Digital forensics of the physical memory

Mariusz Burdach
Mariusz.Burdach@seccure.net

Warsaw, March 2005
last update: July 11, 2005

Table of Contents

Abstract

This paper presents methods by which physical memory from a compromised machine can be analyzed. Through this methods, it is possible to extract useful information from memory such as: a full content of files, detailed information about each process and also processes that were being executed and then were terminated in the past. This paper aims to explain the concepts of digital investigations of volatile memory. Techniques covered by this paper will lead you through the process of analyzing important structures and recovering contents of files from physical memory.
In addition, a technique, that detects hidden User Mode processes, will be discussed in-depth. This technique leads to detect processes which can be hidden by using various methods such as: function hooking or direct kernel object manipulation (DKOM).
Basing on methods discussed in this paper, the proof-of-concept toolkit, called idetect, will be presented. This toolkit can help an investigator to extract some information from memory image or from memory object on a live system.

1. Introduction

In the past, a procedure of making an accurate and a reliable copy of the data from a compromised machine was limited into storages such as hard disks. It means, that a forensic analysis process relied on evidence found on file systems. There are several reasons for using such a procedure. First of all, the acquisition procedure is quite easy and an investigator's experience is not necessary. It is enough to remove power from a compromised machine and then to protect the crime scene. A second reason is more important. In most cases, examination tools, available on the market, can be used only to investigate file systems. There are some forensic tools such as EnCase EE or ProDiscover IR that help digital investigators to preserve some data from live system but for several reasons the tools are much more useful in an incident response process.
It is quite obvious that if we omit volatile data during an acquisition procedure, we can loose evidence. Furthermore, sophisticated methods of infecting computers, used by tools such as the FU rootkit or the SQL Slammer worm, show us that in near future the memory content will be the only place where evidence can be found. An infection of malicious code into a running processes, caused by internet worms and viruses, is more and more popular. For example, the mentioned SQL Slammer resides only in memory and never writes anything to disk.
There are also other advantages of performing memory investigation. Let's suppose, that we need to recover a part of email or a part of a document lost after a word editor crash. Where are we going to look it for? Even a simple task of searching of strings in main memory is sometimes very useful and allows us to extract interesting information such as commands typed by an intruder [6].
Above examples show us that memory investigation is critical for digital forensics. It is worth mentioning that most interesting information can be found when the compromised system was not rebooted. In this paper I will try to discuss some techniques of finding evidence in preserved memory image.

2. Problems with memory acquisition procedure

Most standards and best practice guidelines, such as: the "Computer Security Incident Handling Guide" from NIST or RFC 3227 "Guidelines for Evidence Collection and Archiving", include procedures of gathering volatile data. Some data, which must be acquired, is specified in these papers. For example: current network connections, running processes, users' sessions, kernel parameters, open files etc. But, to gather this data an investigator

must use several tools such as: netstat, lsof, ifconfig, etc. These tools help in collecting only obvious data, leaving most of the system's memory unanalyzed. Moreover, these tools are executed from user mode. Even statically linked tools can print unreliable data because of a kernel level modification.
The perfect tool for collecting volatile data should not rely on an operating system. Such solutions exist and one of them is described in the "Digital Investigation" magazine Vol. 1 No. 1. The described hardware-based solution called Tribble is almost perfect. Unfortunately, the special PCI card must be physically installed in a machine before an intrusion occurs. Obviously, it is impossible to install such a card in each machine in internet.
A memory acquisition procedure should be useful in every environment so in most cases it must be a software solution. The only thing which can be done by an investigator when an intrusion occurs is limiting memory collection process to few steps. This allows him to minimize impact on the compromised machine. He should dump main memory by using only one command. In second step, he should remove power from the compromised machine and then preserve remaining storages such as: hard disks, floppy disks, etc.
The dd tool can be used to dump main memory. This tool does a bit-by-bit copy from one file to another. Additionally, a content of main memory has to be saved on a storage other than local file systems. One of solutions is sending data to a remote host. The well known tool, which supports sending files through network, is the netcat tool.
In Linux operating system there are two files (/dev/mem and /proc/kcore) which correspond to main memory (RAM). The size of dumped memory is equal to the size of RAM. The /proc/kcore object is presented in the ELF core format, so it can be easily analyzed by the gdb tool. The size of the /proc/kcore file is a little bigger because of the ELF file header.

The whole memory can be dumped in the way presented below:

```
#/mnt/cdrom/dd if=/dev/mem | /mnt/cdrom/nc <ip address> <port number>
```

If we have dumped memory image, we can start digital investigation.

3. Introduction to analysis of the physical memory

3.1 Limitations of the paper

To limit the size of this document it was necessary to specify a few conditions:

- The 2.4.20 kernel release is used in all examples. Similar investigations can be performed with other kernel releases.
- The total size of physical memory is less than 896 MB. When physical memory is larger, additional calculations must be performed to localize page frames properly.
- The page frame size is 4 KB. This is the default value used in almost each Linux distribution.

The proof-of-concept toolkit idetect is used to simplify the described investigation. After simple modifications, the presented tools can be used on live systems during an incident response.

3.2 Symbols

During the digital investigation the System.map file can be very helpful. This file is used as a map with addresses of important kernel symbols. Every time you compile a new kernel, the addresses of various symbols are changed. The symbols included in that file provide helpful information for investigators. Let's say that we want to enumerate addresses of system calls. These addresses are stored in the kernel structure called the system call table. The sys_call_table symbol stores an address of this table. Using the cat and the grep commands we receive the address of that table.

```
$ cat /boot/System.map | grep sys_call_table
c030a0f0 D sys_call_table
```

On **Listing 1** first few entries of system call table are presented.

```
(gdb) x/256 0xc030a0f0
0xc030a0f0 <sys_call_table>:    0xc0128fa0      0xc011f8e0      0xc0107aa0      0xc0146cb0
0xc030a100 <sys_call_table+16>: 0xc0146df0      0xc0146220      0xc0146370      0xc0120060
0xc030a110 <sys_call_table+32>: 0xc01462c0      0xc0154510      0xc0154070      0xc0107bb0
0xc030a120 <sys_call_table+48>: 0xc01457f0      0xc0120d40      0xc01536b0      0xc0145b70
0xc030a130 <sys_call_table+64>: 0xc012ca00      0xc0128fa0      0xc014e910      0xc0146b40
…
```

Listing 1. The result of running the gdb tool against the /proc/kcore file.

Entries in this table correspond to names of functions stored in the file /usr/include/asm/unistd.h.

```
#define __NR_exit            1
#define __NR_fork            2
#define __NR_read            3
#define __NR_write           4
#define __NR_open            5
#define __NR_close           6
…
```

For example, the sys_write function is at 0xc0146df0, the sys_open function is at 0xc0146220, and so on.

The Symbol.map file is usually located in the /boot directory on a local file system.

4. An introduction to the digital investigation of the physical memory

Terminology used in the digital investigation of the physical memory is similar to the digital investigation against file systems. We can define data units and meta-data units.

The data unit contains raw data such as execution code or data section from memory mapped file. Additionally, they can contain a content of the stack or some meta data such a process descriptor. The data units are a fixed size. In most systems data unit is equal to 4 KB – this is the default size of the page frame.
The meta data unit is where the descriptive data about various memory structures is stored. This kind of the unit includes structures such as: page descriptors, process descriptors, memory regions, and so on.

4.1 Virtual Address Space

In most examples in this paper, the virtual (linear) addresses are used. All modern operating systems, including Linux, use this kind of addresses to access the contents of memory cells. In the x86 architecture with 32-bit CPU processors, a single 32-bit unsigned integer can be used to address up to 4 GB.
The Linux operating system divides memory into 2 parts. Upper 1 GB (0xc0000000 – 0xffffffff) is reserved for a kernel of operating system (this memory area can be accessed only when the CPU is switched into Kernel Mode). The remaining part of memory (3GB) is called User Land.

4.2 Physical addresses

Physical addresses are used to address memory cells in memory chips. Physical addresses are represented as a 32-bit unsigned integer. The CPU control unit transforms a linear address into a physical address automatically. Helpfully, a calculation from a linear address to a physical one is quite simple and this will be shown several times in the following chapters.

5. Map of system memory

In this chapter important structures of kernel memory are discussed. Only elements, useful for forensic investigators, will be described. It is recommended to use books [1][2] listed in references to find detailed information about each structure discussed in this document.

5.1 Uniform Memory Access

In x86 architecture, the Linux uses physical memory as an homogeneous, shared resource. This method is called Uniform Memory Access (UMA) and it means that the memory of the computer is seen by operating system as a single node. This node is represented as the static pg_data_t structure. The symbol contig_page_data contains the address of this structure. The pg_data_t struct contains information about: a size of a node (it means that it is a total size of physical memory), number of zones in a node, an address of table with page descriptors for this node and many more. At this point, it is important to understand what the zones are.

5.2 Zones

Physical memory (or node) is partitioned into three zones: ZONE_DMA, ZONE_NORMAL and ZONE_HIGHMEM. As we can read in book [1], there are reasons for providing such a fragmentation.

"However, real computer architectures have hardware constraints that may limit the way page frames can be used. In particular, the Linux kernel must deal with two hardware constraints of the 80 x 86 architecture:

- The Direct Memory Access (DMA) processors for ISA buses have a strong limitation: they are able to address only the first 16 MB of RAM.
- In modern 32-bit computers with lots of RAM, the CPU cannot directly access all physical memory because the linear address space is too small.

To scope with these two limitations, Linux partitions the physical memory in three zones."

The size of the ZONE_DMA zone is 16 MB. The size of the ZONE_NORMAL zone is equal to 896 MB – 16 MB (ZONE_DMA). The memory above 896 MB is included in the ZONE_HIGHMEM zone. This last zone contains page frames that cannot be directly accessed by the kernel because of limitation of a single 32-bit unsigned integer.
Each memory zone has its own descriptor of type zone_struct. This structure is defined in the file /usr/src/linux-2.4/include/linux/mmzone.h.
In all examples I used physical memory which size is 128 MB. It means that all users' and kernel data are stored in the ZONE_NORMAL zone and sometimes in the ZONE_DMA zone. Pointers to the mentioned zone descriptors are kept in the zone_table array. The address of the zone_table symbol is stored in the System.map file.

```
$cat System.map | grep zone_table
c03e6238 B zone_table
```

A letter "B" means that the symbol is in the uninitialized data section (known as BSS).

The content of the zone_table array is presented in **Listing 2**.

- zone_table[0] stores an address of the ZONE_DMA descriptor
- zone_table[1] stores an address of the ZONE_NORMAL descriptor
- zone_table[2] stores an address of the ZONE_HIGH descriptor

```
003e6230  80 9b 34 c0 ce ff 02 00  80 9b 34 c0 80 9e 34 c0  |..4.......4...4.|
003e6240  80 a1 34 c0 00 00 00 00  00 00 00 00 00 00 00 00  |..4.............|
003e6250  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
003e6260  ce ff 02 00 01 00 00 00  00 00 00 00 00 00 00 00  |................|
```

Listing 2. Fragment of physical memory with the zone_table array.

At address 0xc0349e80 we can find the zone descriptor for the ZONE_NORMAL zone.
Each zone descriptor contains a lot of information about its own page frames. There is also an address of the mem_map array. This table stores page descriptors of each page frame in the zone. Before looking closer at the mem_map array, let's focus on page frames and page descriptors.

5.3 Page frames and page descriptors

Most data used by the CPU are stored in physical memory in a form of pages frames (In fact, physical memory is partitioned into a fixed-length page frames). Each page frame is 4KB large – this is the default value. Sometimes x86 processors can use different sizes of page frame, such as 4 MB or 2 MB, but the standard memory allocation unit is 4 KB (only a standard size of page frames is discussed in this document). In page frames all volatile data is stored. For instance, when a file, which has a size of 7 KB is mapped, its content (code and data segments) will be stored in physical memory in two page frames. When a process requests a memory item, the system will use a linear (virtual) address to access requested data. To read data properly a hardware Memory Management Unit (MMU) translates a virtual address automatically to a physical one. The page may be marked as paged in or paged out. If the page is paged in then an access to memory can be proceed after translating a virtual address to a physical address. If the requested page is paged out, the MMU has to locate this page in the swap area and then load it into physical memory. These two possibilities will be discussed in next sections.

A kernel uses page descriptors to keep track of all physical pages. Each page frame has a corresponding page descriptor. In a page descriptor the information about state of page is stored.
A structure of page descriptor is defined in the file /usr/src/linux-2.4/include/linux/mm.h.

```
typedef struct page {
        struct list_head list;
        struct address_space *mapping;
        unsigned long index;
        struct page *next_hash;
        atomic_t count;
        unsigned long flags;
        struct list_head lru;
        union {
                struct pte_chain *chain;
                pte_addr_t direct;
        } pte;
        unsigned char age;
        struct page **pprev_hash;
        struct buffer_head * buffers;
  struct buffer_head * buffers;
#if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
        void *virtual;
#endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */
} mem_map_t;
```

The mapping field is a pointer to an address_space struct. The virtual field is an address of the physical page frame where data is stored. When the total amount of the physical memory is less than 896MB then it is easy to calculate a real (physical) address of each page frame by removing the PAGE_OFFSET (0xc0000000) from an address pointed by the virtual field.
The list field contains pointers to next and previous page descriptors which belong to the same memory region.
As it was mentioned, all page descriptors are stored in the one global mem_map array.

5.4 The mem_map array

In fact, we can identify three mem_map arrays. Each zone has its own array. When the first mem_map array (for the ZONE_DMA zone) finishes then the second mem_map array (for the ZONE_NORMAL zone) starts.
If we know the size of the page descriptor, it will be quite easy to find the beginning address of the mem_map array for the ZONE_NORMAL. The ZONE_NORMAL has a special meaning because most page frames, allocated by users' processes, belong to this zone.
For most 2.4.x kernels the size of the page descriptor is equal to 56 (0x38) bytes. We know that the total size of the ZONE_DMA is equal to 16 MB (0x01000000). We need 4096 page frames to address the ZONE_DMA zone (0x00000000 – 0x01000000). The size of the mem_map array for the ZONE_DMA zone is [0x38] bytes * [0x1000] = 0x38000.
I have to mention that Linux operating system maps virtual addresses into physical addresses starting from PAGE_OFFSET. For instance, the virtual address 0xc0000000 corresponds to the physical address 0x00000000, the address 0xc0001000 corresponds to 0x00001000 one and so on.
It is also important to note that physically the mem_map array is placed in page frames which belong to the ZONE_NORMAL zone. The location of the mem_map array is shown in **Figure 1**.

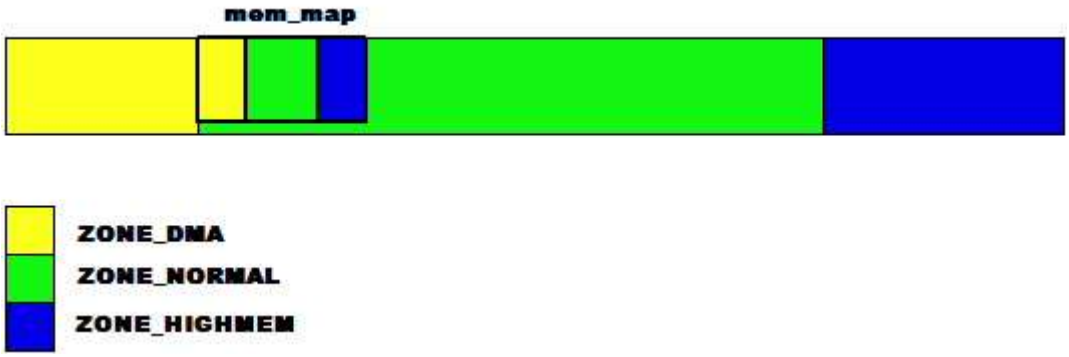

Figure 1. The mem_map array is stored in the ZONE_NORMAL zone.

Basing on the above scheme, we can assume that the mem_map for the ZONE_DMA zone starts from the physical address 0x01000000. In fact, the mem_map for ZONE_DMA starts from offset 0x30. Now we can easily locate the beginning physical address of the ZONE_NORMAL which is equal to: 0x01000030 + 0x00038000 = 0x01038030 (the virtual address of the mem_map array for ZONE_NORMAL zone is 0xc1038030).

I focused on the ZONE_NORMAL zone but digital investigators must also look at the ZONE_DMA zone. On some conditions, the page frames, which belong to users' processes, can be allocated in the ZONE_DMA zone. It happens when all page frames in the ZONE_NORMAL have been already allocated.
When files are mapped into the main memory, their inode structures (this inode structure will be discussed in the next section) have the associated address_space structure. The mapping field in page descriptor points exactly to this structure.

5.5 The address_space struct

This object can be associated with a regular file. So when a new process is created, an executable file is mapped into a process address space and then the address_space object is initialized in the memory. Each memory mapped file has its own address_space structure. The address_space structure is defined in /usr/src/linux-2.4/include/linux/fs.h source file.

```
struct address_space {
        struct list_head        clean_pages;    /* list of clean pages */
        struct list_head        dirty_pages;    /* list of dirty pages */
        struct list_head        locked_pages;   /* list of locked pages */
        unsigned long           nrpages;        /* number of total pages */
        struct address_space_operations *a_ops; /* methods */
        struct inode            *host;          /* owner: inode, block_device */
        struct vm_area_struct   *i_mmap;        /* list of private mappings */
        struct vm_area_struct   *i_mmap_shared; /* list of shared mappings */
        spinlock_t              i_shared_lock;  /* and spinlock protecting it */
        int                     gfp_mask;       /* how to allocate the pages */
};
```

The address_space object includes doubly linked lists of all page descriptors of mapped file stored in the main memory. If we sum all page descriptors we should receive the total number of physical page frames that is equal to a value kept in the nrpages field. The main role of address_space object is linking these page frames with methods associated with the mapped file. Two fields in the address_space structure are really important for us. The host field points to the inode structure of the memory mapped file, the second one - the i_mmap field points to the memory region to which these page frames belongs.

5.6 The inode struct

This structure describes memory mapped file. A lot of useful information can be obtained from this object. An investigator can: determine the directory from which the file was executed (if the inode describes an executable file), and find MAC times and so on.
Such a structure is described in the file /usr/src/linux-2.4/include/linux/fs.h. We don't have to fully understand the role of all fields in the inode structure right now. Let's focus on a few important fields. The i_ino field contains the inode number. The i_dentry field points to a dirent structure which describes the directory and contains the name of memory mapped file. The i_atime, i_mtime and i_ctime fields correspond to the access, the modification and the change times, known as the MAC times. The i_mapping field points to well known address_space structure.

5.7 The dentry struct

As it was mentioned, the dentry structure describes a directory object. This object is defined in the file /usr/src/linux-2.4/include/linux/dcache.h. The dentry structure includes the d_iname array. In this array the name of file is stored.

5.8 Memory regions

We remember that in the address_space structure there is a field which points to the proper memory region. Each memory region is described by the vm_area_struct structure. This object represents memory regions reserved for the User Mode process. For instance, each file, mapped into the process address space, contains at least three regions. First region includes an executable code, the second one represents an initialized data segment, the last one represents the heap. There are also additional memory regions for the stack, shared libraries and so on. In the pseudo file system procfs each process has a file called maps. This file contains addresses of all memory regions. For example, as it is illustrated in **Listing 3**, the process with PID = 1143 has 12 memory regions.

```
$ cat /proc/1143/maps

08048000-0804c000 r-xp 00000000 08:02 385967     /usr/sbin/atd
0804c000-0804d000 rw-p 00003000 08:02 385967     /usr/sbin/atd
0804d000-0804f000 rwxp 00000000 00:00 0
40000000-40015000 r-xp 00000000 08:02 337446     /lib/ld-2.3.2.so
40015000-40016000 rw-p 00014000 08:02 337446     /lib/ld-2.3.2.so
40016000-40018000 rw-p 00000000 00:00 0
4001d000-40028000 r-xp 00000000 08:02 337467     /lib/libnss_files-2.3.2.so
40028000-40029000 rw-p 0000a000 08:02 337467     /lib/libnss_files-2.3.2.so
42000000-4212e000 r-xp 00000000 08:02 433839     /lib/tls/libc-2.3.2.so
4212e000-42131000 rw-p 0012e000 08:02 433839     /lib/tls/libc-2.3.2.so
42131000-42133000 rw-p 00000000 00:00 0
bfffe000-c0000000 rwxp fffff000 00:00 0
```

Listing 3. Memory regions of selected process.

Each region is described by the vm_area_struct structure which is defined in the /usr/src/linux-2.4/include/linux/mm.h source file.

```
struct vm_area_struct {
        struct mm_struct * vm_mm;
        unsigned long vm_start;
        unsigned long vm_end;
```

```
        struct vm_area_struct *vm_next;
        pgprot_t vm_page_prot;
        unsigned long vm_flags;
        rb_node_t vm_rb;
       struct vm_area_struct *vm_next_share;
        struct vm_area_struct **vm_pprev_share;
        struct vm_operations_struct * vm_ops;
        unsigned long vm_pgoff;
        struct file * vm_file;
        unsigned long vm_raend;
        void * vm_private_data;
};
```

Every memory region starts from the address contained in the vm_start field. The vm_end field contains the last used address. All regions, which belong to a process address space, are linked by the vm_next field. This field points to the address of the next memory region occupied by the process. There are some flags which are associated with the page frames of the memory region. If a particular region contains the code of a mapped file, then the following flags are set: VM_READ, VM_EXEC and VM_EXECUTABLE. If a region is associated with a mapped file, then the vm_file field points to the object of type file struct. The file structure contains a field which points to a inode structure.

In the vm_area_struct structure the vm_mm field points to a special data structure. This structure of type mm_stuct is called a memory descriptor.

5.9 The mm_struct struct

Every User Mode process contains only one mm_struct descriptor. This structure contains all information related to a process address space. The mm_struct is defined in /usr/src/linux-2.4/include/linux/sched.h.

```
struct mm_struct {
        struct vm_area_struct * mmap;
        …
        pgd_t * pgd;
        …
        atomic_t mm_count;
        int map_count;
…
        } mm_stat;
```

The mmap field points to a linked list of all memory regions of type vma_area_struct.
The mm_struct object has a pointer to the Page Global Directory (the pgd field). The value, stored in this field, can be used to find all page frames of a process. It is also useful if we want to localize page frames which are swapped out to disk. The mm_count field informs us about a quantity of page frames which are allocated by a process. Additionally, all memory descriptors are linked by a doubly linked list (see **Figure 3**).

5.10 The task_struct object

This is the last kernel structure described in this paper. Every process is represented by a process descriptor that includes information about the current state of a process. This structure is defined in the /usr/src/linxu-2.4/include/linux/sched.h file. As we can assume, there is the mm field which points to the mm_struct structure. In case of kernel threads, which are also described by the task_struct, the pointer to the mm_struct object stores value equal to NULL.

All structures of type task_struct are linked by a doubly linked list. **Figure 2** illustrates a doubly linked list which links all existing process descriptors. The head of this list is kept in a structure called the init_task_union.

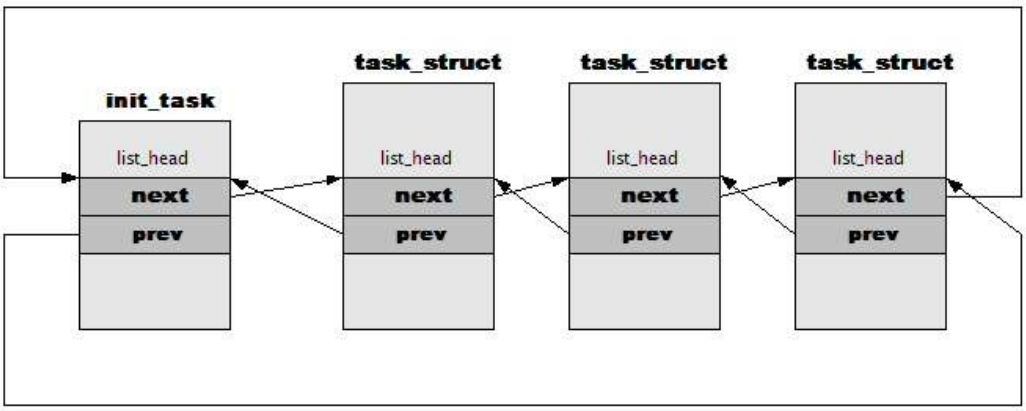

Figure 2. A doubly linked list.

The address of the init_task_union is exported. The init_task_union represents the process number 0 called swapper. If we find its address, we can enumerate almost all processes in a running state.

The init_task_union symbol is stored in the data section as it is shown below.

```
$ cat /boot/System.map | grep init_task_union
c034c000 D init_task_union
```

To list all processes we can go through this doubly linked list. Notice, that this technique is resistant to attempts of function hooking that the aim is hiding some processes. It happens like that because we read this list directly from the physical memory object. But what about processes which are unlinked from such a list? A technique called DKOM (Direct Kernel Object Manipulation) can be used by an intruder to hide some processes. In the Linux operating system linked process descriptors are used by the scheduler to reserve some time for the CPU. Thus, there are methods of changing the scheduler code in the way which makes possible to create the second list of processes which are seen only by the scheduler. A method of detecting such processes will be presented in the next section.
Figure 3 illustrates relations between all previously described objects. This map will be very helpful during digital investigations of the physical memory.

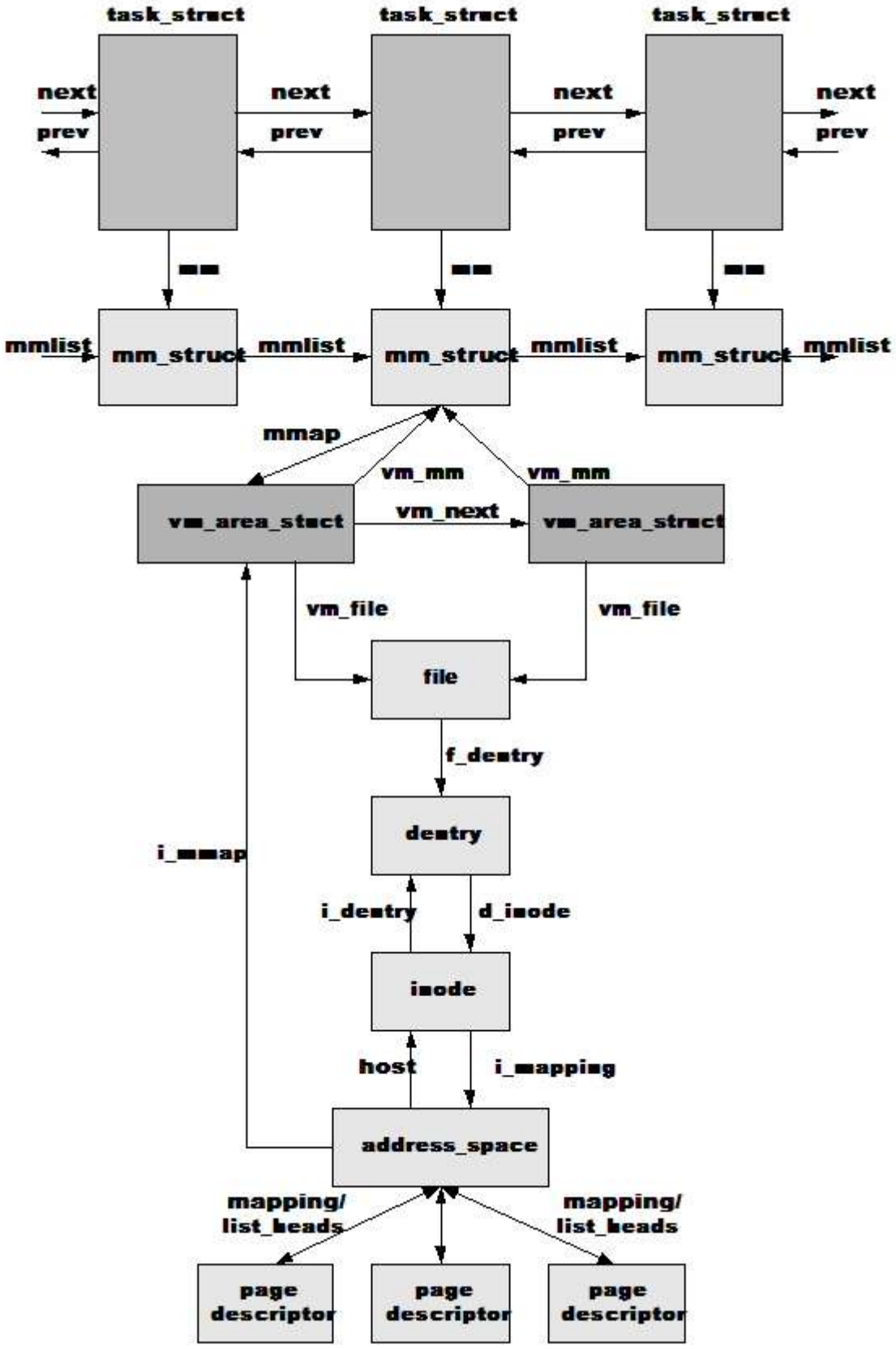

Figure 3. Relations between data structures of each process.

I have already described important structures of the kernel memory, so now we can use this knowledge to find evidence in the physical memory of the compromised machine.

6. Recovering content of files from the main memory and the memory image

As it is illustrated in **Figure 3**, it is quite easy to identify all page frames which belong to a memory mapped file. I mean page frames which are still in the main memory. Starting from the task_struct structure of the selected process, we can localize the address_struct structure, then we can enumerate all page descriptors and finally read addresses pointed by the virtual field of each page descriptor. As it was mentioned previously, the virtual field points to an address where the content of the requested file is stored.
Similar operations can be also performed on a live system during incident handling. Therefore, this method is an alternative to coping the exe objects from the /proc file system. A big advantage of such methods is that we don't use the ptrace() function. Incident handlers often use tools such as memfetch, pcat to dump a suspicious process. Unfortunately, these tools use the ptrace() function. When a suspicious process is already traced or is protected against such tools, there is no possibility to dump the whole address space of a suspicious process in right way. Let's discuss the simple example.
To perform this simple test I run the nc tool and then set the value, stored in the ptrace field, to other value than 0 (the ptrace field is included in the task_struct structure). It means that process is already traced. The fragment of a loadable kernel module, which performs this task, is listed below.

```
task_lock(current);
current->ptrace=1;
task_unlock(current);
```

Next, as it is illustrated in **Listing 4**, I wanted to dump the code segment of the nc process by using the pcat and the memgrep tools.

```
[root@mh302 bin]# ps -ef | grep nc
mariusz  9111  2563  0 22:08 pts/3    00:00:00 nc

[root@mh302 memgrep-0.8.0]# ./memgrep -p 9111 -d -a text -l 100 | more
ptrace(ATTACH): Operation not permitted
memgrep_initialize(): Couldn't open medium device.

[root@mh302 bin]# ./pcat 9111
./pcat: ptrace PTRACE_ATTACH: Operation not permitted
```

Listing 4. Attempts of dumping the code section from the selected process.

As you may notice, it is impossible to dump the content of the text section. A method of dumping selected page frames directly from the memory is much more effective. As it is shown in **Listing 5**, I used the tool called taskenum to enumerate all page descriptors of the memory region with the code segment of the suspicious process.

```
$ ./taskenum -a |grep nc
Pid:   9111 [nc] task_struct:[0xc38f7ff4] mm:[0xc068a880]

$ ./taskenum -p 9111|more
The process number 9111
Pid: 9111 [nc] task_struct:[0xc38f8000] mm:[0xc068a880]
Number of memory regions: 10

Mapping address: c29e9528
Number of pages: 4
page desc addresses: c10abfd8  c1074278 c10473e0 c1095de8
Address range: 8048000-804c000 (vma: c223dc80) i file (nc) c1d50400 i dir c30831
80 i inode c29e9480 i mapping c29e9528
```

```
...
```

Listing 5. Using the taskenum tool to enumerate page descriptors of a selected process.

Every page descriptor contains the virtual field which points to a physical address. Let's examine the first page descriptor which is available at the address 0xc10abfd8.

```
0xc10abfd8:     0xc1074278      0xc29e9528      0xc29e9528      0x00000001
0xc10abfe8:     0xc1059c48      0x00000003      0x010400cc      0xc1095e04
0xc10abff8:     0xc10473fc      0x03549124      0x00000099      0xc1279fa4
0xc10ac008:     0xc3a7a300      0xc3123000
```

This page frame starts at 0x03123000 (0xc3123000 – 0xc0000000).

Now we can dump the page frame using the dd tool. We must only convert the physical address into the decimal notation.

```
$dd if=/dev/mem of=page bs=1 count=4096 skip=51523584
```

Unfortunately this technique may not allow us to dump the whole address space of a suspicious process. To dump all page frames we must consider that some page frames can be swapped out. Additionally, in the address space of the process there are page frames which are shared between processes and there are page frames which contain data such as the stack or the heap. To localize all pages frames we must enumerate all page table entries of the process. The mm_struct object has the pgd field which points to the Page Global Directory. Entries of this table contain pointers to the Page Tables. Finally, we can identify entries which point to page frames or to swapped out page frames in the swap file (in this case entries of the Page Table store indexes).

7. Detecting and recovering files executed by an intruder

As we know, all page frames have corresponding descriptors which keep track of the current stage of each page frame. When a page frame is not used, the corresponding page descriptor is added to a group of free pages. More details about the Linux memory management and the Linux system algorithms can be found in the books [1][2]. But, when a page frame is released, not all fields of the page descriptor are cleared. For example, the mapping field still points to the address_space structure. If we try to analyze the content of the adress_space structure we can notice that the i_mmap field is cleared. This is what we are looking for.
The adderss_space structure contains information about other linked page frames. Furthermore the host field points to the inode structure of a used file. The inode structure contains useful information related to that file such as last access time. The dirent field points to the directory of type dirent, so the name of the file can be read.
Let's consider the simple example. Suppose, that after gaining access to a machine an intruder tried to execute the nc tool to create a very simple backdoor. A few seconds later this tool was terminated, as it is illustrated in **Listing 6**.

```
[bh@mh302 mariusz]$ date
Thu Mar 24 16:56:36 CET 2005
[bh@mh302 mariusz]$ nc -l -p 8888
 punt!
[bh@mh302 mariusz]$ date
Thu Mar 24 16:56:58 CET 2005
```

```
[bh@mh302 mariusz]$
```
Listing 6. Actions performed by the intruder.

Next day, the intrusion was detected and an investigator dumped the physical memory from the compromised machine. The proof-of-concept tool called pfenum was used to enumerate all released page frames. As it is shown in **Listing 7**, before running the tool we must set some values in the source code of the pfenum tool.

```
#define PAGES_NR 98288  <-total number of pages
#define MEMMAP 0xc100030  //an address of global mem_map table
#define MEMIMG "/home/mariusz/mem_image"  //path to physical memory image
```
Listing 7. Setting values in the source file of the pfenum tool.

As it is illustrated in **Listing 8**, the pfenum tool identifies released page frames which point to address_space structures. Additionally, the i_mmap field of the address_space structure must be cleared.
As we can see below, it was possible to find all page frames which contain the code of the suspicious file. We must note that the memory was dumped approximately 24 hours after the intrusion.

```
#pfenum –a

nc <-- vma = 0 [released] atime: 1111679811 This page frame nr: 13661 pointed by mapping at: cd7cb534 ->
vma: 0 mm_struct: 0 phaddress: c355d000 total nr of pf: 5
nc <-- vma = 0 [released] atime: 1111679811 This page frame nr: 48274 pointed by mapping at: cd7cb534 ->
vma: 0 mm_struct: 0 phaddress: cbc92000 total nr of pf: 5
nc <-- vma = 0 [released] atime: 1111679811 This page frame nr: 63105 pointed by mapping at: cd7cb534 ->
vma: 0 mm_struct: 0 phaddress: cf681000 total nr of pf: 5
nc <-- vma = 0 [released] atime: 1111679811 This page frame nr: 76405 pointed by mapping at: cd7cb534 ->
vma: 0 mm_struct: 0 phaddress: d2a75000 total nr of pf: 5
nc <-- vma = 0 [released] atime: 1111679811 This page frame nr: 87855 pointed by mapping at: cd7cb534 ->
vma: 0 mm_struct: 0 phaddress: d572f000 total nr of pf: 5
```
Listing 8. Using the pfenum tool to enumerating released page frames.

As it is shown below, we can determine the last access time to the nc file:

```
[root@mh302 kenelrh3a]# perl -e 'printf("%s\n",scalar localtime(1111679811))'
Thu Mar 24 16:56:51 2005
```

It is possible to dump the content of such a file. As it was mentioned in this paper, the virtual field of a page descriptor points to a physical address of the corresponding page frame. Let's consider the first detected page descriptor:

```
nc <-- vma = 0 [released] atime: 1111679811 This page frame nr: 13661 pointed by mapping at: cd7cb534 ->
vma: 0 mm_struct: 0 phaddress: c355d000 total nr of pf: 5
```

The page frame starts at 0x0355d000 (0xc355d000 – 0xc0000000). We can extract the content of that page frame from the memory image using the dd tool in the following way:

```
$dd if=mem_image of=p13661 bs=1 skip=55955456 count=4096
```

The physical address in a decimal notation must be provided as a value for the skip parameter. The address 0x0355d000 is represented in the decimal form as 55955456.

We repeat the same steps for remaining page frames.

```
dd if=mem_image of=p48274 bs=1 skip=197730304 count=4096
dd if=mem_image of=p63105 bs=1 skip=258478080 count=4096
dd if=mem_image of=p76405 bs=1 skip=312954880 count=4096
dd if=mem_image of=p87855 bs=1 skip=359854080 count=4096
```

By using the file tool we can identify the beginning of the suspicious file.

```
$ file *
p13661: data
p48274: data
p63105: data
p76405: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2
.2.5, dynamically linked (uses shared libs), stripped
p87855: data
```

To determine the order of page frames we must take a look into the page descriptor of the first page frame (the number of the page descriptor is 76405)

```
0xc14149c8:     0xcd7cb534      0xc10bac88      0xcd7cb534      0x00000000
0xc14149d8:     0x00000000      0x00000002      0x012c000c      0xc142a8c4
0xc14149e8:     0xc10baca4      0x00000000      0x00000026      0xc1d51bf0
0xc14149f8:     0xcea5c850      0xd2a75000
```

The next page descriptor is 0xc10bac88.

```
0xc10bac88:     0xc14149c8      0xc14b1278      0xcd7cb534      0x00000001
0xc10bac98:     0xc138cb30      0x00000002      0x012c000c      0xc14149e4
0xc10baca8:     0xc14b1294      0x00000000      0x00000026      0xc1d51bf4
0xc10bacb8:     0xcea5cc60      0xc355d000
```

Now, we must attach the page frame number 13661 to the page frame number 76405.

```
$ cp p76405 nc_suspicious
$ cat p13661 >> nc_suspicious
```

We repeat the same steps for remaining page frames. So the third page frame is 87855, the forth one is 48274 and the last one is 63105. After attaching all page frames we should receive the file executed by the intruder in the past.

```
$ chmod +x nc_suspicious
$ ./nc_suspicious
Cmd line:
```

8. Detecting all User Mode processes

This chapter discusses the method of detecting all processes run in the User Mode. This technique relies on enumerating all page descriptors that are allocated by an operating system. Through this method, it is possible to find all objects of type mm_struct. I have already mentioned that every User Mode process has only one memory descriptor. In first step, we'll try to explore page descriptors of memory mapped files by selecting memory regions of type vma_area_struct. Additionally, we must verify flags which are associated with the pages of the memory region. They are stored in the vm_flags field of the memory

region descriptor. However, we want to detect all User Mode processes, so it is enough to focus on executable files. The memory region, that maps an executable file, has the VM_EXECUTABLE flag set. As it is shown in **Figure 3**, the mapping field of a page descriptor points to the address_space structure. The i_mmap field of the address_space structure points to the memory region descriptor of type vma_area_struct. The vm_mm field of the memory region points to the memory descriptor that owns the region. As it was pointed previously, the kernel threads are not detected.

Before moving on, it is worth mentioning that we must check page descriptors reserved for the ZONE_NORMAL zone and for the ZONE_DMA zone. If the kernel has not enough space to allocate page frames in the ZONE_NORMAL zone then it allocates page frames in the ZONE_DMA zone.

The proof–of-concept tool called procenum uses the technique discussed earlier in this chapter. This tool prints all User Mode processes illustrated in **Listing 9**.

```
$ procenum –a

table entry: [0] name: [sendmail.sendmail] mm_struct: [c3ed4d80]
table entry: [1] name: [procenum2] mm_struct: [c04da880]
table entry: [2] name: [syslogd] mm_struct: [c3ed4880]
table entry: [3] name: [sshd] mm_struct: [c3ed4480]
table entry: [4] name: [init] mm_struct: [c3ed4180]
table entry: [5] name: [eventwriterd] mm_struct: [c37bc380]
table entry: [6] name: [mh-handoff-receiver] mm_struct: [c1d75b80]
table entry: [7] name: [bash] mm_struct: [c04da280]
table entry: [8] name: [more] mm_struct: [c04da680]
table entry: [9] name: [mingetty] mm_struct: [c314f580]
table entry: [10] name: [mh-register-client] mm_struct: [c1d75180]
table entry: [11] name: [qsppproxy] mm_struct: [c37bce80]
table entry: [12] name: [flowchaser] mm_struct: [c3ed4980]
table entry: [13] name: [java] mm_struct: [c1d75780]
table entry: [14] name: [mh-handoff-sender] mm_struct: [c1d75580]
…
```

Listing 9. Using the procenum tool to detect all User Mode processes.

We have already discussed the basic concept of detecting User Mode processes. Therefore, suppose that all page descriptors of the memory mapped file are swapped out, as it is shown in **Listing 10**. See the next section to solve this problem.

```
./taskenum -p 939|more
Process: 939
Pid: 939 [portmap] task_struct:[0xc331e000] mm:[0xc3ed4080]
Nr of vmas: 18
Mapping: c3189628 Nr page frames: 0 ←
8048000-804b000 (vma: c3aadf00), file (portmap) c3a97400, dirent c33a0d00, inode c3189580, mapping
c3189628
```

Listing 10. The memory region with swapped out page frames.

9. Solving problem with swapped-out pages

In fact, in Linux operating system there are a few page frames of each process which are never swapped out. Even if the all page frames, which contain the memory mapped file, are

swapped out there are at least one additional page frame which contains the array of PMD entries of type pmd_t. These page frames also belong to the process address space. If we list the content of such a page descriptor we will see that the mapping field points directly to the mm_struct of the normal process.

If we run the procenum tool in the debug mode, we will see detailed data about each page descriptor. In **Listing 11** we can see that two additional page frames belong to the address space of the process atd.

```
$ ./procenum –v
…
ADDED!!! c314f480
Page [c1092070] 10680 belongs to mapping c314f480 -> vma: c2b11e80 ->mm_struct: c314
f480 physical address: c29b8000 page count: 1 name: atd
Page never used -->
Page [c10920e0] 10682 belongs to mapping c314f480 vma: c2b11e80 i mm_struct c314f480 i jej fizyczna
c29ba000 ilestron 1 i nazwa atd
zawartosc [61] atd wynosi: c314f480
```

Listing 11. Detailed information about the page frames.

Let's take a look at the content of the page descriptor at the address 0xc1092070.

```
0xc1092070:     0x00000000      0x00000000      0xc314f480      0xbfc00000
0xc1092080:     0x00000000      0x00000001      0x01000000      0x00000000
0xc1092090:     0x00000000      0x00000000      0x00000000      0x00000000
0xc10920a0:     0x00000000      0xc29b8000
```

At offset 0x8 there is the mapping field which points to the address at 0xc314f480 – this is the memory descriptor of type mm_struct of the executable file.

The fragment of the mm_struct object is presented below:

```
0xc314f480:     0xc2b11e80      0xc2b11618      0xc2b11e00      0x40017000
0xc314f490:     0xc29be000      0x00000001      0x00000001      0x0000000c
0xc314f4a0:     0x00000000      0xc314f4a4      0xc314f4a4      0xc314f1ac
0xc314f4b0:     0xc22868ac      0x08048000      0x0804b30c      0x0804c320
0xc314f4c0:     0x0804c5d8      0x0804c610      0x0804f000      0xbfffecb0
0xc314f4d0:     0xbfffff10      0xbffffff1e     0xbffffff1e     0xbffffffee
...
```

Now, take a look at the content of the physical page frame kept at the address 0x029b8000. As we can see in the previous chapter, it is necessary to perform a simple operation to receive the real physical address of the page frame.

```
0x029b8000:     0x00000000      0x00000000      0x00000000      0x00000000
0x029b8010:     0x00000000      0x00000000      0x00000000      0x00000000
0x029b8020:     0x00000000      0x00000000      0x00000000      0x00000000
…
0x029b8fe0:     0x00000000      0x00000000      0x00000000      0x00000000
0x029b8ff0:     0x00000000      0x00000000      0x000d4f00      0x000d4c00
```

As it is shown above, this page frame contains only two entries which are index numbers in the swap page file.

10. Conclusion

The digital investigation of the physical memory from a compromised machine is a very new field in forensic analysis. All methods, discussed in this paper, are only a small step towards performing full investigation of the physical memory. A lot of work must be done to analyze each page frame of the memory and to correlate it with the corresponding data structures. It is necessary to correlate this data with the evidence preserved in the swap page files and in the file systems. In this document only the Linux physical memory was discussed, but similar work can be done for the Windows memory.

11. References

[1] Daniel P. Bovet, Marco Cesati. "Understanding the Linux Kernel, 2nd Edition".
[2] Mel Gorman. "Understanding the Linux Virtual Memory Manager".
[3] RFC 3227. "Guidelines for Evidence Collection and Archiving".
[4] NIST SP 800-61. "Computer Security Incident Handling Guide".
[5] Brian D. Carrier, Joe Grand. "A hardware-based memory acquisition procedure for digital investigations". Digital Investigation Vol. 1 No. 1..
[6] Mariusz Burdach. "Forensic Analysis of a Live Linux System, Part Two", http://securityfocus.com/infocus/1773.
[7] Current version of the idetect toolkit is available at http://forensic.seccure.net/.