University of Mannheim
Technical Report TR-2011-001

Internals of Windows Memory Management (not only) for Malware Analysis

Carsten Willems

University of Mannheim, Germany

Abstract. This document presents insights from extensive reverse engineering efforts of the memory management mechanisms of Windows XP. The focus lies on (1) the mechanisms which are used to map executable modules into the address space and (2) the role of the page fault handler in this context.

1 Introduction

Malware is an ubiquitous threat, which is growing from day to day. In order to fight it and mitigate its effects, it is necessary to analyze the upcoming malicious samples. There are two main different approaches to that analysis in general, one static and a dynamic one. The static approach tries to comprehend the concrete semantics of either the complete program, or parts of it, by disassembling its instructions [2,8,22,23,9]. In most cases malicious applications are protected in several manners, e.g. by encryption, code obfuscation and anti-reversing tricks in general [11,24]. Accordingly, before their functionality can be analyzed, these protection layers have to be removed [29,49], which is a very time consuming process.

In the dynamic approach the malicious application is more viewed as a *black box* and its behaviour is monitored while it is executed. The monitoring itself can be realized from different points of the application and operating system stack. It can be done from within the monitored application itself, from the kernel, or from outside the system by using a hypervisor. Depending on the particular location of the monitoring component, different techniques can be applied to intercept the interaction between the malware and the operating system, e.g. system call hooks [47,51], binary instrumentation [10,25,46], virtual machine introspection [1,16,14] and so on. The two main disadvantages of dynamic analysis are the fact that it most often is easy to detect [17] and, furthermore, that in each analysis run only one possible execution path is monitored [30].

Sometimes static and dynamic methods are combined [36,33,26] to speed up the analysis process. For instance the sample is executed for a certain time and then, e.g. after all protection and decryption layers are hopefully passed, the resulting instructions are dumped and disassembled [44,6]. This diminishes the necessary effort of the analyzer and benefits from the advantages of both approaches.

In nearly all of those analysis scenarios a detailed understanding of the underlying operating system internals is mandatory. Especially in dynamic analysis a full understanding of the memory management internals is essential, since there the analyzing components are either embedded directly into the operating system or they are highly interacting with it. In order to learn the functionality of a running process without having its source code at hand, it is mandatory to reconstruct its data structures, and this cannot be done without knowing the way how these are mapped into the available memory.

Besides malware analysis, also the field of digital memory forensics [18,37] heavily depends on this kind of knowledge. Most malicious applications still are aiming at the Windows operating system, which obviously is not an open-source system. Though there already exist excellent work in describing its internals [34,39,38], there still is a lack of information about some lower level mechanisms.

Hence, in the following we will investigate the memory management of Windows XP and focus on the mechanisms which are used to map executable modules into the address space and the role of the page fault handler in this context. For that purpose, we will briefly explain some background information in section 2 and then delve into the memory internals in section 4.

2 Memory Management Background

In this section we will present some fundamentals which are necessary to understand the remainder of this work. Since our approach heavily relies on the Windows paging mechanism, we start with a short preface about paging in general. After that we will briefly present the highly related mechanism of *translation lookaside buffers.*

2.1 Paging

Most contemporary operating systems use a paging mechanism to realize virtual address spaces and abstract from their physical memory. The virtual memory is divided into equal sized pages and the physical memory into frames of the same size, and each page can be mapped to an arbitrary frame. Pages that are currently not mapped are temporarily stored on secondary storage, e.g. in a page file on the hard disk. Effects of paging are that physical memory is used much more efficiently and can be also smaller than the virtual memory offered by the OS. Furthermore, a page-level memory protection scheme can be applied, e.g. to mark particular memory ranges as read-only or non executable. Before paging the x86 CPUs only offered segmentation for memory protection, which offered less flexibility and operated with a much more coarse granularity.

The mapping between virtual and physical addresses normally is done transparently to the running applications by the *memory management unit* (MMU), which is a dedicated part of the CPU. Under some special conditions, the hardware cannot resolve the mapping on its own without the help of the OS. In such an event a *page fault* is generated, which has to be handled by the *page fault handler* of the OS. Examples of such events are accessing pages that are currently not mapped into the physical memory or trying to write to read-only memory.

Besides these reaction on page faults, the OS also maintains some memory related system threads. Those periodically exchange data between the physical memory and the hard disk, e.g. to always maintain sufficient available memory.

For managing the paging mechanism, the MMU and the related system functions use some specific data structures, namely *page tables*. Since the address spaces of different processes are isolated from each other, each process uses its own instances of these structures. Each *page table entry* (PTE) describes the memory range of one associated page. In order to locate the PTE which is associated to a particular memory address, it is split up into two parts: the upper bits are an index into the page table, and the lower bits are an offset to the start address of the relating frame.

The conventional x86 architecture uses pages of 4 KB in size, but there exist extensions with 2 MB or 4 MB pages. Since most processes only use a very small amount of their available virtual memory, multiple levels of page table structures are used to reduce the space necessary for managing the memory. In two-level paging an additional *page directory* is employed, where each *page directory entry* (PDE) points to a separate page table (Figure 1). In that scenario, the virtual address is split into three parts: one index into the page directory, one index into the resulting page table and the last part as an offset into the frame.

Obviously the usage of multiple-level paging involves a performance degradation, since multiple memory lookups have to be performed in order to locate the resulting entry. To speed up this process, *Table lookaside buffers* (TLB) are used. These are very fast caches that store information about the virtual-to-physical address mapping from the last memory accesses.

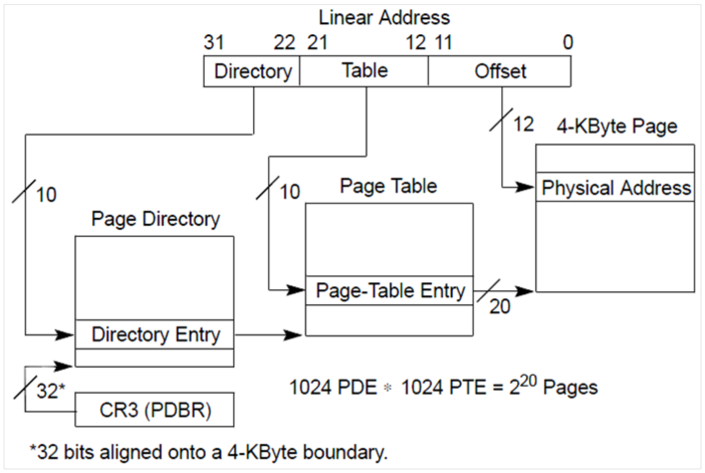

Fig. 1. Two Level Paging [20].

Each PTE contains different information about the related page and its current state (lower part of figure 2). The most important one is bit 0, which specified if the PTE is present (or valid) or not. Only if this bit is set, the MMU can perform the address resolution on its own. If it is cleared the OS will be invoked to take action. In that case, all the other PTE fields can be used by the OS arbitrarily. One other essential field of valid PTEs is the *page base address* or *frame number*, which specifies to which physical frame the page is currently mapped. Furthermore, there are a lot of bit fields that are used for memory protection and for the page substitution strategy. A PDE is quite similar to a PTE with the main difference that it points to a PTE instead of a frame.

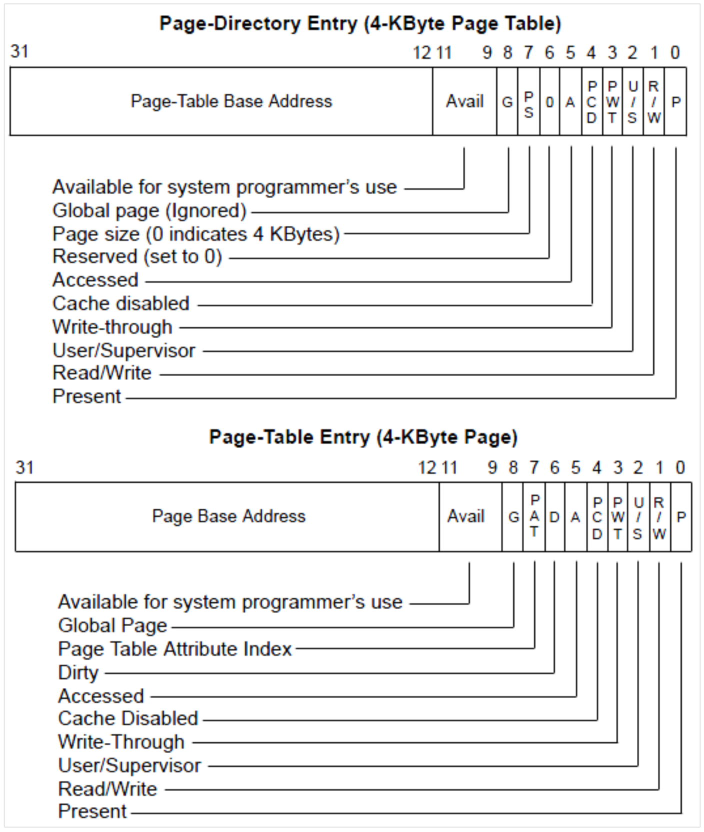

Fig. 2. PDE and PTE Fields [20].

As mentioned, all paging structures are maintained separately for each running process. Switching between different address spaces is done by simply modifying the pointer that specifies the page directory (or page table in case of a single-level paging). In the Intel architecture this pointer is kept in the *page directory base register* (PDBR), which is stored within the upper 20 bits of the *control register 3* (CR3).

Detailed information on paging and its related facilities and data structures can be found in the *Intel Software Developers Manuals* [20].

2.2 Translation Lookaside Buffers

Modern hardware architectures use several mechanism to speed up memory accesses to overcome the performance gap between the CPU and RAM. One method to improve the effectiveness of virtual memory is to use *translation lookaside buffers (TLBs)*. Without those resolving a virtual to a physical address may require several memory read operations, depending on the amount of page levels. A TLB is fast associative memory that stores the last recently used mappings between *virtual page numbers (VPN)* and resulting *physical frame numbers (PFN)*. If memory is accessed for the second time, while its mapping is still stored in the TLB, no further page walk has to be performed.

There exist many different ways to implement TLBs. Some are rather small and contained within the CPU, others are larger and placed between the CPU and the different caches. Sometimes there are distinct TLBs for data and instruction addresses, the x86 architecture offers a DTLB for the former and an ITLB for the latter one. Normally the DTLB and ITLB are synchronized, but there are projects in which they are enforced to become unsynchronized. This can be done for several reasons, e.g. to implement the NX feature on CPUs that does not offer support for that [45], to unpack compressed applications [44] or even to realize *memory cloaking* [43].

Besides the VPN and the PFN, each TLB entry also has to store a copy of the most important other PTE flags, e.g. the accessed and the dirty bit as well as the memory protection flags. Accordingly, one critical requirement for the operating system is to keep the TLB entries and the PTE entries coherent. Therefore, it has the possibility to *flush* the complete TLB or just selected entries. For instance on a context switch all process related entries have to be removed from the TLB, since those are meaningless and incorrect when switching to a different address space.

3 Memory Protection and Exploitation Techniques

In this section we present contemporary memory protection techniques as well as the countermeasures to overcome them. We have arranged them in a reasonable chronicle order, i.e. we alternate the particular protection techniques with the appropriate exploitation mechanisms, starting with *Data Execution Prevention* and ending with *JIT spraying*.

3.1 No eXecute

One important contemporary security technique is called *No eXecute* (NX). It is also known under different other names, depending on the CPU or OS vendor who specifies it: W $\oplus$ X, eXecute disable (XD), *Enhanced Virus Protection* bit or *Data Execution Prevention* (DEP). Originally this feature was introduced under the name $W \oplus X$ with OpenBSD 3.3 in May 2003. Another famous advocate of related techniques is the PaX [45] project.

The aim of this technique is to enforce the distinction between data and code memory. In general there is no differentiation between code and data on the commonly used *von Neumann architecture*. Each byte in memory can either be used as code, if its address is loaded into the instruction pointer, or it can be used as data if it is accessed by a load or store operation. One very striking effect of this is that attackers are able to conceal malicious code as harmless data and then later on, with the help of some vulnerability, execute it as code.

The NX protection scheme can either be implemented by hardware or simulated in software. On newer x86/x64 machines there is one dedicated PTE flag that controls if a certain page is executable or not. If this flag is set, a page fault is invoked on the attempt to execute code from the corresponding page. Therefore, besides the hardware support the operating system and, especially the page fault handler, has to implement this security feature as well. In the Windows world, DEP was introduced with Windows XP Service Pack 2 as an option and, nowadays, is enabled by default with current Windows version.

On an Intel machine, the *physical address extension* (PAE) mode has to be enabled in order to use the NX flag within the PTEs. Originally this extension was introduced to enable the usage of a physical address space that is larger than 4 GB. As an implication the PTE size was increased from 32 to 64 bit, while offering also that flag. Though most current Windows installations do not use the address space enlargement, they have PAE enabled to offer hardware support for their DEP feature.

3.2 Return Oriented Programming

In order to overcome the NX protection, attackers use different methods. One very powerful way is to locate and execute useful instructions in one of the loaded system or application library modules. In the beginning, attackers performed *return to libc* [13] attacks, in which they called helpful library functions, e.g. to start a different process or to modify security settings on the local system. A more sophisticated generalization of this approach is *return oriented programming* (ROP) [40,21]. Here attackers do not call complete library functions, but instead they use only small code chunks and chain those together to implement their desired functionality.

In order to concatenate code chunks, called gadgets, the attacker has to locate useful instruction sequences which are trailed by a *return* (RET) operation. Having those gadgets at hand, the attacker can prepare a list of their start addresses, and prepare the stack in a way such that the first of these address is placed in a location of a regular saved RET address. When the overwritten RET address is popped from the stack, the ROP sequence is started. Each time a small code chunk has been completed and the trailing return operation is executed, the starting address of the next chunk is popped from the stack. In order to further control and customize the executed code, function arguments can also be interleaved into the list of code addresses on the stack.

This technique is particularly powerful on CISC architectures, because of their variable length instruction set. Since an instruction has not to be memory-aligned, it can be executed with an arbitrary byte offset with respect to its intended start address. This has the effect that an attacker is not limited to the set of the intended instructions, but can also jump directly into the middle of an existing instruction, which results in a totally different one. Nevertheless, there exist also approaches for performing ROP on RISC architectures [7].

3.3 Address Space Layout Randomization

Another security technique which is often used in modern computer systems is *Address Space Layout Randomization (ASLR)*. With that mechanism all memory locations are no longer fixed and predictable, but are chosen randomly. Consequently, the stack, the heap and the loaded modules will have different memory addresses each time a process is started. Therefore, it is much harder for attackers to find usable memory locations, e.g. the beginning of a certain function or the effective location of malicious code or data. Accordingly, when the memory layout is randomized and an exploit is performed that does not take this into account, the process will rather crash than being exploited. Therefore, memory randomization should always be combined with a crash monitor facility. Once again the Pax project [45] was one of the pushing forces for this protection scheme. Also modern Windows operating systems use ASLR.

Of course there are countermeasures for this security scheme as well. One possible approach is to locate and utilize non-randomized memory, since early implementations of ASLR do not randomize the complete memory but only some parts of it. Even the current implementation in Windows allows the disabling of ASLR on a per-module-base. Accordingly, even if all Windows libraries are ASLR-protected, a lot of custom libraries are not and constitute an easy exploitation target for attackers. In [12] the protection could be bypassed, because the *Internet Explorer* DLL *mscorie.dll* was distributed with ASLR disabled. There are a bunch of more different ways to defeat ASLR [15,32].

3.4 Other Memory Protections

In the preceding section we have presented two of the most contemporary memory protection techniques: DEP and ASLR. Besides those, many other (memory) protection features have been introduced into operating systems as well as into available compilers. In the following we will summarizes more of those in a sketchy way. A comprehensive list with detailed descriptions can be found in [42].

Stack cookies or *stack canaries* are used to detect stack buffer overwriting [48,35,50]. In each function prologue a special non-predictable value is stored between the local variable area and the return address on the stack. Before leaving the function the epilogue then verifies that value and crashes the process if it was modified.

Also the heap memory can be protected by cookies in a similar way, as it is for example done in Windows XP SP2. Windows Vista goes one step further and encrypts all heap-related meta data to protect it from being overwritten. A related technique that also protects the heap memory is *safe unlinking*, in which additional checks are done before a heap chunk is unlinked [27].

Variable reordering [48] is used to move possibly overwritable buffers to the *end* of the local variable list, such that no other variables are affected by an overwriting attempt. This technique addresses the case in which the vulnerable function is never left, hence the modified RET address is never popped from the stack. Instead some local variables on the stack are modified in order to manipulate the regular control flow even without returning from the function at all.

SEH Handler Validation tries to detect the manipulation of SEH routine addresses. There exist different implementations. Some of them maintain a list of all known and trustable SEH handlers per module [31]. If an exception is to be raised, the particular handler is then checked against this predefined list before it is called. If it is unknown, and hence constitute an illegitimate one, the process is crashed immediately.

3.5 Heap Spraying

Exploitation attempts often consists of two consecutive steps: the first one prepares the current environment for successful exploitation and the second one triggers some vulnerability in the running software. In the preparation step in most cases executable shellcode is placed at some memory location, which the control flow will be redirected to as an effect of a successful exploit. Depending on the situation that particular destination code address may be known in advance or not. Furthermore, it may be possible to explicitly write to that particular address or not. Therefore, even with having a exploitable vulnerability at hand, it may still pose a serious problem to effectively prepare the memory such that after exploitation meaningful shellcode is executed.

One popular approach to prepare the memory in an appropriate way is *heap spraying* [5]. The technique is rather old but has become famous again in the context of web browser and PDF exploits. The idea is to fill the complete heap memory with identical copies of the same malicious code (Figure 3). If large parts of the heap are prepared in such way, chances are high that one of those created shellcode instances is hit when a vulnerability is exploited. In order to increase the effectiveness of this approach, very large NOP-slides are installed before each shellcode instance. A NOP-slide is a very long sequence of NOP operations or some other instructions that do not modify any of the important CPU registers. No matter at which point inside the NOP sequence the execution starts, eventually the shellcode will be reached.

3.6 JIT Spraying

One problem with traditional heap spraying is that on systems with DEP the resulting heap memory is non-executable. Hence, the execution of shellcode which was laid out by this method, will lead to a system crash. A more prevailing spraying technique called *JIT spraying* was introduced in [3,4]. The improving idea focuses on modern interpreter languages which employ *Just in Time* (JIT) compilers, e.g. the *ActionScript Virtual machine* which is included in *Adobe Acrobat Reader*. For increasing the computing performance the script code is compiled into native machine instructions just right before it is executed. Obviously, the compiler output has to reside in executable memory. By predicting the JIT compiler results and designing an appropriate compiler input it is possible to generate an output which can be misused as shellcode.

The idea shown in [3,41] uses long chains of XOR operations with 4 Byte immediate values as JIT compiler input. Since the immediate values are copied 1:1 into the resulting output, it is possible to encode x86 machine instructions by them. Due to the properties of the CISC instruction set, the resulting code will have a complete different semantic when it is not executed from the beginning but with a 1-byte offset. In that

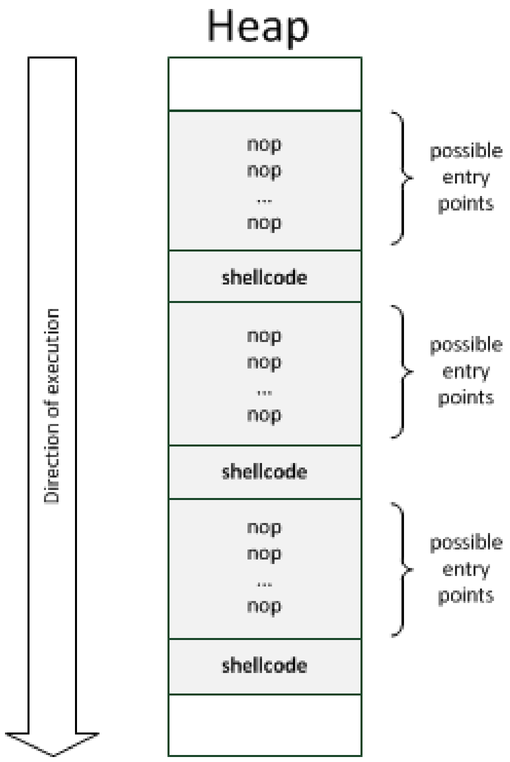

Fig. 3. Heap Spray.

case the first instruction is determined by the immediate value which was used with the first XOR operation.

4 Windows Memory Management

In order to instrument the Windows memory management mechanisms for dynamic malware analysis, a detailed understanding about the Windows paging internals, especially about the underlying data structures and the involved algorithms is necessary. Therefore, in the following section we will give detailed background information about the memory management internals of the 32 bit version of Windows. Most of the findings were obtained by reverse engineering Windows XP, but newer Windows version do not differ significantly in these fundamental system essentials.

4.1 Memory Objects

The 32 bit Windows versions split the available 4 GB virtual address space into a lower 2 GB part of per-process user space and another upper 2 GB region of global kernel space memory. Optionally this partitioning can be changed to a 3 GB user space region plus only 1 GB for the kernel. Each running process has its own user space memory, which cannot be accessed directly from other processes. This is realized by modifying the page directory pointer in the CR3 register on each context switch. All processes use the same kernel space memory, hence the same system page table entries for the upper 2 GB. Since we are focusing solely on user space exploits, we do not consider kernel space memory in the remainder of this paper.

Windows memory can contain several different types of code or data regions and each of those is affected by paging: stacks, heaps, mapped modules as well as process and thread environment and control blocks. Memory can also be characterized by the type of the source from which its content arises. Some memory regions contain data from files, which are mapped into the address space, others reflect mapped data which is actually stored in physical devices and others contain volatile data which only exists in the physical memory itself and will be gone once it is powered off. The two most important source types result in *private* and *file-backed* memory. While the first one is used for all dynamic memory, like stack, heap or system management blocks, the latter one is either associated with regular files or with one of the page files. Windows always uses memory-mapped files when executable files are loaded into a process. Hence, it is a very important construct, especially with respect to memory execution protection.

Windows differs between files which are mapped as *data* and those which are mapped as executable *images*. Data files may have random content and structure and are simply mapped one to one from file layout to memory layout. Image files, on the other hand, have to be stored in the *portable executable* (PE) format [19,28] and may contain data as well as executable code. A PE file contains several sections, which each are associated with different usage characteristics. Data sections may be read-only or writable and they may contain initialized or non-initialized data. Code sections in general are executable and read-only, but may be modified to being writable in some scenarios as well. To support reusing of code and reduce the amount of necessary memory, code sections are therefore loaded with a *copy on write* flag. This means that all processes that load the same executable will effectively use the same physical memory until one process modifies that memory. In that case, a private copy of the affected page is created for that process and all other processes will still remain using the old unmodified page.

The type of a mapped file does *not* determine if it is loaded as a data or an image file, but the code that loads it into memory has to specify that. Therefore, image files can be mapped as data files as well. In contrary to that, if a data file is not stored in the PE format, the system loader will refuse to load it as an image.

In general data files as well as image files can be mapped to arbitrary virtual memory ranges. Nevertheless, when mapping a file an optional base address can be specified. Windows then tries to load the file to that specified address and will return an error, if the specified range is already occupied. If no base address is given, Windows tries to determine a free memory range on its own. In the case of image files, a favored base address is always given in the PE header. However, if the specified address is already used, image files may be relocated as well.

For file-backed memory a bunch of different management objects are involved: *virtual address descriptors*, *sections*, *subsections*, *segments*, *PPTEs* and *control areas*. Figure 4 [38] gives an overview of the relationships between those different object types. In the following we will briefly explain the different objects and illustrate when and how they are created and used.

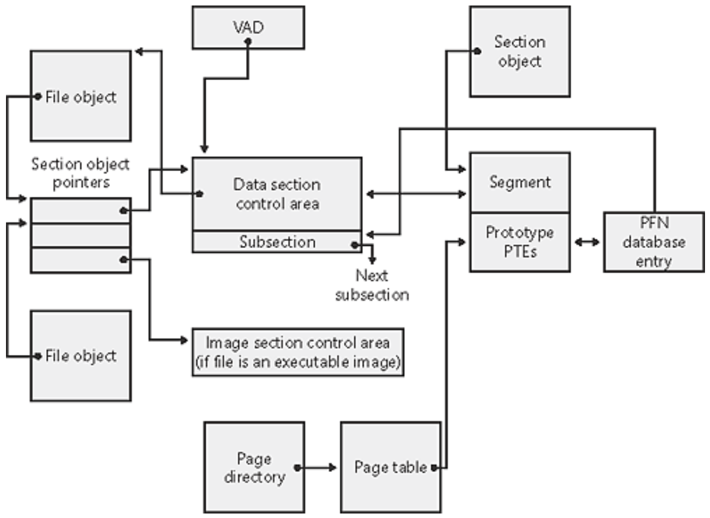

Fig. 4. Memory Management Objects for Mapped Files [38].

One purpose of this large collection of different objects is to avoid multiple copies of the same data in memory. On the one hand, this allows inter-process communication through memory-mapped files, and, on the other hand, it allows reduction of the amount of consumed memory. This is especially effective in the case of system libraries, which have to be mapped into all running processes.

Virtual Address Descriptors For managing the user space memory, Windows maintains a tree of *virtual address descriptors* (VAD) per process. For each block of consecutive memory addresses that share the same memory-related settings, one VAD entry exists. Each of those entries stores the following information about the corresponding memory:

– start and end address
– protection settings, e.g. read-only, writable, executable
– is the memory backed by a file or private memory?
– for file backed memory information about the associated file

While all of the following objects in this section are solely related to memory mapped files, the VAD tree and entries are also used for private memory.

Sections Every memory-mapped file is associated with a *section* object, which has to be created before its content can be mapped. Sections are managed system-wide and can be associated with a globally valid name, which enables different processes to map the same section object into their virtual address space while using the same physical frames. Obviously, the same process can also map each section multiple times into its address space into different address regions.

A section objects stores a pointer to the related *segment* object plus a lot of flags that further describe the associated file, e.g. if it is a data or an image file or if it is stored on a network or on a floppy device. The latter information is used for caching strategies, e.g. an application loaded from a floppy disc should still be executable when the disc has been removed from the drive.

Segments Windows uses *segment objects* to manage and represent the content of mapped files. If a section from a file is created for the first time, a segment object is created. If the same process or a different one maps the same file again subsequently, that first created segment object will be used and no new one will be created anymore. Since a segment can be used either to represent a mapped data file *or* a mapped image file, it is possible that two different segment objects exist for one file. This is the case when a file is mapped first as a data file and then as an image or vice versa. One essential part of the segment structure is an array of *prototype PTEs* (PPTE), that further describe the characteristics of the mapped file content.

Prototype PTEs Prototype PTEs are used by the operating system for maintaining shared memory. Their structure is quite similar to real PTEs, but they are never used by the hardware, but solely by the OS. When a file is mapped for the first time, no PTEs are actually created, but only an array of PPTEs is set up. On the actual first access to the related memory, the PPTEs are used as a template and real PTEs are created from them. If afterwards the physical memory has to be reused and the content is paged out to the hard disk, the associated PTEs become invalid and are modified to point to their *parenting* PPTE. This enables the memory manager to efficiently manage areas of shared memory: when such an invalid page becomes valid again and the memory manager maps it to a different physical frame than before, only the contents of the PPTE has to be updated instead of enumerating all processes and modifying probably existing PTEs in their address spaces.

Control Areas *Control areas* (CA) are used to store detailed information about the different parts of a segment in a list of *subsections*. Besides this list, the CA also maintains usage counters, e.g. how often this segment currently is mapped into the address space of the running processes.

Subsections For each mapped file there are one ore more *subsection* objects which store important mapping data about the different regions of it. For data files there normally is only one subsection, since the complete address range has the same characteristics, but for image files there in general are multiple subsections: one for each

PE section plus one for the PE header. This is due to the different characteristics of PE sections, e.g. some may be read-only while others are writable or executable. These differences are reflected by the PPTEs which are created for the pages that belong to each subsection. As mentioned before, the PPTEs for the complete file are stored in one array with the segment object. Each subsection stores a pointer to the first related PPTE in its field *SubsectionBase*. The number of PPTEs associated to each subsection results from the number of pages which are necessary to map its full content to the memory. One example mapping is shown in Figure 5.

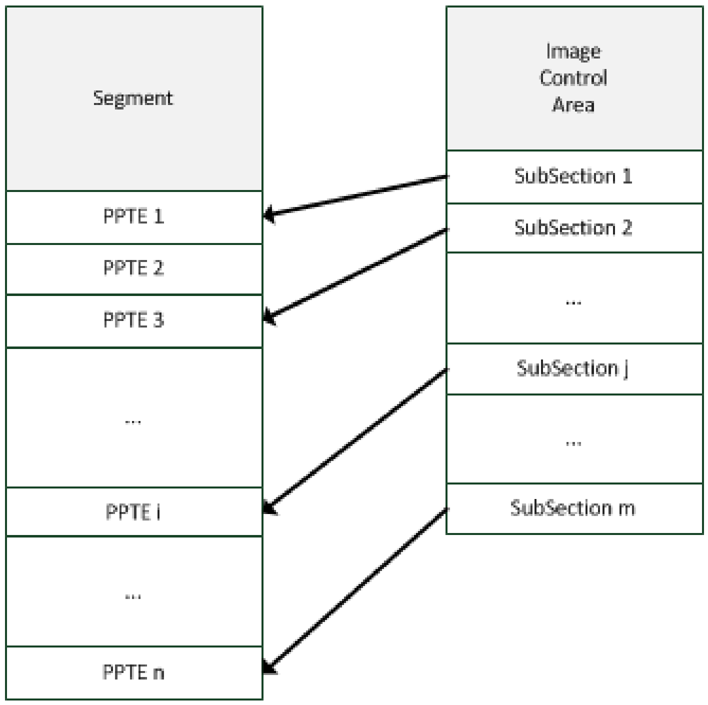

Fig. 5. Subsections and Associated PPTEs.

Section Object Pointers Due to the different mapping and usage characteristics of data files and image files, different segments and, hence, control areas are used. If for example a file is first mapped as a data file, a corresponding *data section control area* is created. If then the same file is mapped as an image, an *image section control area* is created as well. Both objects are of the same type, but there are some differences. The most obvious one is that for data files normally only one subsection is created, while for image files the number of subsections equals the number of PE sections in the related file plus one. In fact, Windows internally maps each executable which is about to be loaded first as a data file and then in a second step as an image. This results in the creation of two different control areas, from which either is used depending on the type of the created view.

To maintain these different control areas per file Windows stores one unique array for each opened file that contains pointer to the related *data* and *image* control areas. Obviously, either of these two pointers may be zero, but not both of them. This array is called *section object pointers* and is pointed to by each file object.

Summary In order to recap the inter-relationship between all the different mentioned memory objects, we give a short explanatory summary. Each process may create multiple file objects, which are only accessible from within the same process. Of course, it is possible to open the same file several times from different processes, but then each time a new file object is created. If a file should be mapped into the address space, a process has to create an associated section object first. Since a section object may be given a system-wide name, it is possible to use it from different processes as well.

For each section that is created for a data file, *one* segment and *one* control area is created as well. No matter how many section objects are created for one file hereafter, and no matter if this is done from one or from different processes, always the same identical and initially created segment and control area objects are used. For sections that are mapped as image files also *one* segment and *one* control area are created, which are different from those for the data file. Accordingly, if a file is mapped as data file *and* as an image file, two instances of segments and control area exist, one of each for the data file mapping and the others for the image file mapping. Pointers to these control areas are stored in the section object pointers array.

4.2 Windows Paging

In the preceding section we have presented the memory related objects which are used by Windows to implement memory mapped files and private memory on a higher abstraction level. As explained in section 2.1 contemporary operating systems use page directories and page tables to manage their virtual address space. The following section will explain how the described high level memory objects are implemented on a lower level.

Datastructures On a 32 bit Windows machine with PAE-kernel a PTE is 64 bits in size and its least significant bit specifies if the entry is valid or not. If this validity-bit is cleared, the PTE can not be used by the MMU and a page fault will be invoked when it is accessed. Those invalid PTEs are called *software PTEs* (in contrast to valid *hardware PTEs*) and all other bits of them can be used by the OS arbitrarily. Windows know several software PTE types and their exact subtype is determined by some of the other fields of the general data type *MMPTE_SOFTWARE* (Listing 1.1).

```
+0x000 Valid          : Pos 0, 1 Bit
+0x000 PageFileLow    : Pos 1, 4 Bits
+0x000 Protection     : Pos 5, 5 Bits
```

```
+0x000 Prototype      : Pos 10, 1 Bit
+0x000 Transition     : Pos 11, 1 Bit
+0x000 Unused         : Pos 12, 20 Bits
+0x000 PageFileHigh   : Pos 32, 32 Bits
```

Listing 1.1. Type MMPTE_SOFTWARE

Depending on these field values, one of the subtypes *Zero PTE*, *Demand Zero PTE*, *Transition PTE*, *Pagefile PTE* or **Prototype PTE* is determined and used. Table 1 shows all the different possibilities.

Subtype	Condition
Zero	all bits of the PTE are cleared
Demand Zero	Prototype=0, Transition=0,PagefileLow=0, PagefileHigh=0
Transition	Prototype=0, Transition=1
Pagefile	Prototype=0, Transition=0,PagefileLow≠0, PagefileHigh≠0
*Prototype	Prototype=1

Table 1. Software PTEs

Zero PTEs Zero PTEs specify entries which are not initialized yet. Windows uses a special form of paging, called *demand paging* in which new allocated memory is not initialized instantly, but just filled with zeroes. On the attempt to actually access this memory, the PTE has to be initialized properly with the correct values. On that occasion the page fault handler looks up the related VAD entry to decide how to further initialize the PTE.

Demand Zero PTEs Demand Zero PTEs are also used for *demand paging*. When empty memory is allocated, again actually no memory is cleared at once, but only specially crafted PTEs are set up. When later on the related memory is accessed, the PF handler simply takes the necessary physical frames from an internal list of already cleared memory.

Transition PTEs Transition PTEs mark entries which just have become invalid. Depending on their state, they may have to be written back to the related mapped (page)file or they will be discarded. For performance reasons Windows does not remove invalid pages from the physical memory at once, but they remain for some time for the case that they may be needed again.

Pagefile PTEs Pagefile PTEs specify memory which is currently swapped out to one of the page files. The exact page file and the related position within is determined by the fields *PagefileLow* and *PagefileHigh*.

**Prototype PTEs* *Prototype PTEs are used for memory which is associated with a section object. There often is a misunderstanding with that particular type of PTE. PPTE themselves are never contained within any page table, but they are stored with the segment objects in some special memory region of the kernel space. A PTE that points to a PPTE has the *Prototype*-field set (like shown in 1), but it is not a prototype PTE itself. Therefore, we refer to this kind of PTE by the name *PPTE.

Algorithms There are different system services of Windows related to memory management, some are called actively when a memory related operation is executed and others are always running in the background, e.g. the system threads which periodically write back modified pages to the hard disk or zero out pages that have become free. For our purposes the following operations are important, hence we will take a further look at them:

- allocation of private memory
- memory-mapping of files, either image- or data-files
- modification of page protection settings
- page fault handling

If we are dealing with private memory, the situation is rather simple. All possible execution pathes to allocate memory lead to **NtAllocateVirtualMemory**. This system routine first performs some validity checks on the specified parameters, then optionally attaches to the address space of a different process, if such has been specified as target. After that, if a particular target memory address has been specified, it checks if the related address range is available. If not a suitable memory region is determined from the list of free memory. Then a VAD entry is created and initialized with the specified page protection settings. Finally, all related PTEs are zeroed out, such that the PF handler will be called to set them up properly on their first access.

In the case of file-backed memory, things are more sophisticated. For mapping a file into memory, first a section objects has to be created. This can be done by calling the native API function **NtCreateSection**, which is a thin wrapper around **MmCreateSection**. If the file is mapped for the first time, a new segment object and control area are created first. Then, depending on the fact if the section is created for a page-, data- or an image-file, different actions are taken:

- **MiCreateDataFileMap** is called for data files
- **MiCreateImageFileMap** is called for images
- **MiCreatePagingFileMap** is called for page file mapped memory

After returning from one of those functions, the final section object is created and initialized accordingly. Notice that the creation of a section includes setting up the segment, the control area, the subsections and the PPTEs (except for data files), but does not create any PTEs.

While mapping a data file, the main purpose of **MiCreateDataFileMap** is to setup the subsection object. In the normal case only one subsection is created, but under some special conditions multiple subsections are used, e.g. if the file is very large. For data files, the subsection field *SubsectionBase* is left blank. This defers the creation of PPTE until the the section is mapped into the memory and finally accessed for the first time. The reasoning behind this is to avoid wasting memory when very large data files are mapped. Instead the *SegmentPteTemplate* field of the segment object is setup properly which can be used to create the PPTEs subsequently if necessary.

Things are a bit different on invocation of **MiCreateImageFileMap**. First the PE header of the specified file is loaded and verified. Then one subsection is created for the PE header and one for each PE section of the image file. There is one exception to this: if a very small file is mapped, only one subsection is used for the complete file, no matter how many PE sections exist. Besides the subsections also the related PPTEs for each of them are created and their page protection flags are set according to the protection settings of the related PE section. These PPTEs will be used as a template for building the real PTEs, when the image section is mapped afterwards.

The page file can be used as base for file-mapped memory as well. In such cases, during the creation of the related section object, the function **MiCreatePagingFileMap** is called. This one first sets up and initializes one subsection and then creates all related PPTEs. For those entries the protection setting specified when calling **NtCreateSection** is used. After initialization a pagefile-backed section object is handled like a data-file section object with only little differences. One difference exist with respect to the creation of PPTEs, which is done immediately for paging-files, but deferred for regular data-files.

After a section is created, it can be mapped into the address space by creating a *view* from it. There can be different views for the same section with different protection flags mapped into the memory of one or multiple processes. All of them will use the same identical physical memory. Obviously, modifications in one of those views are instantly reflected in the other views as well. For creating a view the API **NtMapViewOfSection** is called, which leads to **MmMapViewOfSection**. Depending on the type of the underlying section, either **MiMapViewOfImageSection** or **MiMapViewOfDataSection** is called.

MiMapViewOfDataSection performs the following steps:

- the helper routine **MiAddViewsForSection** is called
- if a base address was specified, it is checked for availability, otherwise a sufficient available memory range is determined
- finally the VAD entry is built using the protection setting specified when calling **NtMapViewOfSection**

MiAddViewsForSection enumerates all subsections and checks if the PPTEs for those already have been created, e.g. the section has been mapped before. If not, the PPTEs are initialized from the *SegmentPteTemplate* of the associated segment.

MiMapViewOfImageSection is very similar to **MiMapViewOfDataSection**: first it is checked if a base address was specified and if the memory is unoccupied yet. If no base is given, an appropriate memory range is selected instead. After that, a VAD entry is created and initialized. No PPTEs have to be created in this case, since they were already initialized when the section object was created. It should be noticed that there is only one VAD entry for the complete image section, though it normally consists of several subsections with different page protections. Therefore, in contrast to a data section, the VAD protection setting for an image file is rather meaningless. Windows will always take a look into the related PPTEs to gather protection information about the associated memory.

Sometimes the protection of already allocated memory has to be modified, either for private or for file-mapped memory. For this task Windows offer the **NtProtectVirtualMemory** API which is a wrapper for **MiProtectVirtualMemory**. This routine has two different execution pathes, depending on the fact if the specified memory is related to a mapped file or if it is private. In the first case the helper routine **MiSetProtectionOnSection** is called, otherwise a loop over all related PTEs is performed and each one is modified by setting the new protection value.

MiSetProtectionOnSection first obtains the current protection of the specified section. This is done by either getting it from the first PTE (if it is valid), from the VAD entry (if it a data file) or from the PPTE (if it is an image file). After that, it enumerates all PTEs in the range of the section and sets their protection to the new value. If some of the PTEs are not valid yet, they are initialized as *PPTEs. For all PTEs that already are valid, the protection flag are modified directly.

Like explained in section 2.1, a page fault occurs if the MMU is not able to perform the translation from a virtual to a physical address on its own. This may happen due to one of the following reasons:

- the related PTE or PDE is invalid
- the requested operation violates the PTE protection settings
- a kernelmode page is tried to be accessed from usermode
- one of the special PTE flags is set which triggers the invocation of the page fault handler

In any of those events a *page fault exception* is triggered. On intel machines thus is is realized by the *interrupt 0xE*, which is handled by **KiTrap0E** under Windows. This trap handler is mainly a wrapper around the system function **MmAccessFault**. This system routine handles both, page faults in user space as well as in kernel space memory. Since we are only interested in user space page faults, those parts of the handler which are related to kernel memory are not further discussed here.

The first action of **MmAccessFault** is to ensure that the related PDE is valid. If it is not currently mapped into physical memory, the dispatcher function **MiDispatchFault** is called to bring it in. After that there are two different cases, depending on the validity flag of the related PTE. In case of a valid PTE, most probably a protection fault has been occurred. Otherwise the PTE is invalid and the PF handler has to determine what kind of data should be mapped into the accessed memory region and take action accordingly.

If the PTE is valid, the following actions are taken:

- if a *write* operation has caused the fault
  - if the *copy-on-write* flag has been set in the PTE, the **MiCopyOnWrite** function is called to handle the fault
  - if the page is not writable at all, an *access violation* (AV) exception is raised
- if an *execute* operation has caused the fault and the particular page is not executable
  - if DEP is deactivated globally or for this process, the PTE is modified to be executable, such that it will not fault the next time

- otherwise an AV exception is raised

For an invalid PTE different actions have to be taken depending on the particular subtype of the software PTE. First it is checked, if there is a VAD related to the specified address. If this is not the case, an invalid memory address was specified and thus an AV exception is raised. Otherwise, and after a few more special case checks, the page fault dispatcher **MiDispatchFault** is called which, depending on the further PTE type, calls one of the helper functions **MiResolveTransitionFault**, **MiResolveDemandZeroFault**, **MiResolvePageFileFault** or **MiResolveProtoPteFault**. Each of these routines handles the fault in its particular way and finally makes the faulting PTE *valid* and sets its memory protection flags appropriately. In case of a page fault for a *PPTEs, the functions **MiResolveProtoPteFault** and **MiCompleteProtoPteFault** are invoked. Inside the latter one the faulting and invalid PTE is created from the associated PPTE of the subsection which is mapped to that particular memory address.

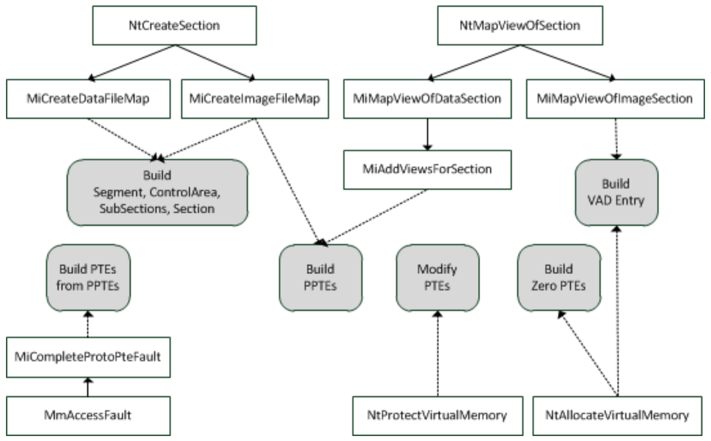

Fig. 6. Windows Paging Functions.

What we care most about, when it comes to bringing a page into physical memory, are the protection flags. In case of private memory, the appropriate flags are always taken directly from the VAD entry. For file backed memory, on the other hand, the settings may come from the PPTE or from the invalid PTE itself. If a section is mapped for the first time, the protection setting of the PPTE will be used. If then the memory protection of an already mapped subsection is modified, the new protection value is stored within the PTE directly. If then the related memory is paged out and later on paged in again, the memory protection will not be restored from the PPTE but from the PTE instead.

Figure 6 gives a schematic overview of the functions described in this section and illustrates when which related memory objects are built or accessed.

5 Conclusions

In this work we have presented some unpublished facts about the internals of the Windows memory management. In particular we have examined the various existing memory management objects, the different kinds of software PTEs and then focused on the involved internal algorithms. The knowledge resulting from this paper gives us a better understanding of the underlying operating system functionality and will assist us in developing better protection solutions in the future.

Acknowledgments

Thanks to Felix Freiling for reading earlier versions of this document and making suggestions for improvements.

References

1. Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.
2. J. Bergeron, Mourad Debbabi, M. M. Erhioui, and Béchir Ktari. Static analysis of binary code to isolate malicious behaviors. In *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, WETICE '99, pages 184–189, Washington, DC, USA, 1999. IEEE Computer Society.
3. Dionysus Blazakis. Interpreter exploitation. In *Proceedings of the 4th USENIX conference on Offensive technologies*, WOOT'10, pages 1–9, 2010.
4. Dionysus Blazakis. Interpreter exploitation: Pointer inference and jit spraying. In *Blackhat DC*, 2010.
5. Feliam's Blog. Filling adobe's heap. `http://feliam.wordpress.com/2010/02/15/filling-adobes-heap/`, 2010.
6. Lutz Boehne. Pandora's Bochs: Automated Unpacking of Malware. Diploma Thesis, University of Mannheim, January 2008.
7. Erik Buchanan, Ryan Roemer, Hovav Shacham, and Stefan Savage. When good instructions go bad: generalizing return-oriented programming to risc. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 27–38, New York, NY, USA, 2008. ACM.
8. Mihai Christodorescu and Somesh Jha. Static analysis of executables to detect malicious patterns. In *In Proceedings of the 12th USENIX Security Symposium*, pages 169–186, 2003.
9. Mihai Christodorescu, Somesh Jha, Sanjit A. Seshia, Dawn Song, and Randal E. Bryant. Semantics-aware malware detection. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.
10. Cristina Cifuentes, Mike Van Emmerik, Norman Ramsey, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Sun Microsystems, Inc., Mountain View, CA, USA, 2002.
11. Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, University of Auckland, July 1997.
12. National Vulnerability Database. Cve-2010-3971. `http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2010-3971`, 2010.

13. Solar Designer. "return-to-libc" attack. *Bugtraq*, August 1997.
14. Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 51–62, New York, NY, USA, 2008. ACM.
15. Tyler Durden. Bypassing pax aslr protection. `http://www.phrack.org/archives/59/p59_0x09_Bypassing%20PaX%20ASLR%20protection_by_Tyler%20Durden.txt`, 2010.
16. Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Song. Dynamic spyware analysis. In *2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, pages 18:1–18:14, Berkeley, CA, USA, 2007. USENIX Association.
17. Peter Ferrie. Attacks on virtual machine emulators. `http://pferrie.tripod.com/papers/attacks.pdf`, 2007.
18. Gabriela Limon Garcia. Forensic physical memory analysis: an overview of tools and techniques. Helsinki University of Technology, 2007.
19. Goppit. Portable Executable File Format - A Reverse Engineer View. *Code Breakers Journal*, Vol. 2(No. 3), 2005.
20. Intel Corporation. Intel: 64 and IA-32 Architectures Software Developer's Manual. Specification, Intel, 2007. `http://www.intel.com/products/processor/manuals/index.htm`.
21. Sebastian Krahmer. x86-64 buffer overflow exploits and the borrowed code chunks exploitation techniques. `http://www.suse.de/~krahmer/no-nx.pdf`.
22. Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries, 2004.
23. Christopher Kruegel, William Robertson, and Giovanni Vigna. Detecting kernel-level rootkits through binary analysis. In *Proceedings of the 20th Annual Computer Security Applications Conference*, ACSAC '04, pages 91–100, Washington, DC, USA, 2004. IEEE Computer Society.
24. Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, CCS '03, pages 290–299, New York, NY, USA, 2003. ACM.
25. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *SIGPLAN Not.*, 40:190–200, June 2005.
26. Thorsten Holz Markus Engelberth, Carsten Willems. Maloffice - detecting malicious documents with combined static and dynamic analysis. In *Proceedings of Virus Bulletin 2009*, 2009.
27. John Mcdonald and Chris Valasek. Practical windows xp/2003 heap exploitation, 2009.
28. Microsoft. Microsoft portable executable and common object file format specification. `http://www.microsoft.com/whdc/system/platform/firmware/pecoff.mspx`.
29. Mihai Christodorescu and Johannes Kinder and Somesh Jha and Stefan Katzenbeisser and Helmut Veith and Technische Universitaet Muenchen. Malware normalization. Technical report, University of Wisconsin, Madison, 2005.
30. Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *In Proceedings of the 2007 IEEE Symposium on Security and Privacy, Oakland*, pages 231–245, 2007.
31. MSDN. /safeseh (image has safe exception handlers). `http://msdn.microsoft.com/en-US/library/9a89h429%28v=VS.80%29.aspx`.
32. Tilo Mueller. "ASLR" smack and laugh reference. In *Seminar on Advanced Exploitation Techniques*. Chair of Computer Science 4, RWTH Aachen, Germany, February 2008.

33. Susanta Nanda, Wei Li, Lap-Chung Lam, and Tzi-cker Chiueh. Bird: Binary interpretation using runtime disassembly. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 358–370, Washington, DC, USA, 2006. IEEE Computer Society.
34. Gary Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.
35. Gerardo Richarte. Four different tricks to bypass stackshield and stackguard protection. *World Wide Web*, 1, 2002.
36. Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Proceedings of the 22nd Annual Computer Security Applications Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.
37. Nicolas Ruff. Windows memory forensics. *Journal in Computer Virology*, 4(2):83–100, 2008.
38. Mark E. Russinovich, David Solomon, and Alex Ionescu. *Windows Internals: Book and Online Course Bundle*. C.B.Learning, United Kingdom, 5th edition, 2010.
39. Sven B. Schreiber. *Undocumented Windows 2000 secrets: a programmer's cookbook*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.
40. Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In Sabrina De Capitani di Vimercati and Paul Syverson, editors, *Proceedings of CCS 2007*, pages 552–61. ACM Press, October 2007.
41. Alexey Sintsov. Writing jit-spray shellcode for fun and profit. `http://dsecrg.com/pages/pub/show.php?id=22`, 2010.
42. Alexander Sotirov and Mark Dowd. Bypassing browser memory protections. In *In Proceedings of BlackHat*, 2008.
43. Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for windows rootkit detection. *Phrack*, 11, No. 63, August 2005.
44. Joe Stewart. Ollybone: Semi-automatic unpacking on ia-32. *Defcon 14*, 2006.
45. PaX Team. Documentation for the pax project - overall description. `http://pax.grsecurity.net/docs/pax.txt`, 2008.
46. Amit Vasudevan and Ramesh Yerraballi. Spike: engineering malware analysis tools using unobtrusive binary-instrumentation. In *Proceedings of the 29th Australasian Computer Science Conference - Volume 48*, ACSC '06, pages 311–320, Darlinghurst, Australia, Australia, 2006. Australian Computer Society, Inc.
47. Redmond Wa, Galen Hunt, Galen Hunt, Doug Brubacher, and Doug Brubacher. Detours: Binary interception of win32 functions. In *In Proceedings of the 3rd USENIX Windows NT Symposium*, pages 135–143, 1998.
48. Perry Wagle and Crispin Cowan. Stackguard: Simple stack smash protection for gcc. In *Proc. of the GCC Developers Summit*, pages 243–255, 2003.
49. Andrew Walenstein, Rachit Mathur, Mohamed R. Chouchane, and Arun Lakhotia. Normalizing metamorphic malware using term rewriting. In *Proceedings of the Sixth IEEE International Workshop on Source Code Analysis and Manipulation*, pages 75–84, Washington, DC, USA, 2006. IEEE Computer Society.
50. Ollie Whitehouse. Analysis of gs protections in microsoft windows vista. *Symantec Advanced Threat Research*, 2007.
51. Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security and Privacy*, 5:32–39, March 2007.