dumps of clean systems for both PAE-enabled systems and non-PAE-enabled systems. We obtained a list of running processes using a kernel debugger (Microsoft Corporation, 2010) and their corresponding paging structures, and we compared this list to our list of paging structures. Our algorithm did not miss a single process space as shown by the debugger. Our algorithm also detected several other process spaces with

Table 2 – Example test results for clean memory images.

Distribution	Kernel Vers.	Size (MB)	Mode	CR3s	Missed[a]	Expired[b]
Centos 5.1	2.6.18-53.el5	512	Non-PAE	43	0	3
Fedora 7	2.6.21-1.3194.fc7	256	Non-PAE	93	0	6
Ubuntu Server 6.06.2	2.6.15-51-server	128	PAE	30	0	10
Ubuntu Server 8.04	2.6.24-19-server	128	PAE	25	0	7
Ubuntu 8.04	2.6.24-19-generic	512	Non-PAE	88	0	12
Windows XP SP1	N/A	512	Non-PAE	68	0	N/A
Windows XP SP2	N/A	256	PAE	61	0	N/A

a Number of process spaces present in the kernel structures but missed by algorithm.
b Processes marked as expired as described in Sections 7.1 and 7.2.

kernel mappings similar to the verified processes, including several copies of valid processes. The extraneous processes that are not copies could possibly be due to idle or terminated processes and more investigation of these processes is left to future work.

7. Rogue process detection

Now that we have outlined our basic process space detection algorithm, we show how it can be applied. Our algorithm can be used to detect rogue processes hiding in memory by comparing a list of expected processes to a list of potential processes as determined by our algorithm. In many cases, we are able to weed out terminated processes enabling us to more easily spot malicious "rogue" processes attempting to hide in memory.

7.1. *Rogue processes in Linux without PAE*

In a current valid page directory on Linux systems without PAE enabled, "entry 0" of the 1024 possible entries always has all 32 bits set to "0". When a process terminates, the Linux kernel will free the page directory and add it to the kernel's linked list of free pages of memory. The pointer to the next page of the kernel's free page list is stored in the first 4 bytes of each page, thereby making the old "entry 0" non-zero. This non-zero "entry 0" allows us to quickly label the page directory as terminated. Additionally, all other userspace entries are cleared, but the kernel mappings are left in tact.

Comparing our list of paging structures to the Linux kernel's task structure list allows us to spot unexpected paging structures. Using the above information, we can identify former processes that have terminated and been removed from the task structure list by the operating system. This allows us to easily distinguish former processes (that were legitimately removed from the task structure list) from potentially malicious or rogue processes (that were removed from the task structure list by malware). When we find a page directory, as detected by our algorithm, we expect it to be one of the following:

- Present in the kernel's copy of the task structure list
- Consistent with the pattern of a terminated task, having only the kernel mappings and a pointer in entry 0
- The master kernel Page Global Directory, also know as swapper_pg_dir (One instance per memory image found low in physical memory) (Bovet and Cesati, 2006)

If there are other page directories found by our algorithm that do not match one of those three groups, they are flagged as potentially malicious. We use this algorithm to detect rootkits, as discussed in Section 7.4.

Although our algorithm can detect terminated Linux processes, unfortunately for forensics purposes, all userspace entries (1–767) of the page directory and the pages they point to are zeroed out promptly after a process expires. This leaves the former page directory as entirely zeros with the exception of a pointer to the next page of free memory in entry 0, and the kernel mappings at entry 768 and higher (the top 1/4 of the 1024 page directory entries).

7.2. *Rogue processes in Linux with PAE*

Our algorithm for Linux PAE systems is similar to non-PAE systems. In PAE systems, after a process has expired, the Linux kernel zeros out all user entries in the first three page directories pointed to by the page directory pointer table. However, the page directory pointer table (a 32-byte structure) is not deleted, leaving 3 pointers to former page directories and the kernel page directory. Our algorithm therefore detects several paging structures with zero userspace entries, which are remnants of former processes.

In newer Linux kernel versions, (for example, the kernel used by Ubuntu 8.04), occasionally two copies of a page directory pointer table exist. This is a remnant of a former process and the Linux kernel has reassigned the page directories from the expired process to a new process. The only remnant of the old process is the copy of the page directory pointer table, with all of the page directory and page pointers overwritten by the newly assigned process.

Once we rule out terminated processes, we can again search for suspicious paging structures. To be non-suspicious we expect the page directory pointer table to be one of the following:

- Present in the task structure list
- Consistent with the pattern of a terminated task as described above
- Be a copy of a legitimate process: the page directory pointer table's first three page directories are reused in an exact copy of a page directory pointer table in the task structure list

If there are other page directory pointer tables found by our algorithm that do not match one of those three groups, they are flagged as potentially malicious, just as they are for non-PAE. Test results of this are discussed in Section 7.4.

Because the userspace page directory entries have been zeroed out or overwritten by a new process, no data from the process' pages is available, which makes forensic analysis difficult.

7.3. *Rogue processes in Windows XP*

In Microsoft Windows XP non-PAE systems, all parsed page directories should be a member of the Windows XP process list or an exact copy of a legitimate page directory. Parsed paging structures not found in the Windows XP process list are suspicious.

In PAE Windows XP, after a process has been terminated, the page directories are zeroed out, clearing all of the pointers to the pages of data. (Note that this does not necessarily include the page content data, just the paging structure of pointers to the data.) The page directory pointer tables are left partially intact, with only the address of page table 0 being overwritten.

On a clean system, all of the page directory pointer tables of valid processes will be found using our algorithm, along with several terminated processes and duplicate page directory pointer tables of valid processes, where the page directory pointer table points to the same 4 page directories as a valid page directory pointer table as mentioned in Section 6.2. In PAE Windows XP systems, there is significantly more noise (copies, terminated processes) than in a typical Linux system; however, it is still possible to spot rogue paging structures not conforming to the copy or terminated process model with an algorithm similar to the Linux algorithm in Section 7.2. Additional research in Windows XP systems is needed to finalize this algorithm and is left to future work.

7.4. *Detecting processes hidden by Linux rootkits*

To test our code against real-world attacks, we investigated kernel-level rootkits. Kernel-level rootkits modify running kernel code and data structures so that an attacker can hide malicious processes. We tested our algorithm on several Linux kernel-level rootkits to verify that we could locate the address spaces associated with the hidden processes. The tested rootkits used several different methods for modifying the kernel including redirecting system calls, unlinking tasks from lists, and hooking the Virtual File System (VFS).

For each test, we took two images of the same system: a clean image and an infected image. We created the first image by launching an unhidden “malicious” process without the target rootkit installed, allowing us to record the CR3 of the “malicious” process for verification later. The image is considered clean even with the “malicious” process running because we created the “malicious” process for testing purposes only, and all the process does is print a message once per second. After creating the clean image, we infected the system with the target rootkit and hid the “malicious” process. Then, we took a second snapshot of the memory to create the infected image. After creating both images, we tested our tool to determine if we could locate the hidden process in the infected image.

One of the rootkits we tested was the Enyelkm rootkit (2010). This rootkit can be used to hide files, processes, and directories by modifying the system call table. In particular, it modifies the getdents system call that is used by the ps utility to list processes. The getdents system call lists directory entries, and ps uses the getdents system call to list the currently running processes as defined in the /proc directory.

To test the Enyelkm rootkit, we created a clean image and an infected image as described above. We recorded the CR3 for the “malicious” process in the clean image and results showed that we were able to find the paging structures for a process with a matching CR3 for both the clean image and the infected image. In the infected system, the ps utility did not show the “malicious” process as a process that was active on the system. However, our algorithm was able to locate the “malicious” paging structures in the infected image. We also walked the kernel’s list of tasks in the infected image and found the “malicious” process. To increase the sophistication of the rootkit, we enhanced it by removing the “malicious” process from the kernel’s list of task structures. Additional functionality would need to be included with the rootkit to ensure the process is properly scheduled, but for our purposes, this modification is sufficient to test our algorithm. After adding the patch, we created two new images and ran our tool on the infected image in a consistency-checking mode. This mode compares the process list found from walking the kernel’s task structures against the process list found using our algorithm. The results of this test are shown below:

```
...
Match: 0x0DA39000 – dd. terminated = F
Match: 0x0DA15000 – sshd. terminated = F
WARNING!: process at 0x0D9BB000; no match.
  # userspace entries: 3 terminated = F
Match: 0x0D959000 – sshd. terminated = F
Match: 0x0D954000 – hald. terminated = F
Match: 0x0D8AB000 – sshd. terminated = F
Process at 0x0D888000; no match.
  # userspace entries: 1 terminated = T
Match: 0x0D87A000 – bash. terminated = F
...
```

The results show information for each potential CR3 value located in the image using our algorithm. If a corresponding CR3 is found in the kernel’s list of tasks as well, then “Match” is printed along with the process name. For example, the “sshd” process was found with a CR3 value of 0x0DA15000. If a corresponding CR3 is not found in the kernel’s list of tasks and the process does not appear to be terminated (terminated = F), then a warning message is printed. In this case, the warning message accurately locates the hidden process with a CR3 value of 0x0D9BB000.

We repeated our tests with other process-hiding rootkits including Intoxonia and Override. Both Intoxonia and Override modify the system call table to hide processes. We were also able to find the paging structures for processes hidden by these rootkits.

8. Separating VM host and guest processes

Modern ×86 platforms support virtualization, which can help users run multiple operating systems on the platform at the same time, but it can also help attackers isolate their malicious processes. The ability to detect and analyze the use of virtualization in memory images has become increasingly more important as virtualization becomes more and more widespread (Gartner). Since we are interested in locating all paging structures for processes in a given memory image, it is important for us to consider how our algorithm will work on memory images containing virtual machines. We can apply our same algorithm to find processes associated with the virtual kernel and host kernel.

8.1. *Kernel-mapping size distinguishes processes*

If a virtual machine is running during the creation of a memory image, the virtual machine’s processes will also be present in the physical memory image. Because the virtual machine’s kernel is likely to be a different size than the host’s kernel, it is very simple to distinguish between the two kernels because of the kernel mapping size. Our algorithm can locate both sets of paging structures in one run if the flag threshold is

Table 3 – Partial output of processing a Linux memory image with a running Linux virtual machine (VMWare).

Physical address[a]	Num. 0x1E3 entries	Num. userspace entries
0x1EDE1000	127	48
0x1E80F000	127	9
0x1D83B000	127	10
0x1D833000	127	3
0x1D5E2000	79	6
0x187D4000	79	15
0x10008000	79	8
0x05B08000	79	19

a Physical address is base of page directory (CR3).

adjusted lower. If virtualization is suspected for a given memory image, in many cases running the algorithm with a lower flag threshold count will locate the guest kernel's paging structures. For example, in Table 3, a non-PAE Ubuntu 8.04 host with 512 MB of memory has 127 0x1E3 kernel mappings, while the non-PAE Ubuntu 6.10 VMWare guest with 320 MB of memory has 79 0x1E3 kernel mappings. The mapping count will be identical for all processes belonging to a particular Linux kernel; the mapping count will be similar for processes belonging to a particular Windows XP kernel.

8.2. *Fingerprinting operating systems*

Another advantage of using ×86-based detection mechanisms is the ability to simultaneously locate paging structures for different operating systems. In addition to being able to separate paging structures by kernel size (mapping count), the flag values may differ as well. For example, we can obtain separate paging structures for a Linux host running a Windows XP virtual machine. Not only will the flag entry counters be different for the separate kernels, the flags will have different values between Windows XP and Linux systems. An example of this is shown in Fig. 3 where a non-PAE Linux host contains a PAE Windows XP SP2 guest in VMWare (2010).

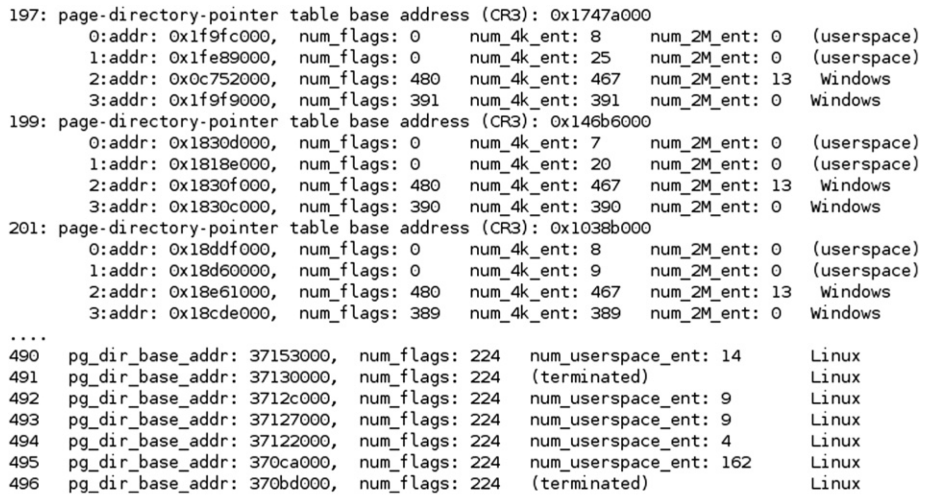
The case where one kernel is non-PAE and another is PAE requires an extra step. Analyzing the image in non-PAE mode (4 KB scans) will find all page directories, including each of the four page directories belonging to page directory pointer tables for PAE mode. To find the page directory pointer tables, a second pass through the memory image must be performed, scanning at the page directory pointer table size (32 bytes).

9. Limitations of the algorithm

There are a number of important limitations that apply to our algorithm. We describe these limitations below and suggest enhancements to our algorithm to mitigate these limitations.

The first limitation we consider is that we make some assumptions about when and how page tables are laid out in memory. For example, we assume that there are no false page directories that have data similar to the expected hardware flags (0x1E3/0x63) in the correct positions. This noise would introduce false positives into our algorithm. We could enhance our algorithm to further inspect targets of potential page tables to help eliminate these false positives. On the other hand, we also assume that legitimate paging structures are intact such that none of the 0x1E3/0x63 entries have been modified. However, if these flags have been modified, then much of the underlying system must also be modified to properly maintain a stable system. Furthermore, in such a case, we could loosen the constraints in our algorithms to minimize the false negatives.

The second limitation we consider is that we assume paging will be turned on and in use for all running processes. If a process does not have paging structures associated with it, then this algorithm will not be able to locate it. It is certainly possible to create a process that runs on the ×86 platform with paging turned off, but this is much more difficult than using the existing support for process creation, especially when the



```
197: page-directory-pointer table base address (CR3): 0x1747a000
       0:addr: 0x1f9fc000,  num_flags: 0     num_4k_ent: 8     num_2M_ent: 0   (userspace)
       1:addr: 0x1fe89000,  num_flags: 0     num_4k_ent: 25    num_2M_ent: 0   (userspace)
       2:addr: 0x0c752000,  num_flags: 480   num_4k_ent: 467   num_2M_ent: 13   Windows
       3:addr: 0x1f9f9000,  num_flags: 391   num_4k_ent: 391   num_2M_ent: 0   Windows
199: page-directory-pointer table base address (CR3): 0x146b6000
       0:addr: 0x1830d000,  num_flags: 0     num_4k_ent: 7     num_2M_ent: 0   (userspace)
       1:addr: 0x1818e000,  num_flags: 0     num_4k_ent: 20    num_2M_ent: 0   (userspace)
       2:addr: 0x1830f000,  num_flags: 480   num_4k_ent: 467   num_2M_ent: 13   Windows
       3:addr: 0x1830c000,  num_flags: 390   num_4k_ent: 390   num_2M_ent: 0   Windows
201: page-directory-pointer table base address (CR3): 0x1038b000
       0:addr: 0x18ddf000,  num_flags: 0     num_4k_ent: 8     num_2M_ent: 0   (userspace)
       1:addr: 0x18d60000,  num_flags: 0     num_4k_ent: 9     num_2M_ent: 0   (userspace)
       2:addr: 0x18e61000,  num_flags: 480   num_4k_ent: 467   num_2M_ent: 13   Windows
       3:addr: 0x18cde000,  num_flags: 389   num_4k_ent: 389   num_2M_ent: 0   Windows
....
490   pg_dir_base_addr: 37153000,  num_flags: 224   num_userspace_ent: 14       Linux
491   pg_dir_base_addr: 37130000,  num_flags: 224   (terminated)                Linux
492   pg_dir_base_addr: 3712c000,  num_flags: 224   num_userspace_ent: 9        Linux
493   pg_dir_base_addr: 37127000,  num_flags: 224   num_userspace_ent: 9        Linux
494   pg_dir_base_addr: 37122000,  num_flags: 224   num_userspace_ent: 4        Linux
495   pg_dir_base_addr: 370ca000,  num_flags: 224   num_userspace_ent: 162      Linux
496   pg_dir_base_addr: 370bd000,  num_flags: 224   (terminated)                Linux
```

Fig. 3 – This partial output contains the analysis of a 1 GB memory image with a Non-PAE Linux (Ubuntu 8.04) host containing a 512 MB PAE Windows (XP SP2) guest virtual machine. The Windows XP entries show the page directory pointer table and four corresponding page directories. The Linux entries marked "terminated" have a pointer in entry 0 and no other userspace entries.

system must support both processes with paging turned on and processes with paging turned off. Furthermore, it may be possible to detect processes that are running with paging turned off using a different algorithm.

The third limitation we consider is that although our algorithm relies mostly on ×86 structures, we use some information about the running operating system to enhance our algorithm. This means that not all of our algorithm's features will apply to every memory image. For example, Red Hat, Fedora, and similar derivatives often put a pointer in entry zero of the page directory even when the process has not been deleted or removed from the task structure list, so our method for detecting potential deleted processes will not work correctly on these systems. This case provides less information about whether or not the paging structures belongs to a former process; however, the more general parts of our algorithm will still apply.

10. Future work

10.1. *Hypervisor detection*

From a security and forensics perspective, virtualization detection is becoming increasingly important because of the concept of hypervisor rootkits, where a hypervisor hides itself from the running instance of the operating system (Rutkowska, 2006; Zovi, 2006). Several research groups propose that hidden hypervisors can be detected by searching for ×86 structures present in physical memory (Ptacek a,b; Fritsch, 2008).

Assuming that an unaltered physical memory dump could be acquired by some external DMA (Direct Memory Access) means such as Firewire or PCI, our algorithm could potentially be applied to locate page structures showing the mappings of another kernel or hypervisor if it uses paging. For example, Bluepill, an ultra-thin hypervisor rootkit, takes 16 pages of memory (Rutkowska, 2007). If a hypervisor makes use of paging like Bluepill does, it should be possible to detect its private ×86 paging structures and distinguish them from its guest paging structures. Research about common hypervisor flag settings and number of expected page directory entries is left for future work.

10.2. *Expansion of current work*

The majority of our testing and fine-tuning focused around a Linux-based platform. We would like to further research how our algorithms can be applied to different versions of the Windows operating system. We have positive results from Windows XP, and future work would involve expanding our testing to Windows Vista and Windows 7 platforms. We would also like to expand to Mac OS X and other platforms implementing ×86 paging. Additionally, we would like to expand our rogue paging structures detection algorithms to other operating systems.

Also, our algorithms center around 32-bit operating systems. A natural expansion of our work would be to include 64-bit operating systems. This would include researching the kernel fingerprint in the page directory entries by observing the least significant bits on the flag values and adjusting our algorithm accordingly, and also analyzing additional levels of paging structures.

10.3. PAE – *speed of algorithm*

In a non-PAE image, parsing out paging structures takes a few seconds on a standard machine for moderate sized memory images (2 GB or less). But for PAE mode, we must now add an additional step to our memory parsing and also must analyze in 32-byte block increments (the size of a page directory pointer table) instead of 4096 byte increments (the size of a page directory), making it roughly 128 times slower, as the increments we analyze must be 128 (4096/32) times smaller. However, the approximate location of these page directories in memory are predictable, dependent on platform so a faster algorithm would need to attempt to leverage the expected page directory location information without incrementing the false negatives (misses), as malware may hide anywhere in memory.

10.4. *Additional testing*

Another goal would be to test our algorithms on more malware and rootkits. Many Linux kernel-level rootkits that hide processes operate by hooking the Virtual File System (VFS), and we have successfully demonstrated that our algorithm locates paging structures for processes hidden by this method as implemented in several different rootkits. We would like to expand on our rootkit and malware detection as rootkits continue to evolve.

11. Conclusions

We have described an algorithm that can locate paging structures in a memory image based on ×86 paging structures. The algorithm is adjustable for varied platforms and has been successfully tested on different versions of both Linux and Windows XP. Our algorithm can detect rogue paging structures for processes hidden in memory based on locating the paging structures. For memory images containing virtual guest operating systems, in many cases our algorithm can successfully determine the paging structures for each kernel. Using our algorithm, forensic examiners may be able to more easily discover complete process lists for a variety of memory images.

Acknowledgments

Thank you to Andrew Tappert, Tom O'Connor, and Sandy Ring of the Pikewerks Corporation (Pikewerks Corporation, 2010) for allowing us to collaborate on your Second Look™ project. Thank you to AFRL/RYT ATSPI Technology Office for sponsoring the Small Business Technology Transfer grant that funded the initial research of our algorithm. Distribution Statement A– Approved for public release: distribution is unlimited. Public release document number 88 ABW-10-1535.

The work upon which this publication is based was performed pursuant to AFRL Contract number FA8650-07-C-1205 under Subcontract Agreement to Pikewerks Corporation.

Thank you to technical staff of The Johns Hopkins University Applied Physics Laboratory for help installing the test rootkits and for the help with developing the algorithm for Linux PAE rogue paging structures detection.

REFERENCES

Arium. ARIUM – Intel JTAG debugger, http://www.arium.com/products/3/Intel-JTAG-Debuggers.html; January 2010.

Bovet DP, Cesati M. Understanding the Linux kernel. 3rd ed. Sebastopol, CA: O'Reilly Media, Inc.; 2006.

Burdach M. Physical memory forensics. USA: Black Hat; August 2006.

DFRWS. Digitial forensics research workshop. Forensics challenge, http://www.dfrws.org/2008/challenge/index.shtml; 2008.

Enyelkm rootkit, http://www2.packetstormsecurity.org/cgi-bin/search/search.cgi?searchval%ue=enyelkm.

Fritsch H. Analysis and detection of virtualization-based rootkits. Munchen: Technische Universitat; August 2008.

Gartner. Gartner says virtualization will be the highest-impact trend in infrastructure and operations market through 2012, http://www.gartner.com/it/page.jsp?id=638207.

Intel Corporation. Intel®64 and IA-32 architectures software developer's manual, vol. 3A. Denver, CO: Intel Corporation; 2007.

Kornblum JD. Using every part of the buffalo in Windows memory analysis. Digital Investigation 2007;4:24–9. December 2006.

Microsoft Corporation. Microsoft windows debugger (WinDbg), http://msdn.microsoft.com/en-us/library/cc266321.aspx.

Petroni N, Walters A, Fraser T, Arbaugh W. FATKit: a framework for the extraction and analysis of digital forensic data from volatile system memory. Digital Investigation December 2006; 4:197–210.

Pikewerks Corporation. Second look™, http://pikewerks.com.

Ptacek T. Side-channel detection attacks against unauthorized hypervisors, http://matasano.com.

Ptacek T. The ×86 memory system and why it's hard to virtualize securely, http://matasano.com.

PyFlag, http://www.pyflag.net.

Russinovich M, Solomon D. Microsoft Windows internals. 4th ed. Redmond, WA: Microsoft Press; 2005.

Rutkowska J. Subverting vista kernel for fun and profit. USA: Black Hat; August 2006.

Rutkowska J. IsGameOver() anyone? USA: Black Hat; August 2007.

Schuster A. Searching for processes and threads in Microsoft Windows memory dumps. Digital Investigation June 2006;3S: S10–6.

Schuster A. Memory analysis: copies of page directories, http://computer.forensikblog.de/en/2007/05/copies_of_page_directories%.html; 2007a.

Schuster A. Memory analysis: searching for page directories (1), http://computer.forensikblog.de/en/2007/05/searching_page_directories_1%.html; 2007b.

Schuster A. Memory analysis: searching for page directories (2), http://computer.forensikblog.de/en/2007/05/searching_page_directories_2%.html; 2007c.

Stewart J. Truman – The reusable unknown Malware analysis net, http://www.secureworks.com/research/tools/truman.html.

VMWare. VMWare server, http://www.vmware.com/.

Wu J. Search PDEs in memory dump of Windows XP SP2 with PAE, http://jingqiwu.blogspot.com/2007/12/search-pdes-in-memory-dump-of-windows%.html; 2007.

Zovi DD. Hardware virtualization-based rootkits. USA: Black Hat; August 2006.