A Catalog of Windows Local Kernel-mode Backdoor Techniques

August, 2007

skape
mmiller@hick.org

Skywing
Skywing@valhallalegends.com

Contents

Abstract

This paper presents a detailed catalog of techniques that can be used to create local kernel-mode backdoors on Windows. These techniques include function trampolines, descriptor table hooks, model-specific register hooks, page table modifications, as well as others that have not previously been described. The majority of these techniques have been publicly known far in advance of this paper. However, at the time of this writing, there appears to be no detailed single point of reference for many of them. The intention of this paper is to provide a solid understanding on the subject of local kernel-mode backdoors. This understanding is necessary in order to encourage the thoughtful discussion of potential countermeasures and perceived advancements. In the vein of countermeasures, some additional thoughts are given to the common misconception that PatchGuard, in its current design, can be used to prevent kernel-mode rootkits.

1 Introduction

The classic separation of privileges between user-mode and kernel-mode has been a common feature included in most modern operating systems. This separation allows operating systems to make security guarantees relating to process isolation, kernel-user isolation, kernel-mode integrity, and so on. These security guarantees are needed in order to prevent a lesser privileged user-mode process from being able to take control of the system itself. A kernel-mode backdoor is one method of bypassing these security restrictions.

There are many different techniques that can be used to backdoor the kernel. For the purpose of this document, a backdoor will be considered to be something that provides access to resources that would otherwise normally be restricted by the kernel. These resources might include executing code with kernel-mode privileges, accessing kernel-mode data, dis-

abling security checks, and so on. To help further limit the scope of this document, the authors will focus strictly on techniques that can be used to provide local backdoors into the kernel on Windows. In this context, a local backdoor is a backdoor that does not rely on or make use of a network connection to provide access to resources. Instead, local backdoors can be viewed as ways of weakening the kernel in an effort to provide access to resources from non-privileged entities, such as user-mode processes.

The majority of the backdoor techniques discussed in this paper have been written about at length and in great detail in many different publications[20, 8, 12, 18, 19, 21, 25, 26]. The primary goal of this paper is to act as a point of reference for some of the common, as well as some of the not-so-common, local kernel-mode backdoor techniques. The authors have attempted to include objective measurements for each technique along with a description of how each technique works. As a part of defining these objective measurements, the authors have attempted to research the origins of some of the more well-known backdoor techniques. Since many of these techniques have been used for such a long time, the origins have proven somewhat challenging to uncover.

The structure of this paper is as follows. In §2, each of the individual techniques that can be used to provide a local kernel-mode backdoor are discussed in detail. §3 provides a brief discussion into general strategies that might be employed to prevent some of the techniques that are discussed. §4 attempts to refute some of the common arguments against preventing kernel-mode backdoors in and of themselves. Finally, §5 attempts to clarify why Microsoft's PatchGuard should not be considered a security solution with respect to kernel-mode backdoors.

2 Techniques

To help properly catalog the techniques described in this section, the authors have attempted to include objective measurements of each technique. These measurements are broken down as follows:

- **Category**

  The authors have chosen to adopt Joanna Rutkowska's malware categorization in the interest of pursuing a standardized classification[34]. This model describes three types of malware. Type 0 malware categorizes non-intrusive malware; Type I includes malware that modifies things that should otherwise never be modified (code segments, MSRs, etc); Type II includes malware that modifies things that should be modified (global variables, other data); Type III is not within the scope of this document[33, 34].

  In addition to the four malware types described by Rutkowska, the authors propose Type IIa which would categorize writable memory that should effectively be considered write-once in a given context. For example, when a global DPC is initialized, the `DpcRoutine` can be considered write-once. The authors consider this to be a derivative of Type II due to the fact that the memory remains writable and is less likely to be checked than that of Type I.

- **Origin**

  If possible, the first known instance of the technique's use or some additional background on its origin is given.

- **Capabilities**

  The capabilities the backdoor offers. This can be one or more of the following: kernel-mode code execution, access to kernel-mode data, access to restricted resources. If a technique allows kernel-mode code execution, then it implicitly has all other capabilities listed.

- **Considerations**

  Any restrictions or special points that must be made about the use of a given technique.

- **Covertness**

  A description of how easily the use of a given technique might be detected.

Since many of the techniques described in this document have been known for quite some time, the authors have taken a best effort approach to identifying sources of the original ideas. In many cases, this has proved to be difficult or impossible. For this reason, the authors request that any inaccuracy in citation be reported so that it may be corrected in future releases of this paper.

2.1 Image Patches

Perhaps the most obvious approach that can be used to backdoor the kernel involves the modification of code segments used by the kernel itself. This could include modifying the code segments of kernel-mode images like `ntoskrnl.exe`, `ndis.sys`, `ntfs.sys`, and so on. By making modifications to these code segments, it is possible to hijack kernel-mode execution whenever a hooked function is invoked. The possibilities surrounding the modification of code segments are limited only by what the kernel itself is capable of doing.

2.1.1 Function Prologue Hooking

Function hooking is the process of intercepting calls to a given function by redirecting those calls to an alternative function. The concept of function hooking has been around for quite some time and it's unclear who originally presented the idea. There are a number of different libraries and papers that exist which help to facilitate the hooking of functions[21]. With respect to local kernel-mode backdoors, function hooking is an easy and reliable method of creating a backdoor. There are a few different ways in which functions can be hooked. One of the most common techniques involves overwriting the prologue of the function to be hooked with an architecture-specific jump instruction that transfers control to an alternative function somewhere else in memory. This is the approach taken by Microsoft's Detours library[21]. While prologue hooks are conceptually simple, there is actually quite a bit of code needed to implement them properly.

In order to implement a prologue hook in a portable and reliable manner, it is often necessary to make use of a disassembler that is able to determine the size, in bytes, of individual instructions. The reason for this is that in order to perform the prologue overwrite, the first few bytes of the function to be hooked must be overwritten by a control transfer instruction (typically a jump). On the Intel architecture, control transfer instructions can have one of three operands: a register, a relative offset, or a memory operand. Each operand type controls the size of the jump instruction that will be needed: 2 bytes, 5 bytes, and 6 bytes, respectively. The disassembler makes it possible to copy the first n instructions from the function's prologue prior to performing the overwrite. The value of n is determined by disassembling each instruction in the prologue until the number of bytes disassembled is greater than or equal to the number of bytes that will be overwritten when hooking the function.

The reason the first n instructions must be saved in their entirety is to make it possible for the original function to be called by the hook function. In order to call the original version of the function, a small stub of code must be generated that will execute the first n instructions of the function's original prologue followed by a jump to instruction $n + 1$ in the original function's body. This stub of code has the effect of allowing the original function to be called without it being diverted by the prologue overwrite. This method of implementing function prologue hooks is used extensively by Detours and other hooking libraries[21].

Recent versions of Windows, such as XP SP2 and Vista, include image files that come with a more elegant way of hooking a function with a function prologue overwrite. In fact, these images have been built with a compiler enhancement that was designed specifically to improve Microsoft's ability to hook its own functions during runtime. The enhancement involves creating functions with a two byte no-op instruction, such as a `mov edi, edi`, as the first instruction of a function's prologue. In addition to having this two byte instruction, the compiler also prefixes 5 no-op instructions to the function itself.

The two byte no-op instruction provides the necessary storage for a two byte relative short jump instruction to be placed on top of it. The relative short jump, in turn, can then transfer control into another relative jump instruction that has been placed in the 5 bytes that were prefixed to the function itself. The end result is a more deterministic way of hooking a function using a prologue overwrite that does not rely on a disassembler. A common question is why a two byte no-op instruction was used rather than two individual no-op instructions. The answer for this has two parts. First, a two byte no-op instruction can be overwritten in an atomic fashion whereas other prologue overwrites, such as a 5 byte or 6 byte overwrite, cannot. The second part has to do with the fact that having a two byte no-op instruction prevents race conditions associated with any thread executing code from within the set of bytes that are overwritten when the hook is installed. This race condition is common to any type of function prologue overwrite.

To better understand this race condition, consider what might happen if the prologue of a function had two single byte no-op instructions. Prior to this function being hooked, a thread executes the first no-op instruction. In between the execution of this first no-op and the second no-op, the function in question is hooked in the context of a second thread and the first two bytes are overwritten with the opcodes associated with a relative short jump instruction, such as `0xeb` and `0xf9`. After the prologue overwrite occurs, the first thread begins executing what was originally the second no-op instruction. However, now that the function has been hooked, the no-op instruction may have been changed from `0x90` to `0xf9`. This may have disastrous effects depending on the context that the hook is executed in. While this race condition may seem unlikely, it is nevertheless feasible and can therefore directly impact the reliability of any solution that uses prologue overwrites in order to hook functions.

Category: Type I

Origin: The concept of patching code has "existed since the dawn of digital computing"[21].

Capabilities: Kernel-mode code execution

Considerations: The reliability of a function prologue hook is directly related to the reliability of the disassembler used and the number of bytes that are overwritten in a function prologue. If the two byte no-op instruction is not present, then it is unlikely that a function prologue overwrite will be able to be multiprocessor safe. Likewise, if a disassembler does not accurately count the size of instructions in relation to the actual processor, then the function prologue hook may fail, leading to an unexpected crash of the system. One other point that is worth mentioning is that authors of hook functions must be careful not to inadvertently introduce instability issues into the operating system by failing to properly sanitize and check parameters to the function that is hooked. There have been many examples where legitimate software has gone the route of hooking functions without taking these considerations into account[38].

Covertness: At the time of this writing, the use of function prologue overwrites is considered to not be covert. It is trivial for tools, such as Joanna Rutkowska's System Virginity Verifier[32], to compare the in-memory version of system images with the on-disk versions in an effort to detect in-memory alterations. The Windows Debugger (windbg) will also make an analyst aware of differences between in-memory code segments and their on-disk counterparts.

2.1.2 Disabling SeAccessCheck

In Phrack 55, Greg Hoglund described the benefits of patching `nt!SeAccessCheck` so that it never returns access denied[19]. This has the effect of causing access checks on securable objects to always grant access, regardless of whether or not the access would normally be granted. As a result, non-privileged users can directly access otherwise privileged resources. This simple modification does not directly make it possible to execute privileged code, but it does indirectly facilitate it by allowing non-

privileged users to interact with and modify system processes.

Category: Type I

Origin: Greg Hoglund was the first person to publicly identify this technique in September, 1999[19].

Capabilities: Access to restricted resources.

Covertness: Like function prologue overwrites, the `nt!SeAccessCheck` patch can be detected through differences between the mapped image of `ntoskrnl.exe` and the on-disk version.

2.2 Descriptor Tables

The x86 architecture has a number of different descriptor tables that are used by the processor to handle things like memory management (GDT), interrupt dispatching (IDT), and so on. In addition to processor-level descriptor tables, the Windows operating system itself also includes a number of distinct software-level descriptor tables, such as the SSDT. The majority of these descriptor tables are heavily relied upon by the operating system and therefore represent a tantalizing target for use in backdoors. Like the function hooking technique described in 2.1.1, all of the techniques presented in this subsection have been known about for a significant amount of time. The authors have attempted, when possible, to identify the origins of each technique.

2.2.1 IDT

The *Interrupt Descriptor Table* (IDT) is a processor-relative structure that is used when dispatching interrupts. Interrupts are used by the processor as a means of interrupting program execution in order to handle an event. Interrupts can occur as a result of a signal from hardware or as a result of software asserting an interrupt through the `int` instruction[23]. The IDT contains 256 descriptors that are associated with the 256 interrupt vectors supported by the processor. Each IDT descriptor can be one of three types of gate descriptors (task, interrupt, trap) which are used to describe where and how control should be transferred when an interrupt for a particular vector occurs. The base address and limit of the IDT are stored in the `idtr` register which is populated through the `lidt` instruction. The current base address and limit of the `idtr` can be read using the `sidt` instruction.

The concept of an IDT hook has most likely been around since the origin of the concept of interrupt handling. In most cases, an IDT hook works by redirecting the procedure entry point for a given IDT descriptor to an alternative location. Conceptually, this is the same process involved in hooking any function pointer (which is described in more detail in 2.5). The difference comes as a result of the specific code necessary to hook an IDT descriptor.

On the x86 processor, each IDT descriptor is an eight byte data structure. IDT descriptors that are either an interrupt gate or trap gate descriptor contain the procedure entry point and code segment selector to be used when the descriptor's associated interrupt vector is asserted. In addition to containing control transfer information, each IDT descriptor also contains additional flags that further control what actions are taken. The Windows kernel describes IDT descriptors using the following structure:

```
kd> dt _KIDTENTRY
   +0x000 Offset           : Uint2B
   +0x002 Selector         : Uint2B
   +0x004 Access           : Uint2B
   +0x006 ExtendedOffset   : Uint2B
```

In the above data structure, the `Offset` field holds the low 16 bits of the procedure entry point and the `ExtendedOffset` field holds the high 16 bits. Using this knowledge, an IDT descriptor could be hooked by redirecting the procedure entry point to an alternate function. The following code illustrates how this can be accomplished:

```
typedef struct _IDT
{
  USHORT          Limit;
  PIDT_DESCRIPTOR Descriptors;
```

```
} IDT, *PIDT;

static NTSTATUS HookIdtEntry(
  IN UCHAR DescriptorIndex,
  IN ULONG_PTR NewHandler,
  OUT PULONG_PTR OriginalHandler OPTIONAL)
{
  PIDT_DESCRIPTOR Descriptor = NULL;
  IDT             Idt;

  __asm sidt [Idt]

  Descriptor = &Idt.Descriptors[DescriptorIndex];

  *OriginalHandler =
    (ULONG_PTR)(Descriptor->OffsetLow +
                (Descriptor->OffsetHigh << 16));

  Descriptor->OffsetLow  =
    (USHORT)(NewHandler & 0xffff);
  Descriptor->OffsetHigh =
    (USHORT)((NewHandler >> 16) & 0xffff);

  __asm lidt [Idt]

  return STATUS_SUCCESS;
}
```

In addition to hooking an individual IDT descriptor, the entire IDT can be hooked by creating a new table and then setting its information using the `lidt` instruction.

Category: Type I; although some portions of the IDT may be legitimately hooked.

Origin: The IDT hook has its origins in *Interrupt Vector Table* (IVT) hooks. In October, 1999, Prasad Dabak et al wrote about IVT hooks[31]. Sadly, they also seemingly failed to cite their sources. It's certain that IVT hooks have existed prior to 1999. The oldest virus citation the authors could find was from 1994, but DOS was released in 1981 and it is likely the first IVT hooks were seen shortly thereafter[7]. A patent that was filed in December, 1985 entitled *Dual operating system computer* talks about IVT "relocation" in a manner that suggests IVT hooking of some form[3].

Capabilities: Kernel-mode code execution.

Covertness: Detection of IDT hooks is often trivial and is a common practice for rootkit detection tools[32].

2.2.2 GDT / LDT

The *Global Descriptor Table* (GDT) and *Local Descriptor Table* (LDT) are used to store segment descriptors that describe a view of a system's address space[1]. Segment descriptors include the base address, limit, privilege information, and other flags that are used by the processor when translating a logical address (`seg:offset`) to a linear address. Segment selectors are integers that are used to indirectly reference individual segment descriptors based on their offset into a given descriptor table. Software makes use of segment selectors through segment registers, such as `CS`, `DS`, `ES`, and so on. More detail about the behavior on segmentation can be found in the x86 and x64 system programming manuals[1].

In Phrack 55, Greg Hoglund described the potential for abusing conforming code segments[19]. A conforming code segment, as opposed to a non-conforming code segment, permits control transfers where CPL is numerically greater than DPL. However, the CPL is not altered as a result of this type of control transfer. As such, effective privileges of the caller are not changed. For this reason, it's unclear how this could be used to access kernel-mode memory due to the fact that page protections would still prevent lesser privileged callers from accessing kernel-mode pages when paging is enabled.

Derek Soeder identified an awesome flaw in 2003 that allowed a user-mode process to create an expand-down segment descriptor in the calling process' LDT[40]. An expand-down segment descriptor inverts the meaning of the limit and base address associated with a segment descriptor. In this way, the limit describes the lower limit and the base address describes the upper limit. The reason this is useful is due to the fact that when kernel-mode routines validate addresses passed in from user-mode, they assume flat segments that start at base address zero.

[1] Each processor has its own GDT

This is the same thing as assuming that a logical address is equivalent to a linear address. However, when expand-down segment descriptors are used, the linear address will reference a memory location that can be in stark contrast to the address that's being validated by kernel-mode. In order to exploit this condition to escalate privileges, all that's necessary is to identify a system service in kernel-mode that will run with escalated privileges and make use of segment selectors provided by user-mode without properly validating them. Derek gives an example of a `MOVS` instruction in the `int 0x2e` handler. This trick can be abused in the context of a local kernel-mode backdoor to provide a way for user-mode code to be able to read and write kernel-mode memory.

In addition to abusing specific flaws in the way memory can be referenced through the GDT and LDT, it's also possible to define custom gate descriptors that would make it possible to call code in kernel-mode from user-mode[23]. One particularly useful type of gate descriptor, at least in the context of a backdoor, is a call gate descriptor. The purpose of a call gate is to allow lesser privileged code to call more privileged code in a secure fashion[45]. To abuse this, a backdoor can simply define its own call gate descriptor and then make use of it to run code in the context of the kernel.

Category: Type IIa; with the exception of the LDT. The LDT may be better classified as Type II considering it exposes an API to user-mode that allows the creation of custom LDT entries (`NtSetLdtEntries`).

Origin: It's unclear if there were some situational requirements that would be needed in order to abuse the issue described by Greg Hoglund. The flaw identified by Derek Soeder in 2003 was an example of a recurrence of an issue that was found in older versions of other operating systems, such as Linux. For example, a mailing list post made by Morten Welinder to LKML in 1996 describes a fix for what appears to be the same type of issue that was identified by Derek[44]. Creating a custom gate descriptor for use in the context of a backdoor has been used in the past. Greg Hoglund described the use of call gates in the context of a rootkit in 1999[19]

Capabilities: In the case of the expand-down segment descriptor, access to kernel-mode data is possible. This can also indirectly lead to kernel-mode code execution, but it would rely on another backdoor technique. If a gate descriptor is abused, direct kernel-mode code execution is possible.

Covertness: It is entirely possible to write have code that will detect the addition or alteration of entries in the GDT or each individual process LDT. For example, PatchGuard will currently detect alterations to the GDT.

2.2.3 SSDT

The *System Service Descriptor Table* (SSDT) is used by the Windows kernel when dispatching system calls. The SSDT itself is exported in kernel-mode through the `nt!KeServiceDescriptorTable` global variable. This variable contains information relating to system call tables that have been registered with the operating. In contrast to other operating systems, the Windows kernel supports the dynamic registration (`nt!KeAddSystemServiceTable`) of new system call tables at runtime. The two most common system call tables are those used for native and GDI system calls.

In the context of a local kernel-mode backdoor, system calls represent an obvious target due to the fact that they are implicitly tied to the privilege boundary that exists between user-mode and kernel-mode. The act of hooking a system call handler in kernel-mode makes it possible to expose a privileged backdoor into the kernel using the operating system's well-defined system call interface. Furthermore, hooking system calls makes it possible for the backdoor to alter data that is seen by user-mode and thus potentially hide its presence to some degree.

In practice, system calls can be hooked on Windows using two distinct strategies. The first strategy involves using generic function hooking techniques which are described in 2.1.1. The second strategy involves using the function pointer hooking technique which is described in 2.5. Using the function pointer

hooking involves simply altering the function pointer associated with a specific system call index by accessed the system call table which contains the system call that is to be hooked.

The following code shows a very simple illustration of how one might go about hooking a system call in the native system call table on 32-bit versions of Windows[2]:

```
PVOID HookSystemCall(
  PVOID SystemCallFunction,
  PVOID HookFunction)
{
  ULONG SystemCallIndex =
    *(ULONG *)((PCHAR)SystemCallFunction+1);
  PVOID *NativeSystemCallTable =
    KeServiceDescriptorTable[0];
  PVOID OriginalSystemCall =
    NativeSystemCallTable[SystemCallIndex];

  NativeSystemCallTable[SystemCallIndex] = HookFunction;

  return OriginalSystemCall;
}
```

Category: Type I if prologue hook is used. Type IIa if the function pointer hook is used. The SSDT (both native and GDI) should effectively be considered write-once.

Origin: System call hooking has been used extensively for quite some time. Since this technique has become so well-known, its actual origins are unclear. The earliest description the authors could find was from M. B. Jones in a paper from 1993 entitled *Interposition agents: Transparently interposing user code at the system interface*[27]. Jones explains in his section on related work that he was unable to find any explicit research on the subject prior of agent-based interposition prior to his writing. However, it seems clear that system calls were being hooked in an ad-hoc fashion far in advance of this point. The authors were unable to find many of the papers cited by Jones. Plaguez appears to be one of the first (Jan, 1998) to publicly illustrate the usefulness of system call hooking in Linux with a specific eye toward security in Phrack 52[30].

Capabilities: Kernel-mode code execution.

Considerations: On certain versions of Windows XP, the SSDT is marked as read-only. This must be taken into account when attempting to write to the SSDT across multiple versions of Windows.

Covertness: System call hooks on Windows are very easy to detect. Comparing the in-memory SSDTs with the on-disk versions is one of the most common strategies employed.

2.3 Model-specific Registers

Intel processors support a special category of processor-specific registers known as *Model-specific Registers* (MSRs). MSRs provide software with the ability to control various hardware and software features. Unlike other registers, MSRs are tied to a specific processor model and are not guaranteed to be supported in future versions of a processor line. Some of the features that MSRs offer include enhanced performance monitoring and debugging, among other things. Software can read MSRs using the `rdmsr` instruction and write MSRs using the `wrmsr`[23].

This subsection will describe some of the MSRs that may be useful in the context of a local kernel-mode backdoor.

2.3.1 IA32_SYSENTER_EIP

The Pentium II introduced enhanced support for transitioning between user-mode and kernel-mode. This support was provided through the introduction of two new instructions: `sysenter` and `sysexit`[3]. When a user-mode application wishes to transition to kernel-mode, it issues the `sysenter` instruction. When the kernel is ready to return to user-mode,

[2]System call hooking on 64-bit versions of Windows would require PatchGuard to be disabled

[3]AMD processors also introduced enhanced new instructions to provide this feature

it issues the `sysexit` instruction. Unlike the the `call` instruction, the `sysenter` instruction takes no operands. Instead, this instruction uses three specific MSRs that are initialized by the operating system as the target for control transfers[23].

The `IA32_SYSENTER_CS` (`0x174`) MSR is used by the processor to set the kernel-mode `CS`. The `IA32_SYSENTER_EIP` (`0x176`) MSR contains the virtual address of the kernel-mode entry point that code should begin executing at once the transition has completed. The third MSR, `IA32_SYSENTER_ESP` (`0x175`), contains the virtual address that the stack pointer should be set to. Of these three MSRs, `IA32_SYSENTER_EIP` is the most interesting in terms of its potential for use in the context of a backdoor. Setting this MSR to the address of a function controlled by the backdoor makes it possible for the backdoor to intercept all system calls after they have trapped into kernel-mode. This provides a very powerful vantage point.

For more information on the behavior of the `sysenter` and `sysexit` instructions, the reader should consult both the Intel manuals and John Gulbrandsen's article[23, 15].

Category: Type I

Origin: This feature is provided for the explicit purpose of allowing an operating system to control the behavior of the `sysenter` instruction. As such, it is only logical that it can also be applied in the context of a backdoor. Kimmo Kasslin mentions a virus from December, 2005 that made use of MSR hooks[25]. Earlier that year in February, `fuzen_op` from rootkit.com released a proof of concept[12].

Capabilities: Kernel-mode code execution

Considerations: This technique is restricted by the fact that not all processors support this MSR. Furthermore, user-mode processes are not necessarily required to use it in order to transition into kernel-mode when performing a system call. These facts limit the effectiveness of this technique as it is not guaranteed to work on all machines.

Covertness: Changing the value of the `IA32_SYSENTER_EIP` MSR can be detected. For example, PatchGuard currently checks to see if the equivalent AMD64 MSR has been modified as a part of its polling checks[36]. It is more difficult for third party vendors to perform this check due to the simple fact that the default value for this MSR is an unexported symbol named `nt!KiFastCallEntry`:

```
kd> rdmsr 176
msr[176] = 00000000`804de6f0
kd> u 00000000`804de6f0
nt!KiFastCallEntry:
804de6f0 b923000000      mov     ecx,23h
```

Without having symbols, third parties have a more difficult time of distinguishing between a value that is sane and one that is not.

2.4 Page Table Entries

When operating in protected mode, x86 processors support virtualizing the address space through the use of a feature known as *paging*. The paging feature makes it possible to virtualize the address space by adding a translation layer between linear addresses and physical addresses[4]. To translate addresses, the processor uses portions of the address being referenced to index directories and tables that convey flags and physical address information that describe how the translation should be performed. The majority of the details on how this translation is performed are outside of the scope of this document. If necessary, the reader should consult section `3.7` of the Intel *System Programming Manual*[23]. Many other papers in the references also discuss this topic[41].

The paging system is particularly interesting due to its potential for abuse in the context of a backdoor. When the processor attempts to translate a linear address, it walks a number of page tables to determine the associated physical address. When this occurs, the processor makes a check to ensure that the

[4]When paging is not enabled, linear addresses are equivalent to physical addresses

task referencing the address has sufficient rights to do so. This access check is enforced by checking the `User/Supervisor` bit of the *Page-Directory Entry* (PDE) and *Page-Table Entry* (PTE) associated with the page. If this bit is clear, only the supervisor (privilege level 0) is allowed to access the page. If the bit is set, both supervisor and user are allowed to access the page[5].

The implications surrounding this flag should be obvious. By toggling the flag in the `PDE` and `PTE` associated with an address, a backdoor can gain access to read or write kernel-mode memory. This would indirectly make it possible to gain code execution by making use of one of the other techniques listed in this document.

Category: Type II

Origin: The modification of PDE and PTE entries has been supported since the hardware paging's inception. The authors were not able to find an exact source of the first use of this technique in a backdoor. There have been a number of examples in recent years of tools that abuse the supervisor bit in one way or another[29, 41]. PaX team provided the first documentation of their PAGEEXEC code in March, 2003. In January, 1998, Mythrandir mentions the supervisor bit in phrack 52 but doesn't explicitly call out how it could be abused[28].

Capabilities: Access to kernel-mode data.

Considerations: Code that attempts to implement this approach would need to properly support PAE and non-PAE processors on x86 in order to work reliably. This approach is also extremely dangerous and potentially unreliable depending on how it interacts with the memory manager. For example, if pages are not properly locked into physical memory, they may be pruned and thus any `PDE` or `PTE` modifications would be lost. This would result in the user-mode process losing access to a specific page.

Covertness: This approach could be considered fairly covert without the presence of some tool capable of intercepting `PDE` or `PTE` modifications. Locking pages into physical memory may make it easier to detect in a polling fashion by walking the set of locked pages and checking to see if their associated `PDE` or `PTE` has been made accessible to user-mode.

[5]This isn't always the case depending on whether or not the `WP` bit is set in `CR0`

2.5 Function Pointers

The use of function pointers to indirectly transfer control of execution from one location to another is used extensively by the Windows kernel[18]. Like the function prologue overwrite described in 2.1.1, the act of hooking a function by altering a function pointer is an easy way to intercept future calls to a given function. The difference, however, is that hooking a function by altering a function pointer will only intercept indirect calls made to the hooked function through the function pointer. Though this may seem like a fairly significant limitation, even these restrictions do not drastically limit the set of function pointers that can be abused to provide a kernel-mode backdoor.

The concept itself should be simple enough. All that's necessary is to modify the contents of a given function pointer to point at untrusted code. When the function is invoked through the function pointer, the untrusted code is executed instead. If the untrusted code wishes to be able to call the function that is being hooked, it can save the address that is stored in the function pointer prior to overwriting it. When possible, hooking a function through a function pointer is a simple and elegant solution that should have very little impact on the stability of the system (with obvious exception to the quality of the replacement function).

Regardless of what approach is taken to hook a function, an obvious question is where the backdoor code associated with a given hook function should be placed. There are really only two general memory locations that the code can be stored. It can either stored in user-mode, which would generally make it specific to a given process, or kernel-mode, which would make it visible system wide. Deciding which

of the two locations to use is a matter of determining the contextual restrictions of the function pointer being leveraged. For example, if the function pointer is called through at a raised IRQL, such as `DISPATCH`, then it is not possible to store the hook function's code in pageable memory. Another example of a restriction is the process context in which the function pointer is used. If a function pointer may be called through in any process context, then there are only a finite number of locations that the code could be placed in user-mode. It's important to understand some of the specific locations that code may be stored in

Perhaps the most obvious location that can be used to store code that is to execute in kernel-mode is the kernel pools, such as the `PagedPool` and `NonPagedPool`, which are used to store dynamically allocated memory. In some circumstances, it may also be possible to store code in regions of memory that contain code or data associated with device drivers. While these few examples illustrate that there is certainly no shortage of locations in which to store code, there are a few locations in particular that are worth calling out.

One such location is composed of a single physical page that is shared between user-mode and kernel-mode. This physical page is known as `SharedUserData` and it is mapped into user-mode as read-only and kernel-mode as read-write. The virtual address that this physical page is mapped at is static in both user-mode (`0x7ffe0000`) and kernel-mode (`0xffdf0000`) on all versions of Windows NT+[6]. There is also plenty of unused memory within the page that is allocated for `SharedUserData`. The fact that the mapping address is static makes it a useful location to store small amounts of code without needing to allocate additional storage from the paged or non-paged pool[24].

Though the `SharedUserData` mapping is quite useful, there is actually an alternative location that can be used to store code that is arguably more covert. This approach involves overwriting a function pointer with the address of some code from the virtual mapping of the native DLL, `ntdll.dll`. The native DLL is special in that it is the only DLL that is guaranteed to be mapped into the context of every process, including the `System` process. It is also mapped at the same base address in every process due to assumptions made by the Windows kernel. While these are useful qualities, the best reason for using the `ntdll.dll` mapping to store code is that doing so makes it possible to store code in a process-relative fashion. Understanding how this works in practice requires some additional explanation.

The native DLL, `ntdll.dll`, is mapped into the address space of the `System` process and subsequent processes during kernel and process initialization, respectively. This mapping is performed in kernel-mode by `nt!PspMapSystemDll`. One can observe the presence of this mapping in the context of the `System` process through a debugger as shown below. These same basic steps can be taken to confirm that `ntdll.dll` is mapped into other processes as well[7] :

```
kd> !process 0 0 System
PROCESS 81291660  SessionId: none  Cid: 0004
    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000a68
    HandleCount: 256.
    Image: System
kd> !process 81291660
PROCESS 81291660  SessionId: none  Cid: 0004
    Peb: 00000000  ParentCid: 0000
    DirBase: 00039000  ObjectTable: e1000a68
    HandleCount: 256.
    Image: System
    VadRoot 8128f288 Vads 4
...
kd> !vad 8128f288
VAD     level start end   commit
...
81207d98 ( 1) 7c900 7c9af 5 Mapped  Exe
kd> dS poi(poi(81207d98+0x18)+0x24)+0x30
e13591a8  "\WINDOWS\system32\ntdll.dll"
```

To make use of the `ntdll.dll` mapping as a location in which to store code, one must understand the

[6]The virtual mappings are no longer executable as of Windows XP SP2. However, it is entirely possible for a backdoor to alter these page permissions.

[7]The command `!vad` is used to dump the *virtual address directory* for a given process. This directory contains descriptions of memory regions within a given process.

implications of altering the contents of the mapping itself. Like all other image mappings, the code pages associated with `ntdll.dll` are marked as *Copy-on-Write* (COW) and are initially shared between all processes. When data is written to a page that has been marked with COW, the kernel allocates a new physical page and copies the contents of the shared page into the newly allocated page. This new physical page is then associated with the virtual page that is being written to. Any changes made to the new page are observed only within the context of the process that is making them. This behavior is why altering the contents of a mapping associated with an image file do not lead to changes appearing in all process contexts.

Based on the ability to make process-relative changes to the `ntdll.dll` mapping, one is able to store code that will only be used when a function pointer is called through in the context of a specific process. When not called in a specific process context, whatever code exists in the default mapping of `ntdll.dll` will be executed. In order to better understand how this may work, it makes sense to walk through a concrete example.

In this example, a rootkit has opted to create a backdoor by overwriting the function pointer that is used when dispatching IRPs using the `IRP_MJ_FLUSH_BUFFERS` major function for a specific device object. The prototype for the function that handles `IRP_MJ_FLUSH_BUFFERS` IRPs is shown below:

```
NTSTATUS DispatchFlushBuffers(
    IN PDEVICE_OBJECT DeviceObject,
    IN PIRP Irp);
```

In order to create a context-specific backdoor, the rootkit has chosen to overwrite the function pointer described above with an address that resides within `ntdll.dll`. By default, the rootkit wants all processes except those that are aware of the backdoor to simply have a no-operation occur when `IRP_MJ_FLUSH_BUFFERS` is sent to the device object. For processes that are aware of the backdoor, the rootkit wants arbitrary code execution to occur in kernel-mode. To accomplish this, the function pointer should be overwritten with an address that resides in `ntdll.dll` that contains a `ret 0x8` instruction. This will simply cause invocations of `IRP_MJ_FLUSH_BUFFERS` to return (without completing the IRP). The location of this `ret 0x8` should be in a portion of code that is rarely executed in user-mode. For processes that wish to execute arbitrary code in kernel-mode, it's as simple as altering the code that exists at the address of the `ret 0x8` instruction. After altering the code, the process only needs to issue an `IRP_MJ_FLUSH_BUFFERS` through the `FlushFileBuffers` function on the affected device object. The context-dependent execution of code is made possible by the fact that, in most cases, IRPs are processed in the context of the requesting process.

The remainder of this subsection will describe specific function pointers that may be useful targets for use as backdoors. The authors have tried to cover some of the more intriguing examples of function pointers that may be hooked. Still, it goes without saying that there are many more that have not been explicitly described. The authors would be interested to hear about additional function pointers that have unique and useful properties in the context of a local kernel-mode backdoor.

2.5.1 Import Address Table

The *Import Address Table* (IAT) of a PE image is used to store the absolute virtual addresses of functions that are imported from external PE images. When a PE image is mapped into virtual memory, the dynamic loader (in kernel-mode, this is `ntoskrnl`) takes care of populating the contents of the PE image's IAT based on the actual virtual address locations of dependent functions[8]. The compiler, in turn, generates code that uses an indirect `call` instruction to invoke imported functions. Each imported function has a function pointer slot in the IAT. In this fashion, PE images do not need to have any preconceived knowledge of where dependent PE images are

[8]For the sake of simplicity, bound imports are excluded from this explanation

going to be mapped in virtual memory. Instead, this knowledge can be postponed until a runtime determination is made.

The fundamental step involved in hooking an IAT entry really just boils down to changing a function pointer. What distinguishes an IAT hook from other types of function pointer hooks is the context in which the overwritten function pointer is called through. Since each PE image has their own IAT, any hook that is made to a given IAT will implicitly only affect the associated PE image. For example, consider a situation where both `foo.sys` and `bar.sys` import `ExAllocatePoolWithTag`. If the IAT entry for `ExAllocatePoolWithTag` is hooked in `foo.sys`, only those calls made from within `foo.sys` to `ExAllocatePoolWithTag` will be affected. Calls made to the same function from within `bar.sys` will be unaffected. This type of limitation can actually be a good thing, depending on the underlying motivations for a given backdoor.

Category: Type I; may legitimately be modified, but should point to expected values.

Origin: The origin of the first IAT hook is unclear. In January, 2000, Silvio described hooking via the ELF PLT which is, in some aspects, functionally equivalent to the IAT in PE images[35].

Capabilities: Kernel-mode code execution

Considerations: Assuming the calling restrictions of an IAT hook are acceptable for a given backdoor, there are no additional considerations that need to be made.

Covertness: It is possible for modern tools to detect IAT hooks by analyzing the contents of the IAT of each PE image loaded in kernel-mode. To detect discrepancies, a tool need only check to see if the virtual address associated with each function in the IAT is indeed the same virtual address as exported by the PE image that contains a dependent function.

2.5.2 KiDebugRoutine

The Windows kernel provides an extensive debugging interface to allow the kernel itself (and third party drivers) to be debugged in a live, interactive environment (as opposed to after-the-fact, post-mortem crash dump debugging). This debugging interface is used by a kernel debugger program (kd.exe, or WinDbg.exe) in order to perform tasks such as the inspecting the running state (including memory, registers, kernel state such as processes and threads, and the like) of the kernel on-demand. The debugging interface also provides facilities for the kernel to report various events of interest to a kernel debugger, such as exceptions, module load events, debug print output, and a handful of other state transitions. As a result, the kernel debugger interface has "hooks" built-in to various parts of the kernel for the purpose of notifying the kernel debugger of these events.

The far-reaching capabilities of the kernel debugger in combination with the fact that the kernel debugger interface is (in general) present in a compatible fashion across all OS builds provides an attractive mechanism that can be used to gain control of a system. By subverting `KiDebugRoutine` to instead point to a custom callback function, it becomes possible to surepticiously gain control at key moments (debug prints, exception dispatching, kernel module loading are the primary candidates).

The architecture of the kernel debugger event notification interface can be summed up in terms of a global function pointer (`KiDebugRoutine`) in the kernel. A number distinct pieces of code, such as the exception dispatcher, module loader, and so on are designed to call through `KiDebugRoutine` in order to notify the kernel debugger of events. In order to minimize overhead in scenarios where the kernel debugger is inactive, `KiDebugRoutine` is typically set to point to a dummy function, `KdpStub`, which performs almost no actions and, for the most part, simply returns immediately to the caller. However, when the system is booted with the kernel debugger enabled, KiDebugRoutine may be set to an alternate function, `KdpTrap`, which passes the information supplied by

the caller to the remote debugger.

Although enabling or disabling the kernel debugger has traditionally been a boot-time-only decision, newer OS builds such as Windows Server 2003 and beyond have some support for transitioning a system from a "kernel debugger inactive" state to a "kernel debugger active" state. As a result, there is some additional logic now baked into the dummy routine (`KdpStub`) which can under some circumstances result in the debugger being activated on-demand. This results in control being passed to the actual debugger communication routine (`KdpTrap`) after an on-demand kernel debugger initialization. Thus, in some circumstances, `KdpStub` will pass control through to `KdpTrap`.

Additionally, in Windows Server 2003 and later, it is possible to disable the kernel debugger on the fly. This may result in `KiDebugRoutine` being changed to refer to `KdpStub` instead of the boot-time-assigned `KdpTrap`. This behavior, combined with the previous points, is meant to show that provided a system is booted with the kernel debugger enabled it may not be enough to just enforce a policy that `KiDebugRoutine` must not change throughout the lifetime of the system.

Aside from exception dispatching notifiations, most debug events find their way to `KiDebugRoutine` via interrupt `0x2d`, otherwise known as "DebugService". This includes user-mode debug print events as well as kernel mode originated events (such as kernel module load events). The trap handler for interrupt `0x2d` packages the information supplied to the debug service interrupt into the format of a special exception that is then dispatched via `KiExceptionDispatch` (the normal exception dispatcher path for interrupt-generated exceptions). This in turn leads to `KiDebugRoutine` being called as a normal part of the exception dispatcher's operation.

Category: Type IIa, varies. Although on previous OS versions `KiDebugRoutine` was essentially write-once, recent versions allow limited changes of this value on the fly while the system is booted.

Origin: At the time of this writing, the authors are not aware of existing malware using `KiDebugRoutine`.

Capabilities: Redirecting `KiDebugRoutine` to point to a caller-controlled location allows control to be gained during exception dispatching (a very common occurrence), as well as certain other circumstances (such as module loading and debug print output). As an added bonus, because `KiDebugRoutine` is integral to the operation of the kernel debugger facility as a whole, it should be possible to "filter" the events received by the kernel debugger by manipulation of which events are actually passed on to `KdpTrap`, if a kernel debugger is enabled. However, it should be noted that other steps would need to be taken to prevent a kernel debugger from detecting the presence of code, such as the interception of the kernel debugger read-memory facilities.

Considerations: Depending on how the system global flags (`NtGlobalFlag`) are configured, and whether the system was booted in such a way as to suppress notification of user mode exceptions to the kernel debugger, exception events may not always be delivered to `KiDebugRoutine`. Also, as `KiDebugRoutine` is not exported, it would be necessary to locate it in order to intercept it. Furthermore, many of the debugger events occur in an arbitrary context, such that pointing `KiDebugRoutine` to user mode (except within ntdll space) may be considered dangerous. Even while pointing `KiDebugRoutine` to ntdll, there is the risk that the system may be brought down as some debugger events may be reported while the system cannot tolerate paging (e.g. debug prints). From a thread-safety perspective, an interlocked exchange on `KiDebugRoutine` should be a relatively synchronization-safe operation (however the new callback routine may never be unmapped from the address space without some means of ensuring that no callbacks are active).

Covertness: As `KiDebugRoutine` is a non-exported, writable kernel global, it has some inherent defenses against simple detection techniques. However, in legitimate system operation, there are only two legal values for `KiDebugRoutine`: `KdpStub`, and `KdpTrap`.

Though both of these routines are not exported, a combination of detection techniques (such as verifying the integrity of read only kernel code, and a verification that `KiDebugRoutine` refers to a location within an expected code region of the kernel memory image) may make it easier to locate blatant attacks on `KiDebugRoutine`. For example, simply setting `KiDebugRoutine` to point to an out-of-kernel location could be detected with such an approach, as could pointing it elsewhere in the kernel and then writing to it (either the target location would need to be outside the normal code region, easily detectable, or normally read-only code would have to be overwritten, also relatively easily detectable). Also, all versions of PatchGuard protect `KiDebugRoutine` in x64 versions of Windows. This means that effective exploitation of `KiDebugRoutine` in the long term on such systems would require an attacker to deal with PatchGuard. This is considered a relatively minor difficulty by the authors.

2.5.3 KTHREAD's SuspendApc

In order to support thread suspension, the Windows kernel includes a `KAPC` field named `SuspendApc` in the `KTHREAD` structure that is associated with each thread running on a system. When thread suspension is requested, the kernel takes steps to queue the `SuspendApc` structure to the thread's APC queue. When the APC queue is processed, the kernel invokes the APC's `NormalRoutine`, which is typically initialized to `nt!KiSuspendThread`, from the `SuspendApc` structure in the context of the thread that is being suspended. Once `nt!KiSuspendThread` completes, the thread is suspended. The following shows what values the `SuspendApc` is typically initialized to:

```
kd> dt -r1 _KTHREAD 80558c20
...
 +0x16c SuspendApc      : _KAPC
  +0x000 Type           : 18
  +0x002 Size           : 48
  +0x004 Spare0         : 0
  +0x008 Thread         : 0x80558c20 _KTHREAD
  +0x00c ApcListEntry   : _LIST_ENTRY [ 0x0 - 0x0 ]
  +0x014 KernelRoutine  : 0x804fa8a1 nt!KiSuspendNop
  +0x018 RundownRoutine : 0x805139ed nt!PopAttribNop
  +0x01c NormalRoutine  : 0x804fa881 nt!KiSuspendThread
  +0x020 NormalContext  : (null)
  +0x024 SystemArgument1: (null)
  +0x028 SystemArgument2: (null)
  +0x02c ApcStateIndex  : 0 ''
  +0x02d ApcMode        : 0 ''
  +0x02e Inserted       : 0 ''
```

Since the `SuspendApc` structure is specific to a given `KTHREAD`, any modification made to a thread's `SuspendApc.NormalRoutine` will affect only that specific thread. By modifying the `NormalRoutine` of the `SuspendApc` associated with a given thread, a backdoor can gain arbitrary code execution in kernel-mode by simply attempting to suspend the thread. It is trivial for a user-mode application to trigger the backdoor. The following sample code illustrates how a thread might execute arbitrary code in kernel-mode if its `SuspendApc` has been modified:

```
SuspendThread(GetCurrentThread());
```

The following code gives an example of assembly that implements the technique described above taking into account the `InitialStack` insight described in the considerations below:

```
public _RkSetSuspendApcNormalRoutine@4
_RkSetSuspendApcNormalRoutine@4 proc
  assume fs:nothing
  push  edi
  push  esi
  ; Grab the current thread pointer
  xor   ecx, ecx
  inc   ch
  mov   esi, fs:[ecx+24h]
  ; Grab KTHREAD.InitialStack
  lea   esi, [esi+18h]
  lodsd
  xchg  esi, edi
  ; Find StackBase
  repne scasd
  ; Set KTHREAD->SuspendApc.NormalRoutine
  mov   eax, [esp+0ch]
  xchg  eax, [edi+1ch]
  pop   esi
  pop   edi
  ret
_RkSetSuspendApcNormalRoutine@4 endp
```

Category: Type IIa

Origin: The authors believe this to be the first public description of this technique. Skywing is credited with the idea. Greg Hoglund mentions abusing APC queues to execute code, but he does not explicitly call out `SuspendApc`[18].

Capabilities: Kernel-mode code execution.

Considerations: This technique is extremely effective. It provides a simple way of executing arbitrary code in kernel-mode by simply hijacking the mechanism used to suspend a specific thread. There are also some interesting side effects that are worth mentioning. Overwriting the `SuspendApc`'s `NormalRoutine` makes it so that the thread can no longer be suspended. Even better, if the hook function that replaces the `NormalRoutine` never returns, it becomes impossible for the thread, and thus the owning process, to be killed because of the fact that the `NormalRoutine` is invoked at APC level. Both of these side effects are valuable in the context of a rootkit.

One consideration that must be made from the perspective of a backdoor is that it will be necessary to devise a technique that can be used to locate the `SuspendApc` field in the `KTHREAD` structure across multiple versions of Windows. Fortunately, there are heuristics that can be used to accomplish this. In all versions of Windows analyzed thus far, the `SuspendApc` field is preceded by the `StackBase` field. It has been confirmed on multiple operating systems that the `StackBase` field is equal to the `InitialStack` field. The `InitialStack` field is located at a reliable offset (`0x18`) on all versions of Windows checked by the authors. Using this knowledge, it is trivial to write some code that scans the `KTHREAD` structure on pointer aligned offsets until it encounters a value that is equal to the `InitialStack`. Once a match is found, it is possible to assume that the `SuspendApc` immediately follows it.

Covertness: This technique involves overwriting a function pointer in a dynamically allocated region of memory that is associated with a specific thread. This makes the technique fairly covert, but not impossible to detect. One method of detecting this technique would be to enumerate the threads in each process to see if the `NormalRoutine` of the `SuspendApc` is set to the expected value of `nt!KiSuspendThread`. It would be challenging for someone other than Microsoft to implement this safely. The authors are not aware of any tool that currently does this.

2.5.4 Create Thread Notify Routine

The Windows kernel provides drivers with the ability to register a callback that will be notified when threads are created and terminated. This ability is provided through the *Windows Driver Model* (WDM) export `nt!PsSetCreateThreadNotifyRoutine`. When a thread is created or terminated, the kernel enumerates the list of registered callbacks and notifies them of the event.

Category: Type II

Origin: The ability to register a callback that is notified when threads are created and terminated has been included since the first release of the WDM.

Capabilities: Kernel-mode code execution.

Considerations: This technique is useful because a user-mode process can control the invocation of the callback by simply creating or terminating a thread. Additionally, the callback will be notified in the context of the process that is creating or terminating the thread. This makes it possible to set the callback routine to an address that resides within `ntdll.dll`.

Covertness: This technique is covert in that it is possible for a backdoor to blend in with any other registered callbacks. Without having a known-good state to compare against, it would be challenging to conclusively state that a registered callback is associated with a backdoor. There are some indicators that could be used that something is odd, such as if the callback routine resides in `ntdll.dll` or if it resides in either the paged or non-paged pool.

2.5.5 Object Type Initializers

The Windows NT kernel uses an object-oriented approach to representing resources such as files, drivers, devices, processes, threads, and so on. Each object is categorized by an object type. This object type categorization provides a way for the kernel to support common actions that should be applied to objects of the same type, among other things. Under this design, each object is associated with only one object type. For example, process objects are associated with the `nt!PsProcessType` object type. The structure used to represent an object type is the `OBJECT_TYPE` structure which contains a nested structure named `OBJECT_TYPE_INITIALIZER`. It's this second structure that provides some particularly interesting fields that can be used in a backdoor.

As one might expect, the fields of most interest are function pointers. These function pointers, if non-null, are called by the kernel at certain points during the lifetime of an object that is associated with a particular object type. The following debugger output shows the function pointer fields:

```
kd> dt nt!_OBJECT_TYPE_INITIALIZER
...
   +0x02c DumpProcedure    : Ptr32
   +0x030 OpenProcedure    : Ptr32
   +0x034 CloseProcedure   : Ptr32
   +0x038 DeleteProcedure  : Ptr32
   +0x03c ParseProcedure   : Ptr32
   +0x040 SecurityProcedure : Ptr32
   +0x044 QueryNameProcedure : Ptr32
   +0x048 OkayToCloseProcedure : Ptr32
```

Two fairly easy to understand procedures are `OpenProcedure` and `CloseProcedure`. These function pointers are called when an object of a given type is opened and closed, respectively. This gives the object type initializer a chance to perform some common operation on an instance of an object type. In the case of a backdoor, this exposes a mechanism through which arbitrary code could be executed in kernel-mode whenever an object of a given type is opened or closed.

Category: Type IIa

Origin: Matt Conover gave an excellent presentation on how object type initializers can be used to detect rootkits at XCon 2005[8]. Conversely, they can also be used to backdoor the system. The authors are not aware of public examples prior to Conover's presentation. Greg Hoglund also mentions this type of approach[18] in June, 2006.

Capabilities: Kernel-mode code execution.

Considerations: There are no unique considerations involved in the use of this technique.

Covertness: This technique can be detected by tools designed to validate the state of object type initializers against a known-good state. Currently, the authors are not aware of any tools that perform this type of check.

2.5.6 PsInvertedFunctionTable

With the introduction of Windows for x64, significant changes were made to how exceptions are processed with respect to how exceptions operate in x86 versions of Windows. On x86 versions of Windows, exception handlers were essentially demand-registered at runtime by routines with exception handlers (more of a *code-based* exception registration mechanism). On x64 versions of Windows, the exception registration path is accomplished using a more *data-driven* model. Specifically, exception handling (and especially unwind handling) is now driven by metadata attached to each PE image (known as the "exception directory"), which describes the relationship between routines and their exception handlers, what the exception handler function pointer(s) for each region of a routine are, and how to unwind each routine's machine state in a completely data-driven fashion.

While there are significant advantages to having exception and unwind dispatching accomplished using a data-driven model, there is a potential performance penalty over the x86 method (which consisted of a linked list of exception and unwind handlers registered at a known location, on a per-thread basis). A specific example of this can be seen when noting that

all of the information needed for the operating system to locate and call the exception handler for purposes of exception or unwind processing was in one location (the linked list in the NT_TIB) on Windows for x86 is now scattered across all loaded modules in Windows for x64. In order to locate an exception handler for a particular routine, it is necessary to search the loaded module list for the module that contains the instruction pointer corresponding to the function in question. After the module is located, it is then necessary to process the PE header of the module to locate the module's exception directory. Finally, it is then necessary to search the exception directory of that module for the metadata corresponding to a location encompassing the requested instruction pointer. This process must be repeated for every function for which an exception may traverse.

In an effort to improve the performance of exception dispatching on Windows for x64, Microsoft developed a multi-tier cache system that speeds the resolution of exception dispatching information that is used by the routine responsible for looking up metadata associated with a function. The routine responsible for this is named RtlLookupFunctionTable. When searching for unwind information (a pointer to a RUNTIME_FUNCTION entry structure), depending on the reason for the search request, an internal first-level cache (RtlpUnwindHistoryTable) of unwind information for commonly occurring functions may be searched. At the time of this writing, this table consists of RtlUnwindex, __C_specific_handler, RtlpExecuteHandlerForException, RtlDispatchException, RtlRaiseStatus, KiDispatchException, and KiExceptionDispatch. Due to how exception dispatching operates on x64 [39], many of these functions will commonly appear in any exception call stack. Because of this it is beneficial to performance to have a first-level, quick reference for them.

After RtlpUnwindHistoryTable is searched, a second cache, known as PsInvertedFunctionTable (in kernel-mode) or LdrpInvertedFunctionTable (in user-mode) is scanned. This second-level cache contains a list of the first 0x200 (Windows Server 2008, Windows Vista) or 0xA0 (Windows Server 2003) loaded modules. The loaded module list contained within PsInvertedFunctionTable / LdrpInvertedFunctionTable is presented as a quickly searchable, unsorted linear array that maps the memory occupied by an entire loaded image to a given module's exception directory. The lookup through the inverted function table thus eliminates the costly linked list (loaded module list) and executable header parsing steps necessary to locate the exception directory for a module. For modules which are referenced by PsInvertedFunctionTable / LdrpInvertedFunctionTable, the exception directory pointer and size information in the PE header of the module in question are unused after the module is loaded and the inverted function table is populated. Because the inverted function table has a fixed size, if enough modules are loaded simultaneously, it is possible that after a point some modules may need to be scanned via loaded module list lookup if all entries in the inverted function table are in use when that module is loaded. However, this is a rare occurrence, and most of the interesting system modules (such as HAL and the kernel memory image itself) are at a fixed-at-boot position within PsInvertedFunctionTable[37].

By redirecting the exception directory pointer in PsInvertedFunctionTable to refer to a "shadow" exception directory in caller-supplied memory (outside of the PE header of the actual module), it is possible to change the exception (or unwind) handling behavior of all code points within a module. For instance, it is possible to create an exception handler spanning every code byte within a module through manipulation of the exception directory information. By changing the inverted function table cache for a module, multiple benefits are realized with respect to this goal. First, an arbitrarily large amount of space may be devoted to unwind metadata, as the patched unwind metadata need not fit within the confines of a particular image's exception directory (this is particular important if one wishes to "gift" all functions within a module with an exception handler). Second, the memory image of the module in question need not be modified, improving the resiliency of the technique against naive detection systems.

Category: Type IIa, varies. Although the entries for always-loaded modules such as the HAL and the kernel in-memory image itself are essentially considered write-once, the array as a whole may be modified as the system is running when kernel modules are either loaded or unloaded. As a result, while the first few entries of `PsInvertedFunctionTable` are comparatively easy to verify, the "dynamic" entries corresponding to demand-loaded (and possibly demand-unloaded) kernel modules may frequently change during the legitimate operation of the system, and as such interception of the exception directory pointers of individual drivers may be much less simple to detect than the interception of the kernel's exception directory.

Origin: At the time of this writing, the authors are not aware of existing malware using `PsInvertedFunctionTable`. Hijacking of `PsInvertedFunctionTable` was proposed as a possible bypass avenue for PatchGuard version 2 by Skywing[37]. Its applicability as a possible attack vector with respect to hiding kernel mode code was also briefly described in the same article.

Capabilities: The principal capability afforded by this technique is to establish an exception handler at arbitrary locations within a target module (even every code byte within a module if so desired). By virtue of creating such exception handlers, it is possible to gain control at any location within a module that may be traversed by an exception, even if the exception would normally be handled in a safe fashion by the module or a caller of the module.

Considerations: As `PsInvertedFunctionTable` is not exported, one must first locate it in order to patch it (this is considered possible as many exported routines reference it in an obvious, patterned way, such as `RtlLookupFunctionEntry`. Also, although the structure is guarded by a non-exported synchronization mechanism (`PsLoadedModuleSpinLock` in Windows Server 2008), the first few entries corresponding to the HAL and the kernel in-memory image itself should be static and safely accessible without synchronization (after all, neither the HAL nor the kernel in-memory image may be unloaded after the system has booted). It should be possible to perform an interlocked exchange to swap the exception directory pointer, provided that the exception directory shall not be modified in a fashion that would require synchronization (e.g. only appended to) after the exchange is made. The size of the exception directory is supplied as a separate value in the inverted function table entry array and would need to be modified separately, which may pose a synchronization problem if alterations to the exception directory are not carefully planned to be safe in all possible contingencies with respect to concurrent access as the alterations are made. Additionally, due to the 32-bit RVA based format of the unwind metadata, all exception handlers for a module must be within 4GB of that module's loaded base address. This means that custom exception handlers need to be located within a "window" of memory that is relatively near to a module. Allocating memory at a specific base address involves additional work as the memory cannot be in an arbitrary point in the address space, but within 4GB of the target. If a caller can query the address space and request allocations based at a particular region, however, this is not seen as a particular unsurmountable problem.

Covertness: The principal advantage of this approach is that it allows a caller to gain control at any point within a module's execution where an exception is generated without modifying any code or data within the module in question (provided the module is cached within `PsInvertedFunctionTable`). Because the exception directory information for a module is unused after the cache is populated, integrity checks against the PE header are useless for detecting the alteration of exception handling behavior for a cached module. Additionally, `PsInvertedFunctionTable` is a non-exported, writable kernel-mode global which affords it some intrinsic protection against simple detection techniques. A scan of the loaded module list and comparison of exception directory pointers to those contained within `PsInvertedFunctionTable` could reveal most attacks of this nature, however, provided that the loaded module list retains integrity. Additionally, PatchGuard version 3 appears to guard

key portions of `PsInvertedFunctionTable` (e.g. to block redirection of the kernel's exception directory), resulting in a need to bypass PatchGuard for long-term exploitation on Windows x64 based systems. This is considered a relatively minor difficulty by the authors.

2.5.7 Delayed Procedures

There are a number of features offered by the Windows kernel that allow device drivers to asynchronously execute code. Some examples of these features include *asynchronous procedure calls* (APCs), *deferred procedure calls* (DPCs), *work items*, *threading*, and so on. A backdoor can simply make use of the APIs exposed by the kernel to make use of any number of these to schedule a task that will run arbitrary code in kernel-mode. For example, a backdoor might queue a kernel-mode APC using the `ntdll.dll` trick described at the beginning of this section. When the APC executes, it runs code that has been altered in `ntdll.dll` in a kernel-mode context. This same basic concept would work for all other delayed procedures.

Category: Type II

Origin: This technique makes implicit use of operating system exposed features and therefore falls into the category of obvious. Greg Hoglund mentions these in particular in June, 2006[18].

Capabilities: Kernel-mode code execution.

Considerations: The important consideration here is that some of the methods that support running delayed procedures have restrictions about where the code pages reside. For example, a DPC is invoked at dispatch level and must therefore execute code that resides in non-paged memory.

Covertness: This technique is covert in the sense that the backdoor is always in a transient state of execution and therefore could be considered largely dormant. Since the backdoor state is stored alongside other transient state in the operating system, this technique should prove more difficult to detect when compared to some of the other approaches described in this paper.

2.6 Asynchronous Read Loop

It's not always necessary to hook some portion of the kernel when attempting to implement a local kernel-mode backdoor. In some cases, it's easiest to just make use of features included in the target operating system to blend in with normal behavior. One particularly good candidate for this involves abusing some of the features offered by Window's I/O (input/output) manager.

The I/O model used by Windows has many facets to it. For the purposes of this paper, it's only necessary to have an understanding of how it operates when reading data from a file. To support this, the kernel constructs an *I/O Request Packet* (IRP) with its `MajorFunction` set to `IRP_MJ_READ`. The kernel then passes the populated IRP down to the device object that is related to the file that is being read from. The target device object takes the steps needed to read data from the underlying device and then stores the acquired data in a buffer associated with the IRP. Once the read operation has completed, the kernel will call the IRP's completion routine if one has been set. This gives the original caller an opportunity to make forward progress with the data that has been read.

This very basic behavior can be effectively harnessed in the context of a backdoor in a fairly covert fashion. One interesting approach involves a user-mode process hosting a named pipe server and a blob of kernel-mode code reading data from the server and then executing it in the kernel-mode context. This general behavior would make it possible to run additional code in the kernel-mode context by simply shuttling it across a named pipe. The specifics of how this can be made to work are almost as simple as the steps described in the previous paragraph.

The user-mode part is simple; create a named pipe

server using `CreateNamedPipe` and then wait for a connection. The kernel-mode part is more interesting. One basic idea might involve having a kernel-mode routine that builds an asynchronous read IRP where the IRP's completion routine is defined as the kernel-mode routine itself. In this way, when data arrives from the user-mode process, the routine is notified and given an opportunity to execute the code that was supplied. After the code has been executed, it can simply re-use the code that was needed to pass the IRP to the underlying device associated with the named pipe that it's interacting with. The following pseudo-code illustrates how this could be accomplished:

```
KernelRoutine(DeviceObject, ReadIrp, Context)
{
  // First time called, ReadIrp == NULL
  if (ReadIrp == NULL)
  {
    FileObject = OpenNamedPipe(...)
  }
  // Otherwise, called during IRP completion
  else
  {
    FileObject = GetFileObjectFromIrp(ReadIrp)

    RunCodeFromIrpBuffer(ReadIrp)
  }
  DeviceObject = IoGetRelatedDeviceObject(FileObject)
  ReadIrp = IoBuildAsynchronousFsdRequest(...)
  IoSetCompletionRoutine(ReadIrp, KernelRoutine)
  IoCallDriver(DeviceObject, ReadIrp)
}
```

Category: Type II

Origin: The authors believe this to be the first public description of this technique.

Capabilities: Kernel-mode code execution.

Covertness: The authors believe this technique to be fairly covert due to the fact that the kernel-mode code profile is extremely minimal. The only code that must be present at all times is the code needed to execute the read buffer and then post the next read IRP to the target device object. There are two main strategies that might be taken to detect this technique. The first could include identifying malicious instances of the target device, such as a malicious named pipe server. The second might involve attempting to perform an in-memory fingerprint of the completion routine code, though this would be far from fool proof, especially if the kernel-mode code is encoded until invoked.

2.7 Leaking CS

With the introduction of protected mode into the x86 architecture, the concept of separate privilege levels, or rings, was born. Lesser privileged rings (such as ring 3) were designed to be restricted from accessing resources associated with more privileged rings (such as ring 0). To support this concept, segment descriptors are able to define access restrictions based on which rings should be allowed to access a given region of memory. The processor derives the *Current Privilege Level* (CPL) by looking at the low order two bits of the `CS` segment selector when it is loaded. If all bits are cleared, the processor is running at ring 0, the most privileged ring. If all bits are set, then processor is running at ring 3, the least privileged ring.

When certain events occur that require the operating system's kernel to take control, such as an interrupt, the processor automatically transitions from whatever ring it is currently executing at to ring 0 so that the request may be serviced by the kernel. As part of this transition, the processor saves the value of the a number of different registers, including the previous value of `CS`, to the stack in order to make it possible to pick up execution where it left off after the request has been serviced. The following structure describes the order in which these registers are saved on the stack:

```
typedef struct _SAVED_STATE
{
    ULONG_PTR Eip;
    ULONG_PTR CodeSelector;
    ULONG     Eflags;
    ULONG_PTR Esp;
    ULONG_PTR StackSelector;
} SAVED_STATE, *PSAVED_STATE
```

Potential security implications may arise if there is a condition where some code can alter the saved execution state in such a way that the saved `CS` is modified from a lesser privileged `CS` to a more privileged `CS` by clearing the low order bits. When the saved execution state is used to restore the active processor state, such as through an `iret`, the original caller immediately obtains ring 0 privileges.

Category: Undefined; this approach does not fit into any of the defined categories as it simply takes advantage of hardware behavior relating around how `CS` is used to determine the CPL of a processor. If code patching is used to be able to modify the saved `CS`, then the implementation is Type I.

Origin: Leaking `CS` to user-mode has been known to be dangerous since the introduction of protected mode (and thus rings) into the x86 architecture with the 80286 in 1982[22]. This approach therefore falls into the category of obvious due to the documented hardware implications of leaking a kernel-mode `CS` when transitioning back to user-mode.

Capabilities: Kernel-mode code execution.

Considerations: Leaking the kernel-mode `CS` to user-mode may have undesired consequences. Whatever code is to be called in user-mode must take into account that it will be running in a kernel-mode context. Furthermore, the kernel attempts to be as rigorous as possible about checking to ensure that a thread executing in user-mode is not allowed a kernel-mode `CS`.

Covertness: Depending on the method used to intercept and alter the saved execution state, this method has the potential to be fairly covert. If the method involves secondary hooking in order to modify the state, then it may be detected through some of the same techniques as described in the section on image patching.

3 Prevention & Mitigation

The primary purpose of this paper is not to explicitly identify approaches that could be taken to prevent or mitigate the different types of attacks described herein. However, it is worth taking some time to describe the virtues of certain approaches that could be extremely beneficial if one were to attempt to do so. The subject of preventing backdoors from being installed and persisted is discussed in more detail in section 4 and therefore won't be considered in this section.

One of the more interesting ideas that could be applied to prevent a number of different types of backdoors would be immutable memory. Memory is immutable when it is not allowed to be modified. There are a few key regions of memory used by the Windows kernel that would benefit greatly from immutable memory, such as executable code segments and regions that are effectively write-once, such as the SSDT. While immutable memory way work in principle, there is currently no x86 or x64 hardware (that the authors are aware of) that permits this level of control.

Even though there appears to be no hardware support for this, it is still possible to implement immutable memory in a virtualized environment. This is especially true in hardware-assisted virtualization implementations that make use of a hypervisor in some form. In this model, a hypervisor can easily expose a hypercall (similar to a system call, but traps into the hypervisor) that would allow an enlightened guest to mark a set of pages as being immutable. From that point forward, the hypervisor would restrict all writes to the pages associated with the immutable region.

As mentioned previously, particularly good candidates for immutable memory are things like the SSDT, Window's `ALMOSTRO` write-once segment, as well as other single-modification data elements that exist within the kernel. Enforcing immutable memory on these regions would effectively prevent backdoors from being able to establish certain types of

hooks. The downside to it would be that the kernel would lose the ability to hot-patch itself[9]. Still, the security upside would seem to out-weigh the potential downside. On x64, the use of immutable memory would improve the resilience of PatchGuard by allowing it to actively prevent hot-patching rather than relying on detecting it with the use of a polling cycle.

4 Running Code in Kernel-Mode

There are many who might argue that it's not even necessary to write code that prevents or detects specific types of kernel-mode backdoors. This argument can be made on the grounds of two very specific points. The first point is that in order for one to backdoor the kernel, one must have some way of executing code in kernel-mode. Based on this line of reasoning, one might argue that the focus should instead be given to preventing untrusted code from running in kernel-mode. The second point in this argument is that in order for one to truly compromise the host, some form of data must be persisted. If this is assumed to be the case, then an obvious solution would be to identify ways of preventing or detecting the persistent data. While there may also be additional points, these two represent the common themes observed by the authors. Unfortunately, the fact is that both of these points are, at the time of this writing, flawed.

It is currently not possible with present day operating systems and x86/x64 hardware to guarantee that only specific code will run in the context of an operating system's kernel. Though Microsoft wishes it were possible, which is clearly illustrated by their efforts in Code Integrity and Trusted Boot, there is no real way to guarantee that kernel-mode code cannot be exploited in a manner that might lead to code execution[2]. There have been no shortage of Windows kernel-mode vulnerabilities to illustrate the feasibility of this type of vector[6, 10]. This matter is also not helped by the fact that the Windows kernel currently has very few exploit mitigations. This makes the exploitation of kernel vulnerabilities trivial in comparison to some of the mitigations found in user-mode on Windows XP SP2 and, more recently, Windows Vista.

In addition to the exploitation vector, it is also important to consider alternative ways of executing code in kernel-mode that would be largely invisible to the kernel itself. John Heasman has provided some excellent research into the subject of using the BIOS, expansion ROMs, and the Extensible Firmware Interface (EFI) as a means of running arbitrary code in the context of the kernel without necessarily relying on any hooks directly visible to the kernel itself[16, 17]. Loc Duflot described how to use the System Management Mode (SMM) of Intel processors as a method of subverting the operating system to bypass BSD's `securelevel` restrictions[9]. There has also been a lot discussion around using DMA to directly interact with and modify physical memory without involving the operating system. However, this form of attack is of less concern due to the fact that physical access is required.

The idea of detecting or preventing a rootkit from persisting data is something that is worthy of thoughtful consideration. Indeed, it's true that in order for malware to survive across reboots, it must persist itself in some form or another. By preventing or detecting this persisted data, it would be possible to effectively prevent any form of sustained infection. On the surface, this idea is seemingly both simple and elegant, but the devil is in the details. The fact that this idea is fundamentally flawed can be plainly illustrated using the current state of Anti-Virus technology.

For the sake of argument, assume for the moment that there really is a way to deterministically prevent malware from persisting itself in any form. Now, consider a scenario where a web server at financial institution is compromised and a memory resident rootkit is used. The point here should be obvious: no data associated with the rootkit touches the physical hard-

[9] There are some instances where kernel-mode hot-patching is currently require, especially on x64

ware. In this example, one might rightly think that the web server will not be rebooted for an extended period of time. In these circumstances, there is really no difference between a persistent and non-persistent rootkit. Indeed, a memory resident rootkit may not be ideal in certain situations, but it's important to understand the implications.

Based on the current state-of-the-art, it is not possible to deterministically prevent malware from persisting itself. There are far too many methods of persisting data. This is further illustrated by John Heasman in his ACPI and expansion ROM work. To the authors' knowledge, modern tools focus their forensic analysis on the operating system and on file systems. This isn't sufficient, however, as rootkit data can be stored in locations that are largely invisible to the operating system. While this may be true, there has been a significant push in recent years to provide the hardware necessary to implement a trusted system boot. This initiative is being driven by the Trusted Computing Group with involvement from companies such as Microsoft and Intel[42]. One of the major outcomes of this group has been the Trusted Platform Module (TPM) which strives to facilitate a trusted system boot, among other things[43]. At the time of this writing, the effectiveness of TPM is largely unknown, but it is expected that it will be a powerful and useful security feature as it matures.

The fact that there is really no way of preventing untrusted code from running in kernel-mode in combination with the fact that there is really no way to universally prevent untrusted code from persisting itself helps to illustrate the need for thoughtful consideration of ways to both prevent and detect kernel-mode backdoors.

5 PatchGuard versus Rootkits

There has been some confusion centering around whether or not PatchGuard can be viewed as a deterrent to rootkits. On the surface, it would appear that PatchGuard does indeed represent a formidable opponent to rootkit developers given the fact that it checks for many different types of hooks. Beneath the surface, it's clear that PatchGuard is fundamentally flawed with respect to its use as a rootkit deterrent. This flaw centers around the fact that PatchGuard, in its current implementation, runs at the same privilege level as other driver code. This opens PatchGuard up to attacks that are designed to prevent it from completing its checks. The authors have previously outlined many different approaches that can be used to disable PatchGuard[36, 37]. It is certainly possible that Microsoft could implement fixes for these attacks, and indeed they have implemented some in more recent versions, but the problem remains a cat-and-mouse game. In this particular cat-and-mouse game, rootkit authors will always have an advantage both in terms of time and in terms of vantage point.

In the future, PatchGuard can be improved to leverage features of a hypervisor in a virtualized environment that might allow it to be protected from malicious code running in the context of a guest. For example, the current version of PatchGuard currently makes extensive use of obfuscation in order to presumably prevent malware from finding its code and context structures in memory. The presence of a hypervisor may permit PatchGuard to make more extensive use of immutable memory, or to alternatively run at a privilege level that is greater than that of an executing guest, such as within the hypervisor itself (though this could have severe security implications if done improperly).

Even if PatchGuard is improved to the point where it's no longer possible to disable its security checks, there will still be another fundamental flaw. This second flaw centers around the fact that PatchGuard, like any other code designed to perform explicit checks, is like a horse with blinders on. It's only able to detect modifications to the specific structures that it knows about. While it may be true that these structures are the most likely candidates to be hooked, it is nevertheless true that many other structures exist that would make suitable candidates, such as the `SuspendApc` of a specific thread. These alternative candidates are meant to illustrate the

challenges PatchGuard faces with regard to continually evolving its checks to keep up with rootkit authors. In this manner, PatchGuard will continue to be forced into a reactive mode rather than a proactive mode. If IDS products have illustrated one thing it's that reactive security solutions are largely inadequate in the face of a skilled attacker.

PatchGuard is most likely best regarded as a hall monitor. Its job is to make sure students are doing things according to the rules. Good students, such as ISVs, will inherently bend to the will of PatchGuard lest they find themselves in unsupported waters. Bad students, such as rootkits, fear not the wrath of PatchGuard and will have few qualms about sidestepping it, even if the technique used to sidestep may not work in the future.

6 Acknowledgements

The authors would like to acknowledge all of the people, named or unnamed, whose prior research contributed to the content included in this paper.

7 Conclusion

At this point it should be clear that there is no shortage of techniques that can be used to expose a local kernel-mode backdoor on Windows. These techniques provide a subtle way of weakening the security guarantees of the Windows kernel by exposing restricted resources to user-mode processes. These resources might include access to kernel-mode data, disabling of security checks, or the execution of arbitrary code in kernel-mode. There are many different reasons why these types of backdoors would be useful in the context of a rootkit.

The most obvious reason these techniques are useful in rootkits is for the very reason that they provide access to restricted resource. A less obvious reason for their usefulness is that they can be used as a method of reducing a rootkit's kernel-mode code profile. Since many tools are designed to scan kernel-mode memory for the presence of backdoors[32, 14], any reduction of a rootkit's kernel-mode code profile can be useful. Rather than placing code in kernel-mode, techniques have been described for redirecting code execution to code stored in user-mode in a process-specific fashion. This is accomplished by redirecting code into a portion of the `ntdll` mapping which exists in every process, including the `System` process.

Understanding how different backdoor techniques work is necessary in order to consider approaches that might be taken to prevent or detect rootkits that employ them. For example, the presence of immutable memory may eliminate some of the common techniques used by many different types of rootkits. Likewise, when these techniques are eliminated, new ones will be developed, continuing the cycle that permeates most adversarial systems.

References

[1] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Dec, 2005.

[2] Anonymous Hacker. *Xbox 360 Hypervisor Privilege Escalation Vulnerability*. Bugtraq. Feb, 2007. http://www.securityfocus.com/archive/1/461489

[3] Blanset, David et al. *Dual operating system computer*. Oct, 1985. http://www.freepatentsonline.com/4747040.html

[4] Brown, Ralf. *Pentium Model-Specific Registers and What They Reveal*. Oct, 1995. http://www.rcollins.org/articles/p5msr/PentiumMSRs.html

[5] Butler, James and Sherri Sparks. *Windows Rootkits of 2005*. Nov, 2005. http://www.securityfocus.com/infocus/1850

[6] Cerrudo, Cesar. *Microsoft Windows Kernel GDI Local Privilege Escalation*. Oct, 2004.

http://projects.info-pull.com/mokb/MOKB-06-11-2006.html

[7] CIAC. *E-34: One_half Virus (MS-DOS)*. Sep, 1994. http://www.ciac.org/ciac/bulletins/e-34.shtml

[8] Conover, Matt. *Malware Profiling and Rootkit Detection on Windows*. 2005. http://xcon.xfocus.org/xcon2005/archives/2005/Xcon2005_Shok.pdf

[9] Duflot, Loc. *Security Issues Related to Pentium System Management Mode*. CanSecWest, 2006. http://www.cansecwest.com/slides06/csw06-duflot.ppt

[10] Ellch, John et al. *Exploiting 802.11 Wireless Driver Vulnerabilities on Windows*. Jan, 2007. http://www.uninformed.org/?v=6&a=2&t=sumry

[11] Firew0rker, the nobodies. *Kernel-mode backdoors for Windows NT*. Phrack 62. Jan, 2005. http://www.phrack.org/issues.html?issue=62&id=6#article

[12] fuzen_op. *SysEnterHook*. Feb, 2005. http://www.rootkit.com/vault/fuzen_op/SysEnterHook.zip

[13] Garfinkel, Tal. *Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools*. http://www.stanford.edu/~talg/papers/traps/traps-ndss03.pdf

[14] Gassoway, Paul. *Discovery of kernel rootkits with memory scan*. Oct, 2005. http://www.freepatentsonline.com/20070078915.html

[15] Gulbrandsen, John. *System Call Optimization with the SYSENTER Instruction*. Oct, 2004. http://www.codeguru.com/Cpp/W-P/system/devicedriverdevelopment/article.php/c8223/

[16] Heasman, John. *Implementing and Detecting an ACPI BIOS Rootkit*. BlackHat Federal, 2006. https://www.blackhat.com/presentations/bh-federal-06/BH-Fed-06-Heasman.pdf

[17] Heasman, John. *Implementing and Detecting a PCI Rootkit*. Nov, 2006. http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf

[18] Hoglund, Greg. *Kernel Object Hooking Rootkits (KOH Rootkits)*. Jun, 2006. http://www.rootkit.com/newsread.php?newsid=501

[19] Hoglund, Greg. *A *REAL* NT Rootkit, patching the NT Kernel*. Phrack 55. Sep, 1999. http://phrack.org/issues.html?issue=55&id=5

[20] Hoglund, Greg and James Butler. *Rootkits: Subverting the Windows Kernel*. 2006. Addison-Wesley.

[21] Hunt, Galen and Doug Brubacher. *Detours: Binary Interception of Win32 Functions*. Proceedings of the 3rd USENIX Windows NT Symposium, pp. 135-143. Seattle, WA, July 1999. USENIX.

[22] Intel. *2.1.2 The Intel 286 Processor (1982)*. Intel 64 and IA-32 Architectures Software Developer's Manual. Denver, Colorado: Intel, 34. http://www.intel.com/products/processor/manuals/index.htm.

[23] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 3: System Programming Guide*. Sep, 2005.

[24] Jack, Barnaby. *Remote Windows Kernel Exploitation: Step into the Ring 0*. Aug, 2005. http://www.blackhat.com/presentations/bh-usa-05/BH_US_05-Jack_White_Paper.pdf

[25] Kasslin, Kimmo. *Kernel Malware: The Attack from Within*. 2006. http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf

[26] Kdm. *NTIllusion: A portable Win32 userland rootkit* [incomplete]. Phrack 62. Jan, 2005. http://www.phrack.org/issues.html?issue=62&id=12&mode=txt

[27] M. B. Jones. *Interposition agents: Transparently interposing user code at the system interface.* In Symposium on Operating System Principles, pages 80-93, 1993. http://www.scs.stanford.edu/nyu/04fa/sched/readings/interposition-agents.pdf

[28] Mythrandir. *Protected mode programming and O/S development.* Phrack 52. Jan, 1998. http://www.phrack.org/issues.html?issue=52&id=17#article

[29] PaX team. *PAGEEXEC.* Mar, 2003. http://pax.grsecurity.net/docs/pageexec.txt

[30] Plaguez. *Weakening the Linux Kernel.* Phrack 52. Jan, 1998. http://www.phrack.org/issues.html?issue=52&id=18#article

[31] Prasad Dabak, Milind Borate, and Sandeep Phadke. *Hooking Software Interrupts.* Oct, 1999. http://www.windowsitlibrary.com/Content/356/09/1.html

[32] Rutkowska, Joanna. *System Virginity Verifier.* http://invisiblethings.org/tools/svv/svv-2.3-src.zip

[33] Rutkowska, Joanna. *Rookit Hunting vs. Compromise Detection.* BlackHat Europe, 2006. http://invisiblethings.org/papers/rutkowska_bheurope2006.ppt

[34] Rutkowska, Joanna. *Introducing Stealth Malware Taxonomy.* Nov, 2006. http://invisiblethings.org/papers/malware-taxonomy.pdf

[35] Silvio. *Shared Library Call Redirection Via ELF PLT Infection.* Phrack 56. Jan, 2000. http://www.phrack.org/issues.html?issue=56&id=7#article

[36] skape and Skywing. *Bypassing PatchGuard on Windows x64.* Uninformed Journal. Jan, 2006. http://www.uninformed.org/?v=3&a=3&t=sumry

[37] Skywing. *Subverting PatchGuard version 2.* Uninformed Journal. Jan, 2007. http://www.uninformed.org/?v=6&a=1&t=sumry

[38] Skywing. *Anti-Virus Software Gone Wrong.* Uninformed Journal. Jun, 2006. http://www.uninformed.org/?v=4&a=4&t=sumry

[39] Skywing. *Programming against the x64 exception handling support.* Feb, 2007. http://www.nynaeve.net/?p=113

[40] Soeder, Derek. *Windows Expand-down Data Segment Local Privilege Escalation.* Apr, 2004. http://research.eeye.com/html/advisories/published/AD20040413D.html

[41] Sparks, Sherri and James Butler. *Raising the Bar for Windows Rootkit Detection.* Phrack 63. Jan, 2005. http://www.phrack.org/issues.html?issue=63&id=8

[42] Trusted Computing Group. *Trusted Computing Group: Home.* https://www.trustedcomputinggroup.org/home

[43] Trusted Computing Group. *TPM Specification.* https://www.trustedcomputinggroup.org/specs/TPM/

[44] Welinder, Morten. *modify_ldt security holes.* Mar, 1996. http://lkml.org/lkml/1996/3/6/13

[45] Wikipedia. *Call gate.* http://en.wikipedia.org/wiki/Call_gate