Kernel Malware: The Attack from Within

Kimmo Kasslin
F-Secure
Kuala Lumpur
Malaysia
kimmo.kasslin@f-secure.com

Abstract

The Kernel is the heart of modern operating systems. Code executing in kernel mode has full access to all memory including the kernel itself, all CPU instructions, and all hardware. For this obvious reason only the most trusted software should be allowed to run in kernel mode.

Today, we are facing an emerging threat in the form of kernel-mode malware. By kernel-mode malware we mean malicious software that executes as part of the operating system having full access to the computer's resources. To the end-user this means malware that can bypass software firewalls and can be almost impossible to detect or remove even if the best anti-virus solutions are being used.

This paper will examine the most important malware cases utilizing kernel-mode techniques over the last few years. The research will be limited to malware running on Windows NT and later operating system versions. It will look at the possible motives for the malware authors to move their creations to kernel mode. A detailed analysis of the key techniques making their existence possible will be covered.

Introduction

A modern malware, backdoor, email-worm, spying trojan, operating fully in kernel mode on a Windows NT-based operating system is a scary thought. It would operate with same privileges and share all the same resources as the operating system itself and compete with any security solutions protecting the systems integrity against any malicious activities. This would end up in an arms race between the malware and the security suite. The one that is able to execute first or being able to control the lowest parts of the operating system eventually will be the winner. Usually, this means doing tricks that are not documented or supported by Microsoft and will result in version dependencies causing instability or even system crashes in the worst case. This is a path any serious security software vendor will not take. However, the world is full of examples of malware and proof-of-concept code that does exactly this.

This paper is about malware operating fully or partially in kernel mode. It will discuss basic requirements for such malware and survey the known and semi-documented techniques that make it possible for malware writers to enter into and operate in ring 0. The paper ends with analyzing two sample cases of malware that have played an important role in opening eyes of the security community to the threat kernel malware poses.

Basic Definitions

To have a better understanding of the topic the reader should be familiar with some basic concepts of a modern operating system. Terms such as processes, threads, virtual memory and the difference between user and kernel modes should be familiar to the reader. The most important concepts, including important definitions to avoid misunderstandings are introduced here.

Kernel Mode vs. User Mode

One of the design goals for Windows NT was reliability and robustness [34]. The main requirement was to protect the kernel from external tampering by user-mode applications. To address this, the system was divided into two different modes of operation: user mode and kernel mode. User applications run in user mode. They are unprivileged processes with limited access to system resources. User-mode processes access these resources controlled by the kernel through services provided by the kernel.

Operating system services and third party drivers run in kernel mode. In kernel mode, they have access to all system memory, all CPU instructions, and all hardware. The architecture of the Intel x86 processor supports four privilege levels, or rings, numbered from 0 to 3. The greater numbers mean lesser privileges. Windows uses privilege level 0 for kernel mode and privilege level 3 for user mode.

To provide protection to the virtual memory, the OS keeps track from which privilege level each memory page can be accessed. Pages in system address space can be only accessed from kernel mode, which protects them from user-mode processes. Since code running in kernel mode has full access to system memory, it can easily bypass any security mechanism provided by the OS and destroy its integrity.

Drivers run in kernel mode and third party drivers can be installed and loaded with administrative privileges. This is the only documented and supported way of executing kernel-mode code. Other undocumented means do exist and they are explained in more detail in a later section

For detailed information about the inner workings of Windows NT-based operating systems the author recommends the excellent book by Solomon and Russinovich [34].

Kernel Malware

Kernel malware is malicious software that runs fully or partially at the most privileged execution level, ring 0, having full access to memory, all CPU instructions, and all hardware. It is convenient to divide kernel malware into two subcategories: full-kernel malware and semi-kernel malware.

Full-Kernel Malware

Full-Kernel malware is fully implemented as kernel-mode driver and executes all its code in ring 0. It still requires some help from user mode to get installed into the system either by a dropper component or manually by the user. Once the driver has been installed it will be able to operate by itself.

Semi-Kernel Malware

Semi-Kernel malware executes its code both in ring 0 and 3. This includes malware that consists of a user-mode executable (EXE or DLL) that will drop a kernel-mode driver or use other means to execute code in ring 0. Malware that is fully implemented as kernel-mode driver but executes parts of its payload in ring 3, e.g. through code injection, in the context of some process is also semi-kernel malware.

History and Trends of Kernel Malware

Kernel-mode malware on Windows NT-based systems is not a new phenomenon, they have just been rare. WinNT/Infis [24], which was discovered in November 1999, was the first known full-kernel malware that was designed to run on NT-based systems. It was a memory-resident parasitic kernel-mode driver virus that gained control by hooking the INT 0x2E interrupt handler directly from kernel. This allowed it to monitor every system service call made by user-mode applications and to infect PE EXE files when an open request was made. Win2K/Infis.4608 [35] added support for Windows 2000 and was found just one week after the new operating system was released.

Another documented case involving kernel-mode malware was Virus.Win32.Chatter[1] [2][36], which was found in January 2003. It was a kernel-mode driver that infects only PE SYS files. It hooks nt!NtCreateFile from nt!KiServiceTable and thus gets control on every file open and create operation. However, its infection routine was actually executed in user mode. Therefore it is a semi-kernel malware. The driver copied itself into the address space of the active process and used an undocumented nt!KeUserModeCallBack function exported by the kernel to execute the infection routine in user mode. This might have been the first time for kernel malware to inject parts of its payload from ring 0 to ring 3 to perform some more complex tasks. This code injection from kernel to user mode is an important concept and will be discussed in more detail later in this paper.

Today the number of kernel-mode malware when compared to the total number of malware seen every month is very small. Also, it should be noted that the antivirus industry has not yet seen any complex malware that would fulfill the requirements for full-kernel malware. To get a better view of how real the threat posed by kernel malware is, it is important to find some evidence of real-life malware samples using kernel-mode components.

Antivirus analysts who have been analyzing malware samples since the beginning of 2005 should agree that the number of malware using kernel components has been steadily increasing. To get more proof of this trend a statistical analysis was conducted. The author chose two antivirus vendors and processed all monthly sample collections from January 2003 to August 2006. On average this resulted in around 100000 samples per vendor. The idea was to find out how many kernel-mode drivers the sample collections had, meaning that they are either full-kernel malware or parts of semi-kernel malware. In addition, the number of new malware families that use kernel-mode drivers was identified. It is important to notice that just looking for drivers will not include kernel malware that use other means, like code injection or call gates, to execute their code in ring 0. However, this would require run-time analysis of the samples, which was not possible to achieve within the given time frame.

Each sample in the monthly collection was first checked whether it had a proper PE header. If the result was positive then additional tests were made against the optional header field. Following basic checks were made to include only samples that are possibly kernel-mode drivers for Windows NT and later operating system versions:

- `Magic` field equals `IMAGE_NT_OPTIONAL_HDR32_MAGIC`
- `Subsystem` field equals `IMAGE_SUBSYSTEM_NATIVE`
- `MajorSubsystemVersion` and `MinorSubsystemVersion` fields were checked against the correct platform and version information

In addition all duplicate files were removed by checking their MD5 hash. The results are shown in *Figure 1* below.

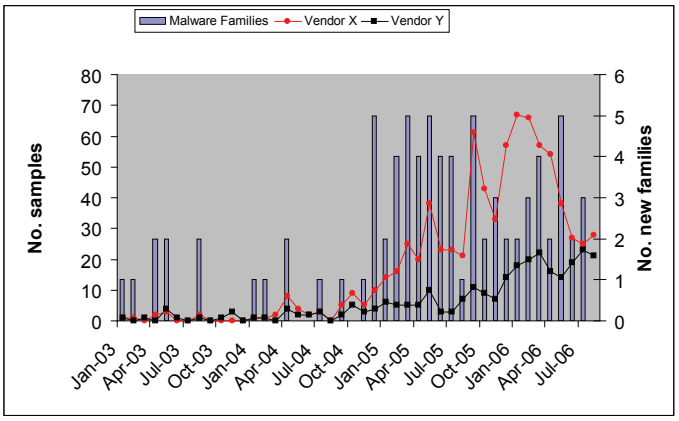

Figure 1. Number of malicious kernel-mode driver samples found from each vendor collection per month. Also, includes the number of new malware families found to use kernel-mode components.

Before discussing more about the results it should be noted that there exists possible factors of uncertainty towards the validity of the results. This is mostly related to the quality of the data these results are based on. The author is not trying to imply that there is anything wrong with the collections, it is just a known problem in the antivirus industry that proper classification and naming

1. Also know as W32.Keck.1933 (Symantec) and W32/Chatter (McAfee).

of malware is impossible with the amount of samples the industry have to handle every day.

Also, normally drivers are embedded inside the main malware component and will be dropped to the system when the main component is executed. However, quite often the dropped driver is named by its characteristics, e.g. Hacktool.Rootkit[2], not by the malware using it. From the end-user's point-of-view this is a good thing but it makes it very difficult for researchers to make any conclusions how many different families are using kernel-mode components without actually executing every sample and checking what kind of components they are dropping to the system.

Despite all the uncertainty the author feels confident to make a conclusion based on the achieved results that number of kernel-mode malware has steadily increased during the investigated time period. This is very evident from year 2005 onwards, which is mostly explained by the increased number of malware starting to use kernel-mode rootkits to hide their presence on the compromised system. There is a noticeable difference in the number of samples from 2005 onwards between the two vendors. In vendor Y's case the curve is more stable where the amount of driver samples has approximately tripled every year. The raw number of new samples per month is not that interesting since these samples contain lots of variants of the same family. Probably more interesting information is the number of new malware families that were identified to be using kernel-mode drivers. This gives a better picture of the kernel malware trend.

During year 2003 an average of 0.67 new malware families per month were identified to utilize kernel-mode drivers. Year 2004 was still quite calm when an average of one new family per month was identified. The trend changed dramatically during year 2005 when an average of 3.42 new kernel malware families was found. Since then things have calmed down a bit but the usage of kernel malware is still growing strongly with an average of 2.63 new families found each month. *Table 1* contains the list of most active families based on the number of new variants seen.

Table 1. Most commonly used kernel-mode malware.

F-Secure	Symantec	McAfee
Backdoor.Win32.Haxdoor	Backdoor.Haxdoor	Backdoor-BAC
Backdoor.Win32.HacDef	Backdoor.HackDefender	HackerDefender trojan
Trojan-Spy.Win32.Banker	Infostealer.Bancos	PWS-Banker trojan
Backdoor.Win32.PcClient	Backdoor.Formador	BackDoor-CKB trojan
Trojan-Spy.Win32.Goldun	Trojan.Goldun	PWS-Goldun trojan
Trojan.Win32.Crypt.t	Spyware.Apropos.C	Adware-Apropos
SpamTool.Win32.Mailbot	Backdoor.Rustock.A	Spam-Mailbot trojan
Trojan-Clicker.Win32.Costrat	Backdoor.Rustock.B	Spam-Mailbot.c trojan

Other important families worth to mention that have been seen to utilize kernel-mode code are Email-Worm.Win32.Bagle and Mydoom-based Email-Worm.Win32.Gurong [10]. In addition usage of kernel-mode rootkit drivers has been very common in IRC bots such as Backdoor.Win32.SdBot and Backdoor.Win32.Rbot.

The high rise in popularity of kernel malware can be mostly explained by the increased motivation for malware authors to hide their creations from detection as long as possible. To hide even better they have started to use kernel-mode rootkit techniques as more and more documentation, examples and fully working examples with full source code has become publicly available. However, there are other motives for malware to move to kernel, probably most important ones being firewall and antivirus scanner bypassing.

Key Techniques Used by Kernel Malware

One reason why kernel malware has been so rare is that developing kernel-mode drivers is not easy. The environment limits the developer's creativity since it offers only a limited number of exported library functions, documentation is limited and there is less examples and source code available that can be used as a template for the malicious work. Today the situation is changing. More information is published about how to do things required by today's malware directly from kernel mode. This includes how to implement better rootkits, how to bypass personal firewalls and how to create backdoors and IRC bots.

To better understand the threat of kernel malware it is important to know how they work. This chapter tries to give a brief introduction on the key techniques that are used by kernel malware.

Executing Code in Ring 0

The first requirement for a malware trying to obtain the powers of kernel malware is to execute its code in the most privileged level – ring 0. The author has seen malware using two different techniques to achieve this, kernel-mode drivers and call gates.

Kernel-Mode Drivers

The only documented way to execute third party kernel-mode code is to install a kernel-mode driver. They are loadable kernel-mode modules, usually having extension .sys, and execute in one of three contexts:

- In the context of user-mode thread that initiated an

2. This is a generic detection name used by Symantec for rootkits.

I/O handler function

- In the context of kernel-mode system thread
- In the context of random thread as a result of an interrupt

Structurally they are identical to any other PE file. Kernel-mode drivers can use support routines that are exported by various components of the OS kernel. These routines support I/O, configuration, Plug and Play, power management, memory management, and numerous other OS features. [34]

Typically, drivers are loaded and started at OS boot time. However, Windows API provides necessary functions that allow loading and unloading drivers at run time. This is done by the Service Control Manager (SCM). A new driver can be installed with the CreateService API function. This function has the following syntax [37]:

```
SC_HANDLE
CreateService(
    SC_HANDLE hSCManager,
    LPCTSTR lpServiceName,
    LPCTSTR lpDisplayName,
    DWORD dwDesiredAccess,
    DWORD dwServiceType,
    DWORD dwStartType,
    DWORD dwErrorControl,
    LPCTSTR lpBinaryPathName,
    LPCTSTR lpLoadOrderGroup,
    LPDWORD lpdwTagId,
    LPCTSTR lpDependencies,
    LPCTSTR lpServiceStartName,
    LPCTSTR lpPassword
);
```

Here, the important parameters are `hSCManager`, `dwServiceType`, `dwStartType`, and `lpBinaryPathName`. The `hSCManager` parameter is a handle to the SCM database, which can be obtained with the `OpenSCManager` API function. The `dwServiceType` parameter specifies the type of the service that will be installed. In this case, it must have the value of `SERVICE_KERNEL_DRIVER`. The `dwStartType` parameter defines the startup behavior of the driver when the system boots up. The final parameter of interest is the `lpBinaryPathName`, which is a pointer to a null-terminated string that contains the fully qualified path to the driver binary file. [37]

A successful call to the `CreateService` API function creates a service object, returns a handle to this object, and installs the service in the SCM database by creating a key with the same name as the service under the following registry key:

`HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services`

The driver can be started by passing the service object handle to the `StartService` API function. This causes the system to perform some actions that are similar to loading a normal user-mode DLL. An image of the driver's PE file is loaded into system address space and the entry point of the driver is called. The entry point is also known as the `DriverEntry`, which has following prototype [22]:

```
NTSTATUS
DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath
);
```

`DriverEntry` routine always runs in the context of the system thread. This routine is called only once during the lifetime of the driver and its only purpose is to initialize the driver. However, this is all that is required for the malware to execute its payload, which quite often involves hooking kernel functions or installing additional notification functions or system threads to perform any further work. [22]

There exists another way to load and execute a kernel-mode driver from user mode.

This method is not documented by Microsoft in any way. The existence of this method was announced by Hoglund [38] in his post to the BugTraq mailing list.

More recent information can be found in the book written by Hoglund and Butler [15]. This method uses the `ZwSetSystemInformation` function exported by `Ntdll.dll` and has the following syntax [23]:

```
NTSYSAPI
NTSTATUS
NTAPI
ZwSetSystemInformation(
    IN SYSTEM_INFORMATION_CLASS
          SystemInformationClass,
    IN OUT PVOID SystemInformation,
    IN ULONG SystemInformationLength
);
```

The `SystemInformationClass` parameter specifies the type of system information to be set. In this case it is set to value of `SystemLoadAndCallImage`, which is part of the `SYSTEM_INFORMATION_CLASS` enumeration. The `SystemInformation` parameter points to the data will be set. In this case, it will be

defined as a structure that has the following syntax [23]:

```
typedef struct _SYSTEM_LOAD_AND_CALL_
IMAGE {
UNICODE_STRING ModuleName;
} SYSTEM_LOAD_AND_CALL_IMAGE, *PSYS-
TEM_LOAD_AND_CALL_IMAGE;
```

The `ModuleName` element specifies the full path of the module that will be loaded in Unicode format. The last parameter of the `ntdll!ZwSetSystemInformation` function specifies the size in bytes of the data pointed by the `SystemInformation` parameter. After a successful function call, the system loads the module into the system address space and calls its entry point. This technique has both good and bad features.

The upside for malware is that it leaves no information into the SCM database. This method provides malware a way to load and execute a kernel-mode driver in a single operation without leaving any traces of its existence in the registry. Because the SCM is not aware of the driver, it is not able to control it in any way. The driver cannot be unloaded and it will survive in memory until the next reboot.

The downside for malware is that the driver will not be started after the system has been rebooted. Another problem is that memory for the driver image will be allocated from the paged pool, which restricts the kind of operations the driver can execute. Otherwise it risks of crashing the system with a blue screen. There are ways to avoid this problem but it will make the driver code more complex and harder to maintain [15].

As expected, the documented way is the one used by most malware to execute code in kernel mode. If it works, there is no reason for malware to search for alternate means. This might change in the future when more security products start to block loading of kernel-mode drivers.

Call Gates

Call gates are a less known technique to execute third party kernel-mode code. They are provided by the Intel architecture and can be used to transfer program control between different privilege levels. Call gates reside in the system address space. A user-mode process has to be able to modify it to add its own call gate. This is possible through two undocumented Windows OS features that are discussed next.

The first method was first discovered by Mark Russinovich [32] and it was further documented by Crazylord [3]. It uses a section object named `\Device\PhysicalMemory`, which allows a user-mode application to map portions of the physical memory into its own address space. This technique is known to malware and has been used before by some malware variants to hide themselves using rootkit techniques [21]. This technique has its problems since it accesses physical memory. The malware needs to calculate virtual to physical address translation correctly. Email-Worm.Win32.Fanbot.j[3] was the first one that did the translation properly and therefore had better support for different OS versions [4].

The second method to access the system address space from user mode is more useful because it uses virtual addresses instead of physical addresses. This method uses the `ZwSystemDebugControl` function exported by `Ntdll.dll` and has the following syntax [23]:

```
NTSYSAPI
NTSTATUS
NTAPI
ZwSystemDebugControl(
    IN DEBUG_CONTROL_CODE ControlCode,
    IN PVOID InputBuffer OPTIONAL,
    IN ULONG InputBufferLength,
    OUT PVOID OutputBuffer OPTIONAL,
    IN ULONG OutputBufferLength,
    OUT PULONG ReturnLength OPTIONAL
);
```

To use this function the caller is required to have the `SeDebugPrivilege` enabled. The important field here is the `ControlCode`, which defines the mode of operation. Valid values are defined in the `DEBUG_CONTROL_CODE` enumeration. However, this enumeration has not been documented since Windows 2000.

Randnut's [26] post to the Bugtraq mailing list showed that new features were added to the `ntdll!ZwSystemDebugControl` function on Windows XP and 2003. The essential information was posted to a public discussion forum by a person known as Bilbo. He claimed that it was possible to write to the system address space by setting the `ControlCode` parameter to the value of nine. He also posted the following example code:

```
/*
* write a buffer to kernel space
*/
struct {
  LPVOID kernel_addr;
  LPVOID user_addr;
DWORD len;
} mem;

void
wr(LPVOID dst, LPVOID src, DWORD len)
```

3. Also known as W32.Fanbot.A@mm (Symanted) and W32/Mytob.gen@MM (McAfee).

```
{
  mem.kernel _ addr = dst;
  mem.user _ addr = src;
  mem.len = len;
ZwSystemDebugControl(9, &mem,
   sizeof(mem), 0, 0, 0);
}
```

This made it possible for easy access to kernel-mode memory without any need to for complex address translation calculations. Lately, `ntdll!ZwSystemDebugControl` function was documented in much detail by Alex Ionescu [20]. This might result in more malware using this approach to enter the kernel. So far the author has not encountered any malware samples that had used this technique.

Microsoft has realized the potential problem these undocumented backdoors to the system address space pose to the overall security of their operating systems. Since Windows Server 2003 SP1 they can no longer be called from user mode [20].

Now that we know the techniques to install the call gate from ring 3 it is time to see how they actually work. Call gates are based on the same techniques the OS uses when it executes kernel-mode code on behalf of the user-mode application. At the lowest level, this separation of privilege levels is implemented by the protection mechanism provided by the Intel processors, also known as protected mode of operation. The reader is assumed to be familiar with concepts such as memory segmentation and privilege levels. For more information, the reader is advised to refer to the Intel manuals [16][17][18][19].

To provide a controlled way to transfer program control between different privilege levels the Intel architecture provides a facility called call gates. A call gate is defined by a call-gate descriptor, which may reside in the Global Descriptor Table (GDT) or in a Local Descriptor Table (LDT). *Figure 2* shows the format of a call-gate descriptor.

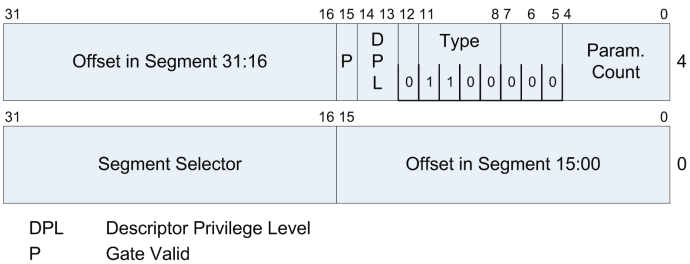

Figure 2. Call-gate descriptor Source: [19]

The `Segment Selector` field in a call gate specifies the code segment to be accessed. The `Offset` field specifies the entry point in the code segment. Generally, it points to the first instruction of a specific procedure. The `Descriptor Privilege Level` (DPL) field indicates the privilege level required to access the selected procedure through the gate. The `Parameter Count` field indicates the number of parameters to copy from the calling procedures stack to the new stack if a stack switch occurs.

A call gate is used by specifying a far pointer to the gate as the target operand in a CALL or JMP instruction. The pointer consists of a segment selector and an offset. The segment selector identifies the call gate and the offset is required, but not used. When the processor has accessed the call gate and the privilege checks has been successfully passed, it uses the segment selector from the call gate to locate the segment descriptor for the destination code segment. It then combines the base address from the code-segment descriptor with the offset from the call gate to form the linear address of the procedure entry point in the code segment. *Figure 3-2* illustrates this process.

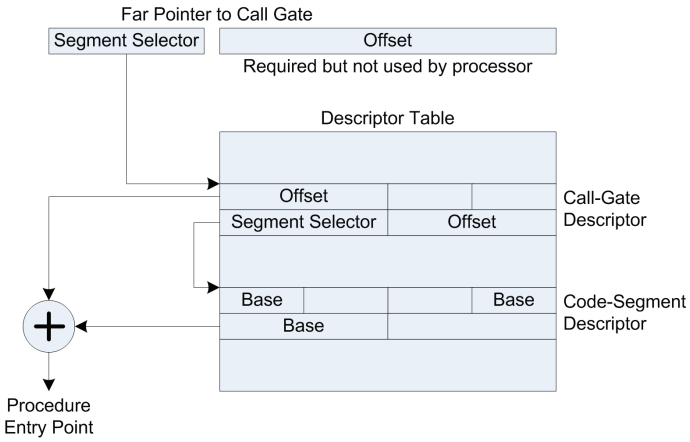

Figure 3. Call-gate mechanism Source: [19]

To execute code residing in user address space at ring 0, all that has to be done is to add a new call gate to the GDT. The address and size of the GDT can be obtained with the SGDT instruction. The GDT is located at the system address space, so either physical memory device or ntdll! `ZwSystemDebugControl` technique can be used to write the new call-gate descriptor to the GDT. The `Offset` field of the new call-gate descriptor will point to the payload code. The `Segment Selector` field should be set to a code segment that executes at ring 0. The final thing to remember is to grant access to the call gate from user mode by setting the DPL field to value 0x3.

Call gates are not a common technique used by today's malware. The author has only encountered one case where a malware used this technique to access ring 0. This malware is detected as Email-Worm.Win32.Gurong.a[4] [10][6]. Gurong.a installed a call gate through the physical

4. Also detected as W32.Mydoom!gen (Symantec) and W32/Mytob.gen@MM (McAfee).

memory device. The ring 0 code was used to hide its process, file, and launch point from the registry. It achieved this by hooking functions from `nt!KiServiceTable` and by removing its process object from internal kernel lists.

Using Kernel-Mode Support Routines

To fulfill the requirements for full-kernel malware the malware has to execute its entire payload in ring 0. The difficulty level of implementing such malware can vary from hard to impossible depending on the set of features it has to support. In case of a basic downloader, it probably needs to perform following operations:

- It allocates memory for storing temporary data
- It accesses internet to download the new payload
- It stores the file on the file system
- It modifies the registry to make sure the new payload will be executed

The Windows kernel provides a set of documented support routines that can be used by kernel-mode drivers to perform various tasks. These are documented in the Windows Driver Kit (WDK) [22], which was previously known as the Windows DDK. In addition WDK contains many source code examples how different tasks can be performed. From the above list all tasks except accessing internet are very simple to implement from kernel mode. Following functions would do the job:

- ExAllocatePoolWithTag / ExFreePoolWithTag
- ZwCreateFile / ZwWriteFile / ZwClose
- ZwCreateKey / ZwSetValueKey / ZwClose

If the reader is familiar with Native API, then Zw* functions should look familiar. An excellent reference with function prototypes is the Native API Reference book by Nebbett [19]. The kernel exports only a subset of the functions that are exported by `Ntdll.dll`. On Windows XP SP2 this subset consists of approximately 130 functions out of 280. In case of unexported functions there are ways to find the correct address to call. The best of them, and often used by malware, is to get this information from `Ntdll.dll` [27]. Every Zw* function in `Ntdll.dll` is implemented in similar way. Below is the disassembled version of `ntdll!ZwWriteVirtualMemory`:

```
0x7C90EA32:  B815010000  MOV   EAX,0x115
0x7C90EA37:  BA0003FE7F  MOV   EDX,0x7FFE0300
0x7C90EA3C:  FF12        CALL  DWORD PTR [EDX]
0x7C90EA3E:  C21400      RET   0x14
```

The first instruction, `MOV EAX, 0x115`, tells the System Service Dispatcher (SSD) that it should call function pointer from `nt!KiServiceTable` located at index 0x115. This function pointer is the address of the real service function that will perform the requested task. In this case it would be `nt!NtWriteVirtualMemory`.

The basic idea is to load `Ntdll.dll` image from disk into memory and then to find the entry point of the required function from the PE header's export table. Following simple piece of code fetches the index which can then be used to get the correct function pointer from the service table.

```
ulSSN = *(PULONG)((PUCHAR)pbNativeAPIFunction + 1);
```

This allows kernel-mode code to have the full Native API in its arsenal. However, there are still limitations what can be done and when. This is mostly caused by the fact that kernel-mode code can execute in one of three contexts – user thread, system thread or random. If the code executes in user thread context and the thread's `PreviousMode` field equals `UserMode`, then extra argument validation is done for the passed parameters. This means that any passed pointers must reside in user address space or otherwise the operation will fail. The easiest way to solve this is to make sure the routines are always called from system thread context.

One topic that has not yet been discussed is how internet can be accesses from ring 0. This is an important feature for the simple downloader since otherwise it will not be able to fetch the new payload. This is currently the hot topic in the field of kernel malware. It is worrying to see how the number of working solutions with full source code is published by various authors [28][29][30][15] that will eventually make it easier for more malware to enter the kernel. This topic is worth another research paper and will not be discussed any further.

Executing Code in Ring 3

In an ideal world kernel malware should do all its tasks from ring 0. However, this is not always feasible since it would require too much effort to implement some libraries that are only available from user mode. In addition, there are situations where it is beneficial for the malware to execute at high enough level. A good example of such malware is a trojan that steal user's banking credentials. Normally, communication between the client and server is encrypted and encryption is done at user mode. If the malware intercepted the data from kernel mode, it would

already be in encrypted format. In this case a better approach would be to execute parts of the code in ring 3 in the context of the process where the credentials are stored before encryption.

Malware that initially executes at ring 0 but later executes parts of its payload in ring 3 has been very rare to the author's knowledge. Virus.Win32.Chatter, which was already mentioned, is one of them. Also, both sample cases that will be investigated in more detail in the case studies use this approach.

There exist two methods that have been used by malware to accomplish this. First one involves allocating memory from the target process, writing the payload into the allocated buffer, and finally making sure that the process will execute the payload either by hooking some function or by modifying the context of one of its threads. The second one uses Asynchronous Procedure Call (APC) mechanism provided by the OS. The benefit of the second technique from the malware point of view is that APC is a normal operation and it would be very hard to identify malicious APC operations amongst legitimate operations performed by the OS itself.

Solomon and Russinovich [34] define APC as a function that provides a way for ring 3 and ring 0 applications to execute code in the context of a chosen user thread and hence a process address space. Windows OS supports both user- and kernel-mode APCs. User-mode APC can be used to execute code in ring 3.

The idea of using user-mode APC to execute code from kernel mode in ring 3 is not new. Anatoly Vorobey already brought out this idea in year 1997 [25]. However, his approach did not take into account the need to allocate and map the ring 3 payload code into the address space of the target process. A complete solution with full source code was published in rootkit.com by Valerino in February 2005 [31]. Below is a skeleton code to execute code from kernel mode in ring 3.

```
pMdl = IoAllocateMdl(pPayloadBuf,
       dwBufSize, FALSE,FALSE,NULL);

// Lock the pages in memory
_ _try
{
    MmProbeAndLockPages(pMdl, KernelMode,
IoWriteAccess);
}
_ _except (EXCEPTION _ EXECUTE _ HANDLER){}

// Map the pages into the specified process
KeStackAttachProcess(pTargetProcess,
                     &ApcState);
MappedAddress = MmMapLockedPagesSpecifyCa
che(pMdl, UserMode, MmCached, NULL, FALSE,
NormalPagePriority);
KeUnstackDetachProcess(&ApcState);

// Initialize APC
KeInitializeEvent(pEvent, NotificationEvent,
                  FALSE);
KeInitializeApc(pApc, pTargetThread, Origi-
nalApcEnvironment, &MyKernelRoutine, NULL,
MappedAddress, UserMode, (PVOID)NULL);

// Schedule APC
KeInsertQueueApc(pApc,pEvent,NULL,0)
```

One important thing with user-mode APC is that the target thread has to be in alertable state before it will call the APC function [37].

So far the author has seen Valerino's technique, which is the most advanced one, to be used by only one malware family, named SpamTool.Win32.Mailbot[5]. In many ways this malware family is special and applies lots of ideas presented by various rootkit researchers into real-life malware. Mailbot will be discussed in more detail in Case Study 2, later in this paper.

Case Study 1: Haxdoor

Haxdoor [7][8] is a powerful backdoor with rootkit and spying capabilities. It has been around for a long time but especially during year 2006 it has received lots of attention because of its involvement in various high-profile phishing, pharming and identity theft attacks [13][14][1]. Haxdoor is a good example of today's malware that utilizes kernel-mode code to make its detection and removal more difficult and to circumvent personal firewalls.

First Haxdoor, named Backdoor.Win32.Haxdoor.a[6], was found in August 2003. It had three components, a PE executable, a DLL and a kernel-mode driver. The executable's main purpose was to install and launch the other components. The DLL was the main part implementing backdoor and information stealing functionality. It utilized the driver when it had to do operations that cannot be done from user mode on Windows NT based platforms. The services include low-level I/O operations, dumping of SAM database to a file and hooking of nt!NtQuerySystemInformation from `nt!KiServiceTable` to hide haxdoor's process from other applications.

From those times Haxdoor's driver has evolved but interestingly many of the old features are still present, includ-

5. Also known as Backdoor.Rustock.A (Symantec) and Spam-Mailbot trojan (McAfee).

6. Also detected as Backdoor.Trojan (Symantec) and BackDoor-BAC Trojan (McAfee).

ing I/O operations to enable/disable the keyboard, playing such tricks as opening and closing the CD-ROM tray or in the worst case resetting the CMOS.

During the last year, Haxdoor driver has changed very little. The driver is actually very simple but still does some extremely powerful tricks to make detection and removal of it hard unless its inner workings are well known. Below is the list of most important features it implements:

- process and file hiding
- protection of its own threads and processes against termination
- protection of its own files against any access
- payload injection into created processes

Surprisingly it still misses some important features such as hiding of its launch points from the registry. Today, basic feature set of a rootkit-enabled malware includes at least hiding of its files, registry keys/values and processes.

Process and file hiding are implemented by replacing function pointers for `nt!NtQuerySystemInformation` and `nt!NtQueryFileInformation` in `nt!KiServiceTable` with special hook functions implemented by the driver. These hook functions first call the original service function that fetches the corresponding information from the system beneath and then remove any entries from the collected data that the malware wants to hide from other applications.

Haxdoor protects its own threads and processes in a similar way. Instead it now hooks `nt!NtOpenThread` and `nt!NtOpenProcess`. If the hook function notices that another process tries to open a handle to the malware process or any of its threads with `PROCESS_TERMINATE` or `THREAD_TERMINATE` access rights, it replaces the target id with the one of the calling thread or process respectively. This results that the thread or process trying to terminate the malware gets terminated instead.

```
.text:0001029D hook_NtOpenProcess proc near            ; DATA XREF: sub_1102A+77↓o
.text:0001029D
.text:0001029D ProcessHandle   = dword ptr  4
.text:0001029D DesiredAccess   = dword ptr  8
.text:0001029D ObjectAttributes= dword ptr  0Ch
.text:0001029D ClientId        = dword ptr  10h
.text:0001029D
.text:0001029D                 inc     eax
.text:0001029E                 test    [esp+DesiredAccess], 1 ; PROCESS_TERMINATE
.text:000102A6                 jz      short @@do_nothing
.text:000102A8                 cmp     byte_1455A, 1
.text:000102AF                 jz      short @@do_nothing
.text:000102B1                 mov     eax, [esp+ClientId]
.text:000102B5                 push    edi
.text:000102B6                 or      eax, eax
.text:000102B8                 jz      short @@do_nothing_1
.text:000102BA                 mov     eax, [eax+CLIENT_ID.UniqueProcess]
.text:000102BC                 mov     ecx, dwProtectedProcCount
.text:000102C2                 mov     edi, offset pProtectedProcBuffer
.text:000102C7                 repne scasd
.text:000102C9                 or      ecx, ecx
.text:000102CB                 jz      short @@do_nothing_1
.text:000102CD                 cmp     eax, dwBackdoorProcId
.text:000102D3                 jz      short @@do_nothing_1
.text:000102D5                 mov     eax, large fs:18h
.text:000102DB                 mov     eax, [eax+NT_TEB.Cid.UniqueProcess]
.text:000102DE                 cmp     eax, pProtectedProcBuffer
.text:000102E4                 jz      short @@deny_operation
.text:000102E6                 cmp     eax, pProtectedProcBuffer+4
.text:000102EC                 jz      short @@deny_operation
.text:000102EE /*
.text:000102EE     Switch target id with the id of the source process -> handle will
.text:000102EE     be opened to the source process instead!
.text:000102EE */
.text:000102EE                 mov     ecx, [esp+4+ClientId]
.text:000102F2                 mov     [ecx], eax
.text:000102F4
.text:000102F4 @@do_nothing_1:                         ; CODE XREF: hook_NtOpenProcess+1B↑j
.text:000102F4                                         ; hook_NtOpenProcess+2E↑j ...
.text:000102F4                 pop     edi
.text:000102F5
.text:000102F5 @@do_nothing:                           ; CODE XREF: hook_NtOpenProcess+9↑j
.text:000102F5                                         ; hook_NtOpenProcess+12↑j
.text:000102F5                 push    [esp+ClientId]
.text:000102F9                 push    [esp+4+ObjectAttributes]
.text:000102FD                 push    [esp+8+DesiredAccess]
.text:00010301                 push    [esp+0Ch+ProcessHandle]
.text:00010305                 call    orig_NtOpenProcess
```

Figure 4. Haxdoor protects its own process against termination.

Haxdoor's file protection scheme is very simple but extremely powerful. Any user-mode code has a hard time trying to get access to any file under protection by the malware driver. When the driver starts, it first opens a handle to the protected files with no share access which gives it exclusive access to them. Just as a precaution it also takes an exclusive lock over the whole file content by using `nt!NtLockFile`. Since this is done from the kernel in the system thread context, the created handle is not accessible from user mode. Only way to bypass the file locking is to do it from kernel.

Probably the most interesting and unique feature of the driver is its payload injection into newly created processes. This is done by hooking `nt!NtCreateProcess` and `nt!NtCreateProcessEx` functions from `nt!KiServiceTable`. The former is used on Windows 2000 whereas the latter is used on Windows XP and Windows Server 2003. The consequence is that when a new process is created the hook will be executed. First, the hook calls the original function so that the new process is actually created. Then, before returning control to the creator it executes its additional payload, which is illustrated in the following figure.

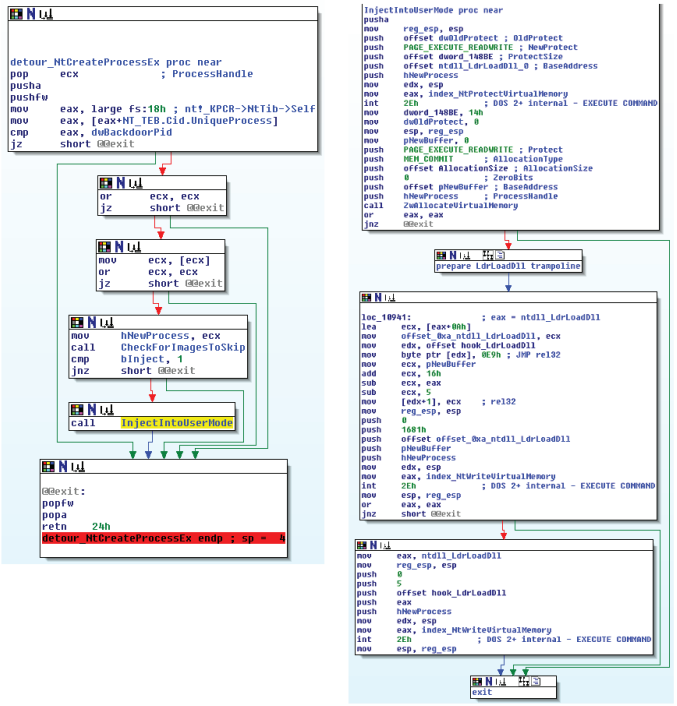

Figure 5. Haxdoor injects its payload into newly created processes before they start to execute.

The payload, `detour_NtCreateProcessEx`, first checks if the calling process belongs to the backdoor or if it is an important system process that should not be tampered with. In case of a normal process it executes the code that will do the actual injection. The purpose of the injection is to copy the position independent code from the driver's data section into the new process' address space and to insert an additional hook into `ntdll!LdrLoadDll`, which will trigger the injected payload when the process starts to execute. First, the driver calls `nt!NtProtectVirtualMemory` to change protection to allow writing to those virtual memory pages that will be overwritten by the hook. Then, it allocates memory from the process' address space and writes the injected payload into it. Finally, it overwrites the beginning of `ntdll!LdrLoadDll` with a relative JMP instruction pointing to the new payload.

The outcome is that whenever the process starts to load a DLL the injected code will be executed. What this code actually does is out of the scope of this paper. Briefly, it will prevent various security product related processes from starting, it will load and execute the backdoor DLL and it will install additional hooks into numerous network related functions that will allow it to conduct phishing, pharming and stealing of user credentials.

Haxdoor is special in many ways. Its early deployment of kernel-mode code and rootkit techniques has made it to stand out from the rest. Its usage of kernel-mode hiding, self-protection and remote injection techniques has made it challenging for security products to deal with. In today's standards its rootkit techniques are nothing spectacular and properly designed kernel-mode real-time antivirus scanners can easily detect and disable its files. However, Haxdoor's code injection technique can still be considered extremely powerful and is still able to bypass several firewalls even though protection against remote code injection is considered to be a basic requirement for a modern firewall.

Case Study 2: Mailbot aka Costrat

Mailbot [9] is the most powerful and stealthiest rootkit seen so far. In many ways it puts into practice the ideas of "Stealth by Design" malware introduced by Joanna Rutkowska in January 2006 [33]. Latest Mailbot variants have only a single kernel-mode driver. However, they are not full-kernel malware since they carry an encrypted DLL that will be executed in ring 3. The DLL is a highly sophisticated spambot with backdoor capabilities. Today, Mailbot's detection and removal is still a challenge to most rootkit detectors and antivirus solutions.

First Mailbot, named SpamTool.Win32.Mailbot.a[7], was found in December 2005. It had three components, a PE executable, a DLL and a kernel-mode driver. The PE executable was a dropper, which installed the other components. The DLL was the main component and it was loaded into Winlogon.exe process when the system started. The kernel-mode driver hooked three functions from `nt!KiServiceTable`, namely `nt!NtEnumerateKey`, `nt!NtEnumerateValueKey` and `nt!NtQueryDirectoryFile`. Their purpose was to hide all files and launch points created by the malware. In addition, it hooked `IRP_MJ_CREATE`, `IRP_MJ_CLOSE` and `IRP_MJ_DEVICE_CONTROL` handler functions from Tcpip driver object to hide all TCP and UDP connections the DLL had established.

The author analyzed his first Mailbot variant, SpamTool.Win32.Mailbot.az[8] [25], in 27th of May 2006 after it was submitted by a malware collector who had noticed that every rootkit detection tool he was using, including F-Secure BlackLight, was not able to find the rootkit. It consisted only of a single kernel-mode driver that was stored as hidden data stream attached to the system32 folder. In three weeks a new and improved version of BlackLight was released that was again able to find the hidden driver from the system [12].

7. Detected also as Hacktool.Spammer (Symantec).
8. Detected also as Backdoor.Rustock.A (Symantec) and SpamMailbot trojan (McAfee).

It was evident that this malware was something special. Since then the author has been closely following its evolution and analyzing its inner workings. This malware applies into real-life many ideas that have been discussed in various rootkit-related sites and security conferences. From the disassembled code it becomes quite evident that the malware author has benefited from source code published in rootkit.com and the book about rootkits [15]. A detailed analysis of the Mailbot was recently published in VB Magazine September issue [5].

A new variant of Mailbot appeared in 3rd of July and it went a step further in its stealth capabilities. Interestingly it was named as Trojan-Clicker.Win32.Costrat.a[9]. Since detailed information is available of the old variants [9][5], the rest of this chapter will talk about some of the new features . *Figure 6* shows the new main routine of a recent Costrat variant.

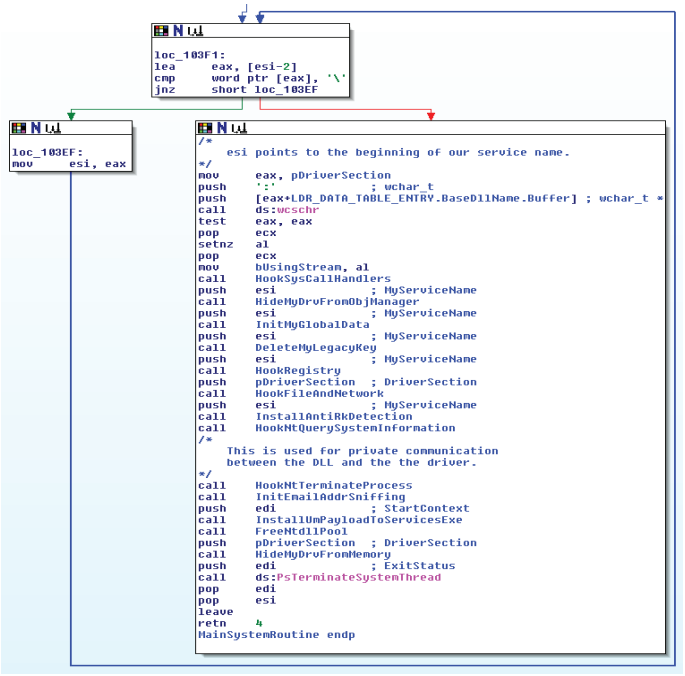

Figure 6. Main routine of Trojan-Clicker.Win32.Costrat.

SpamTool.Win32.Mailbot.az took control of the System Service Dispatcher (SSD) by hooking `INT 0x2E` and `IA32_SYSENTER_EIP MSR` handler function. A thread trying to execute any of the following service functions was redirected to a modified version by setting the thread's `KTHREAD->ServiceTable` field to point to another table created by the malware.

- NtCreateKey
- NtDeviceIoControlFile
- NtEnumerateKey
- NtOpenKey
- NtQueryKey
- NtQuerySystemInformation
- NtSaveKey

Mailbot's approach to hook service functions on a thread-level basis was unique and the stealthiest seen so far. Forensic tools have mainly been looking for hooks from nt!KiServiceTable and from inside system service function. Thus they were not successful in finding mailbot's hooks. The problem was still that some more advanced tools checked the address of `INT 0x2E` and `IA32_SYSENTER_EIP MSR` handler functions. If the address was outside the kernel module, it was a clear indication that something suspicious was going on.

Costrat solved this problem by searching for unused memory inside the kernel module and then redirecting the hook through this address. This is clearly illustrated by the following kernel debugger dumps:

```
kd> rdmsr 176
msr[176] = 00000000`806c15bd

kd> lm a 806c15bd
start    end        module name
804d7000 806e2000   nt

kd> u 806c15bd
nt!_NULL_IMPORT_DESCRIPTOR <PERF>
(nt+0x1ea5bd):
806c15bd e9a249bd77       jmp
lzx32!SysCallHookGen (f8295f64)
```

The debugger output shows that the handler function is in address `0x806c15bd` which is inside the kernel module. The disassembly of the handler function shows how Costrat redirects the execution through this address to the real hook function.

Latest Costrat variants have introduced a new feature that has not previously been available. They hook a new system service function named `NtTerminateProcess`. The actual hooking is implemented in identical way but the hook itself is special. The purpose of the hook is to allow for the injected DLL to communicate with the driver without leaving any extra traces to the system. Figure 7 shows the disassembly of the kernel-mode hook function and the relevant code from the DLL where the covert channel is used.

9. Detected also as Backdoor.Rustock.B (Symantec) and Spam-Mailbot.c trojan (McAfee).

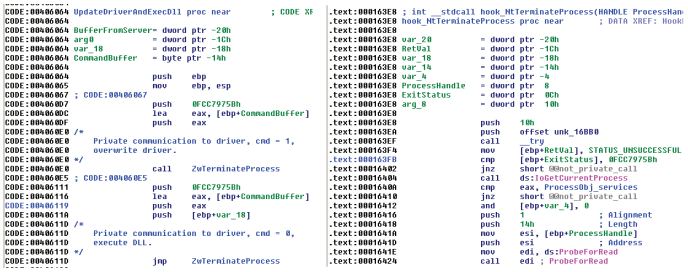

Figure 7. User-mode DLL uses the private communication channel and instructs the driver to update itself.

Latest Costrat variants are able to update themselves on-the-fly from the control servers. The DLL uses its private communication channel to instruct the driver to replace its file on the disk with the new version. Then it commands the driver to inject and execute the new DLL into services.exe. Finally the old DLL releases its resources and terminates. From this point onwards the new DLL is active and interoperates with the old driver. The new kernel-mode code is taken into use only after reboot. Lately this feature has been used quite often to update the list of IP addresses and DNS names it uses when it needs to connect to its control server. This makes it extremely hard to disable the bot network by trying to shutdown its control servers.

Mailbot aka Costrat is powerful. It has features that should not go public. One example is its ability to fully bypass filter drivers. Real-time antivirus scanners and some firewalls are often implemented as filter drivers. They can't do anything if the malicious data never goes through them.

Conclusion

Current security solutions, including antivirus scanners and firewalls, have not been designed to protect the end-user against malicious software that operates in ring 0. There are good reasons for this. One reason is that once the malware executes its code with same privileges as the OS itself, it will become an arms race between the good and bad. This has already been seen with rootkits and their anti-detection engines. After the rootkit notices that it is no longer able to hide from the rootkit detector and is going to loose the game, it changes tactics and starts to make a direct attacks against the detector. It might take a more aggressive approach and prevents the rootkit detector from starting. Or it could directly patch the rootkit detector's code to change its inner logic.

The statistical analysis that was performed, and discussed in the earlier section on History and Trends, showed that there has been a visible rise in the number of malware employing kernel-mode modules as part of their payload. Majority of the modules have been kernel-mode rootkits that the malware uses to hide its main component and thus make its detection and removal as difficult as possible.

This paper has shown the basic techniques that kernel malware is using to do their job. Their main role has been to perform some specific tasks for the main user-mode component. However, the scene is changing. There has been lots of interest in various research groups to investigate for the possibilities to do more complex tasks directly from kernel. The next big thing is going to be the network side. This year we have already seen presentations talking about how backdoors can be implemented directly from kernel mode using only the NDIS layer and custom TCP/IP stack. We have also seen a presentation about bypassing personal firewalls from kernel-mode.

Finally, the antivirus industry has seen Trojan-Clicker. Win32.Costrat aka Backdoor.Rustock.B. For any malware researcher who has been analyzing kernel malware this case should have been an eye-opener. It has shown that complex tasks can be done from kernel mode without affecting the overall system stability. Now, it is time to start thinking how this threat can be countered.

References

1. AusCERT - Media Release - *Response to recent media coverage of the A-311 Death (aka: Haxdoor) trojan*. Available from: <http://www.auscert.org.au/render.html?it=6581>

2. Chiriac, Mihai. (2003). *Virus Analysis 2: XP, A New Virus Playground*. Virus Bulletin Magazine June 2003. ISSN 0956-9979. pp. 7-8.

3. Crazylord. (2002). *Playing with Windows /dev/(k)mem*. Phrack Magazine, Issue 59. Available from: <http://www.phrack.org/phrack/59/p59-0x10.txt>

4. Florio, Elia. (2005). Feature 2: *When Malware Meets Rootkits*. Virus Bulletin Magazine December 2005. ISSN 0956-9979. pp. 7-10.

5. Florio, Elia; Pathak, Prashant. (2006). Rootkit Analysis: Raising the Bar, Rustock and Advances in Rootkits. Virus Bulletin Magazine September 2006. ISSN 1749-7027. pp. 6-9.

6. F-Secure Virus Information Pages : *Gurong.A*. Available from: <http://www.f-secure.com/v-descs/gurong_a.shtml>

7. F-Secure Computer Virus Information Pages: *Haxdoor*. Available from: <http://www.f-secure.com/v-descs/haxdoor.shtml>

8. F-Secure Computer Virus Information Pages: *Haxdoor.ki*. Available from: <http://www.f-secure.com/v-descs/haxdoor_ki.shtml>

9. F-Secure Rootkit Information Pages: *Mailbot. AZ*. Available from: <http://www.f-secure.com/v-descs/mailbot_az.shtml>

10. F-Secure: *News from the Lab - March of 2006 - How Would You Like Your Bagle Done, with Rootkits on the Side?* Available from: <http://www.f-secure.com/weblog/archives/archive-032006.html#00000841>

11. F-Secure: *News from the Lab – March of 2006 – From Russia with Rootkit*. Available from: <http://www.f-secure.com/weblog/archives/archive-032006.html#00000838>

12. F-Secure: *News from the Lab – June 2006 – Hiding the Unseen*. Available from: <http://www.f-secure.com/weblog/archives/archive-062006.html>

13. F-Secure: *News from the Lab - August of 2006 - Haxdoor.KI Being Spammed*. Available from: <http://www.f-secure.com/weblog/archives/archive-082006.html#00000951>

14. F-Secure: *News from the Lab - October of 2006 - Denmark targeted*. Available from: <http://www.f-secure.com/weblog/archives/archive-102006.html#00000988>

15. Hoglund, Greg; Butler, James. (2005). *Rootkits: Subverting the Windows Kernel*. Upper Saddle River, NJ. Addison-Wesley Professional. 324 pages. ISBN 0-321-29431-9.

16. Intel. (2006). *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 1: Basic Architecture*. Mt. Prospect, IL, Intel Corporation. Available from: < ftp://download.intel.com/design/Pentium4/manuals/25366521.pdf>

17. Intel. (2006). *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2A: Instruction Set Reference, A-M*. Mt. Prospect, IL, Intel Corporation. Available from: <ftp://download.intel.com/design/Pentium4/manuals/25366621.pdf>

18. Intel. (2006). *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 2B: Instruction Set Reference, N-Z*. Mt. Prospect, IL, Intel Corporation. Available from: <ftp://download.intel.com/design/Pentium4/manuals/25366721.pdf>

19. Intel. (2006). *Intel 64 and IA-32 Architectures Software Developer's Manual. Volume 3: System Programming Guide*. Mt. Prospect, IL, Intel Corporation. Available from: <ftp://download.intel.com/design/Pentium4/manuals/25366821.pdf>

20. Ionescu, Alex. (2006). *Subverting Windows 2003 SP1 Kernel Integrity Protection* Available from: <Available from: http://www.tinykrnl.org/recon2k6.pdf>

21. Kasslin, Kimmo; Stahlberg, Mika; Larvala, Samuli; Tikkanen, Antti. (2005). *Hide 'n seek revisited – full stealth is back*. IN: Proceedings of the 15th Virus Bulletin International Conference, 5-7 October 2005, Dublin, Ireland. Abingdon, England, The Pentagon. pp. 147-154.

22. Microsoft: Windows Driver Kit (WDK) – Overview. Available from: <http://www.microsoft.com/whdc/devtools/wdk/default.mspx>

23. Nebbett, Gary. (2000). *Windows NT/2000 Native API Reference*. Indianapolis, Indiana, Sams Publishing. 528 pages. ISBN 1-5787-0199-6.

24. Nikishin, Andy. (1999). *Virus Analysis 2, Inside Infis*. Virus Bulletin Magazine November 1999. ISSN 0956-9979. p. 8.

25. NT Drivers: Usenet Archives - Forum: comp.os.ms-windows.programmer.nt.kernel-mode – User mode APCs. Available from: <http://www.cmkrnl.com/arc-userapc.html>

26. Randnut. (2004). *Multiple WinXP kernel vulns can give user mode programs kernel mode privileges*. BugTraq February 18. Available from: <http://www.securityfocus.com/archive/1/354392>

27. Rootkit.com: *Simple Hooking of Functions not Exported by Ntoskrnl.exe*. Available from: <https://www.rootkit.com/newsread.php?newsid=248>

28. Rootkit.com: *Kernel mode sockets library for the masses* Available from: <https://www.rootkit.com/newsread.php?newsid=416>

29. Rootkit.com: *Kernel mode Ircbot* Available

rom: <https://www.rootkit.com/newsread.php?newsid=467>

30. Rootkit.com: *some ideAs About steAlth for rootkit* Available from: <http://www.rootkit.com/newsread.php?newsid=445>

31. Rootkit.com: *Showtime : *WORKING* CreateProcess in KernelMode!* Available from: <https://www.rootkit.com/newsread.php?newsid=259>

32. Russinovich, Mark. (1998). *NT's "\dev\kmem"*. Available from: <http://www.sysinternals.com/ntw2k/info/tips.shtml#kmem>

33. Rutkowska, Joanna. (2006). *Rootkit Hunting vs. Compromise Detection*. IN: Proceedings of the 2006 Black Hat Federal, 25-26 January, Washington DC. Seattle, WA, Black Hat, Inc.

34. Solomon, David; Russinovich, Mark. (2005). *Microsoft Windows Internals: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. 4th edition. Redmond, Washington. Microsoft Press. 935 pages. ISBN 0-7356-1917-4.

35. Szor, Peter. (2000). *Virus Analyses: Poetry In Motion*. Virus Bulletin Magazine April 2000. ISSN 0956-9979. pp. 6-8.

36. Szor, Peter. (2005). *The Art of Computer Virus Research and Defense*. Upper Saddle River, NJ. Addison-Wesley Professional. 744 pages. ISBN 0-321-30454-3.

37. Microsoft Platform SDK Documentation Available from: <http://msdn.microsoft.com/library/en-us/sdkintro/sdkintro/devdoc_platform_software_development_kit_start_page.asp>

38. Hoglund, Greg. (2000). *Loading Rootkit Using SystemLoadAndCallImage*. BugTraq August 29. Available from: <http://www.securityfocus.com/archive/1/79379>