DETECTING INTRUSIONS ON WINDOWS OPERATING SYSTEMS BY MONITORING SYSTEM SERVICES

System Service の監視による Windows 用侵入検知システム

by

Daisuke Shimamoto

島本 大輔

A Senior Thesis

卒業論文

Submitted to

the Department of Information Science

the Faculty of Science, the University of Tokyo

on February 1, 2005

in Partial Fulfillment of the Requirements
for the Degree of Bachelor of Science

Thesis Supervisor: Akinori Yonezawa　米澤 明憲
Professor of Information Science

ABSTRACT

With the growth of the Internet, the danger of intrusions on computer systems has significantly increased. Accordingly, Intrusion Detection Systems (IDS) are playing an important part in computer security. A number of previous works on IDS for UNIX systems have proposed a method of detecting anomalies by monitoring system call sequences. Though the method is expected to be effective on Windows, applying it to Windows is not straightforward. This is because on UNIX systems, requests to Operating System(OS) can be easily monitored by process tracing mechanisms like *ptrace*. On the other hand, on Windows systems, it is not trivial to achieve monitoring of requests to the OS. In this work, we propose an anomaly detection system implemented by monitoring Windows System Services. Windows System Services provide a similar facility to system calls on UNIX systems. Though a number of techniques for monitoring System Services have been proposed so far, we developed a novel technique and applied it to our system. The technique monitors System Services by intercepting the SYSENTER instruction in the x86 architecture. The SYSENTER instruction is a new instruction added to the x86 architecture in 1997. All System Services made from user mode execute this instruction in Windows XP and later. Therefore, all System Services can be monitored by intercepting SYSENTER. We will run our system on Windows XP and evaluate its effectiveness.

論文要旨

近年のインターネットの成長により、コンピュータシステムに侵入される危険が大きくなっており、それに伴い、侵入検知システム（IDS）の重要性が高まっている。これまで、UNIX系 OS のための IDS のいくつかの研究において、システムコール列の監視により異常を検知する手法が提案されてきた。その手法は Windows 上でも有効であると期待されるが、その手法を Windows に適用することは単純ではない。UNIX 系 OS では ptrace などのプロセストレース機構を用いて OS に対する要求の監視を容易に実装可能であるが、Windows では、OS に対する要求をどのように監視するかが自明ではないためである。本研究では Windows の System Service の監視により実装される異常検知システムを提案する。System Service とは UNIX 系 OS におけるシステムコールと同様の機能をもつ仕組みである。System Service を監視する手法は既に複数存在するが、本研究では x86 アーキテクチャの SYSENTER 命令の捕捉を利用する新しい手法を提案し、本研究で実装した IDS に採用した。SYSENTER 命令は 1997 年に加えられた比較的新しい命令である。Windows XP 以降の Windows では user mode より実行されたすべての System Service が SYSENTER 命令を呼ぶため、この命令を捕捉することにより、System Service を監視することが可能である。このシステムを Windows XP 上で動作させ、その有効性を評価する予定である。

Acknowledgements

We would like to thank Professor Akinori Yonezawa for his help. Especially his insightful advices helped us a lot to carry out our work. We would also like to thank Yoshihiro Oyama for his constructive comments.

Contents

List of Figures

Chapter 1

Introduction

A vast number of malicious programs have been introduced and its effect is a big concern among computer users. Its number has increased dramatically since the introduction of the Internet and is growing day by day. A look at the threat lists on anti-virus vendors' homepages gives us an idea of the speed of its growth. These malicious programs can cause severe damage to the system and are becoming more subtle. In the past, viruses used to be made for fun. In other words, they were more a mere joke program than a malicious one. However recent viruses, for example, may crash the system, attack other systems, steal information from the system, execute shells for remote control, etc. For this reason, many studies have been done on security and is common for computers to have anti-virus programs installed. Many commercial anti-virus programs use pattern matching algorithms to identify malicious programs and are very effective against known viruses but are less effective against new, unknown viruses. Recent viruses such as WORM_BAGLE.AT and WORM_NETSKY.Z[1] have shown that new viruses may spread through the Internet at a rattling speed and may reach one's system before the anti-virus program's virus pattern database is updated. In addition, the two viruses mentioned above have many variants which forces the anti-virus programs to be updated for each of the patterns. The update may take some time to come out and the time may be critical.

Anomaly detection is aimed to capture odd operations in the system and may

[1]The names of the viruses are the terms used by Trend Micro.

be able to detect new, unknown viruses earlier. This will prevent the virus from spreading further. This is why anomaly detection are becoming more important. Anomaly detection is a hot topic with many works done on open-source systems like Linux with some that may turn out highly effective. On the other hand, on Windows, which covers approximately 90% of all personal computers, less have been done. This is due to the fact that the internals of Windows systems are not documented much and not well known. Detection methods for open-source systems also have possibilities of being effective on Windows systems if ported but the opacity of the internal system makes developing very difficult. This is true to computer science fields other than security OS.

A common method to detect anomalies on Linux or UNIX systems is using the interception of system calls [1]. Malicious programs usually behave differently from ordinary programs. Therefore the detection system will intercept the system calls and collect information extracted from it such as the address to return to, parameters passed on or log the system call trace. These informations are traced to detect odd behaviours which indicate viruses or compromises. Interception of system calls (or System Services on Windows which will be related to later) for Windows are frequently used by anti-virus programs. The techniques are actually also applied to rootkits. Several methods on interception of the System Services exist and they each have their own strong points and weak points.

Our work introduces an anomaly detection system for Windows based on System Service tracing using a new interception method. Our system intercepts the System Services via modifying the address switched to after the execution of the SYSENTER instruction. This enables the interception of System Services made by user mode programs. The trace of the System Services will be used to detect anomalies.

This paper is organised as follows. Chapter 2 shows related works on intrusion detection systems. Chapter 3 is on interception techniques for Windows systems. Chapter 4 describes our anomaly detection system. Chapter 5 discusses future works and finally chapter 6 summarises this paper.

Chapter 2

Related Works on Intrusion Detection Systems

Intrusion detection system is an important issue which uses anomaly detecting in the security area and many works have been carried out. One main approach for intrusion detection systems is where the detection is done. In this respect, the systems can be separated into two groups [2]: (1) Network-based IDS and (2) Host-based IDS.

Network-based IDS monitors and analyses the data transfered in the network to detect worms and attacks on hosts. Recent viruses, worms, or malicious softwares rely heavily on the network. For example, Many recent viruses have SMTP engines built inside them to spread virus infected e-mails to others and worms such as Blaster automatically attack other Windows systems through the network. Network-based IDS operate on the network and monitor all the data passing to-and-fro and detect anomalies. The IDS operates independently of the hosts connected to the network which means that anomalies for any kind of OS can be detected. The drawback is that it is almost impossible to detect malicious code executing only on the host without using the network. It also may decrease the performance of the network because of the detecting operations.

Host-based IDS executes on the local host and monitors data found on the host such as processes, file I/O, system calls, logs created, etc. Most anti-virus softwares for personal computers fall into this category. This IDS is effective as it can col-

lect enough information from the host and, thus, has a greater chance of detecting the intrusion. The main disadvantage of this IDS is that it will slow down the performance of the system it is operating on. This is because the IDS will have to take part of the CPU time for logging and analysing to detect intrusions. This is a significant issue and a through tests has been carried out by Gao [3]. The deeper the IDS analyses, the slower the host will get. There are several methods of collecting information. One is monitoring system calls made by the target process. The following are some examples.

Method of monitoring the parameters and return values of the system calls is one example. To give a specific example, if the program passes a parameter including the string *"/usr/bin/bash"* to the system call *exec*, the program may have been attacked by some malicious code and the code is trying to spawn a remote shell. Another example on return values is when the address after returning from some function is outside the range of the code area and pointing to a injected malicious code on the stack. This can happen when a buffer overflow attack had been done on a program with security vulnerabilities and is one of the most common methods of taking over the host. Checking the stack is also a method of detecting anomalies. In [4], the call stack is used to extract the return addresses and generate a execution path. If the execution path runs through a data sections, it may be a sign of anomaly. Monitoring program counters is also a factor used to detect anomalies. In the work by Sekar [5], the IDS keeps track of the system call and the program counter for analysing. A similar work done by Rutkowski [6, 7] analyses the execution path by monitoring the number of instructions executed for a certain operation. If, on some instance, the operation took more instructions than usual, the path may be executing a malicious code inserted.

Another approach for IDS is the comparison of signature-based and anomaly-based detection.

Signature-based IDS statically matches the data to be checked with its database of malicious signatures. Any data matching the database will be detected as anomalies. This detection is similar to it of the more common anti-virus programs. The main drawback is also very similar: The program's effectiveness relies on its database. The database needs to be updated periodically to detect novel attacks.

Anomaly-based IDS has information on the normal activities of the system and any operation which does not follow them will be detected as anomalies. This method is powerful as any peculiar operation is reported and the IDS will give warnings to novel attacks.

Chapter 3

Interception Techniques on Windows Systems

Interception techniques are also known under the name of "hooking" or "spying" techniques. It is a popular technique with much work done on various systems. It may be used for various purposes. The following are some examples of the purposes.

Monitoring calls to functions using intercepting techniques is applied to identify the action of the functions and the program as a whole. It enables the user to recognise when and how the functions are called. This will bring about a better understanding about the program. Analysing some existing programs through its normal outputs is very tiresome and difficult and interception techniques can, for example, make this process easier. Some functions of the OS are not well documented or not documented at all. This is especially true for commercial OS. Interception techniques can offer deeper information about this kind of OS and help system software programmers. Another purpose is to extend the action of an existing functions. For example, functions supplied by commercial programs may not be suitable enough for the user's purpose and he/she may want to extend it to meet his/her needs. The intercepting can enable the user to add extra features.

Various interception techniques for Windows systems have been introduced. In this section, we will show some of the techniques that may have been used for developing an anomaly detection system. We have separated the techniques into two groups, user mode techniques and kernel mode ones.

3.1 User Mode Techniques

User mode techniques are more or less easier to develop and implement compared to kernel mode ones. This is because they do not work in the kernel and will rarely create a Blue Screens of Death(BSODs). This section describes some major methods.

SetWindowsHookEx() [8]

This technique uses the function SetWindowsHookEx() which is officially provided by Microsoft for developers. This "hook" function uses the message handling mechanism on Windows. Windows systems, especially the GUI part, are based on the message handling mechanism. In this mechanism, GUI or timer events are sent to the program as messages to be processed. When SetWindowsHookEx() is called, it captures messages sent to the program before it reaches the original processing code and enables the user to do modification or logging of the messages. For example, the programmer can even block certain messages such as keyboard inputs and the program will not be able to obtain any keyboard inputs.
The fact that this function is officially supported by Microsoft which means that it is well documented is a large advantage. It is very common to use this technique to intercept window messages but it cannot intercept other functions which have nothing to do with the message handling mechanism.

Proxy DLL

Using proxy Dynamic Link Libraries (DLL) is an intuitive method to intercept functions. Windows programs import functions located in the Export Address Table(EAT) of the DLL. This import is done by the Windows loader at the runtime of the program.
The original DLL is substituted by an user made proxy DLL which contains the same functions as the original but each redefined. Then when the target program calls a function residing in the DLL will call the user defined functions instead of the original. The user defined functions usually calls the original function during its execution. For example, intercepting functions of the Winsock library(i.e.

wsock32.dll) enables one to check network activities done by the target program. A working example of this DLL has been created by Matt Pietrek and explained in [9].

There are three drawbacks of this method. Firstly, the interception will only be active if the target program uses the user made proxy DLL. Therefore, if the target program uses an original DLL or does not use the DLL at all, the interception will not be done. Secondly, the DLL searching order of the directories has changed since Windows XP SP1. Prior to Windows XP SP1, the Windows loader would search for the DLL in the following order: (1) Same directory as the program invoked, (2) the current directory, (3) the system directory (usually C:\WINDOWS on Windows XP unless modified at installation), (4) directories listed in the PATH environment variable. However, for Windows XP SP1 and its successors (including Windows Server 2003), it will search firstly in the system directory, secondly in the current directory, lastly in the PATH environment. This change has been made to prevent a malicious code writer from locating a proxy DLL in the same directory as the program and intercepting the function calls in order to attack the system. The third drawback is the large amount of code writing necessary to make the DLL. The programmer will need to write as many functions as the original DLL has. For example, kernel32.dll exports nearly 1000 functions resulting in the programmer writing as much.

Patching the Program

This technique rewrites the program in order to intercept functions. The patching may be done either on the program file itself or in the memory after the program is loaded.

This technique is very effective as it is able to not only intercept functions but modify the program itself and is usually used by commercial software to update its files. A well known example is Windows Update used by Windows systems for updating its system. When Windows Update is launched, a patch file is downloaded and applied to the original file to be updated. This is useful because the file to update may be large and it may not be practical to download a new version as a

whole.
The main drawback of this technique is that the target program is usually already compiled and only a binary exists. Therefore, it is difficult to detect which part of the program to patch and that a implementation of a disassembler is needed to analyse the machine instructions. Another drawback is the uninstalling of the interception when it not needed anymore. The programmer needs to make a program to take the patch off.

Patching the API

This technique is similar to rewriting the program. The interception is achieved by modifying the API itself. A control transfer instruction such as a JMP or CALL instruction is inserted at the beginning of the function. After the function is intercepted, the patched code will execute the original instruction in order to maintain regular operation.
The limitation of this technique is that it is necessary for the target function to have at least 5 bytes. This is due to the fact that the JMP or CALL instruction is 5 bytes long. If the target function is less than 5 bytes, the JMP or CALL instruction will exceed the original length and the target program will be corrupted and will not execute correctly.
An example is given in Figure 3.1. The figure shows the code before and after the interception. A working example named Detours has been developed by Microsoft's research department [10].

Input Address Table Patching

The technique of patching the Input Address Table(IAT) is very common. It is fairly easy to implement and powerful.
The IAT is a table residing in the Windows program file. This file format is called the Portable Executable(PE) format which is based on the Common Object File Format(COFF) used by UNIX [11, 12]. The program file also contains the EAT mentioned previously and the IAT is used for the opposite purpose. It is used to import the addresses of externally defined functions. This importing of the

Before patching

```
;; Target function
Sleep:
  push ebp        ;; 1 byte
  mov  ebp, esp   ;; 2 bytes
  push ebx        ;; 1 byte
  push esi        ;; 1 byte
  push edi        ;; Sleep+5
   ⋮
;; Trampoline function
UntimedSleep:
  jmp  Sleep
;; Detour function
TimedSleep:
   ⋮
```

After patching

```
;; Target function
Sleep:
  jmp  TimedSleep  ;; 5 bytes
  push edi         ;; Sleep+5
   ⋮
;; Trampoline function
UntimedSleep:
  push ebp
  mov  ebp, esp
  push ebx
  push esi
  jmp  Sleep+5
;; Detour function
TimedSleep:
   ⋮
  jmp  UntimedSleep
```

Figure 3.1: A example of patching the API. The first 5 bytes of the target function moved to the trampoline function and overwritten by a jmp instruction to the Detour function.

functions is done by the Windows loader at program start up. The reason for this complexity is due to the fact that the directories of the DLL files are not fixed and may be different in each computer. Thus, the addresses of the functions need to be defined dynamically.

The interception of the functions are done by overwriting this IAT. This is done by modifying the loaded image inside the memory. The programmer will usually store the original addresses inside the IAT of the program and overwrite them with the address of the user defined functions. The user defined codes are able to do the job needed, call the original function stored and return back to the program. This is only the main part of the intercepting and more has to be done to intercept

functions but that is out of the scope of this paper. Further information on the implementation of this technique can be found on [13].
This technique enables the interception of functions imported by DLL and is very powerful when it is necessary to intercept these functions. However, paradoxically, the drawback is that there is no way to intercept functions that are not in the IAT. It is very common for programs (including malicious programs) to obtain the addresses of the DLL exported functions by calling the *LoadLibrary()* and *GetProcAddress()* functions exported by Kernel32.dll. The *LoadLibrary()* function, which takes the name of the DLL as its parameter, will load the DLL into the memory image of the program. A successive call to the *GetProcAddress()* function will enable the program to obtain the address of any function exported by the loaded DLL. This method of obtaining the address does not use the IAT and as a result, the interception will not be done.
The Detours [10] mentioned previously also has this feature of rewriting the IAT.

3.2 Kernel Mode Techniques

In the following techniques, the interception takes place in kernel mode. The main parts of the OS are processed in kernel mode, thus, this technique is more powerful than user mode techniques. However, the implementation is difficult compared to user mode techniques as an error in the kernel may end up in BSODs.

System Service Descriptor Table Patching

This technique is highly powerful. It is a technique which was introduced by Russinovich [14] and many works relating to this technique have been done. A work on intrusion prevention systems by Battistoni is also implemented by this technique [15] and another work on restricting the user to running only approved softwares is described in [16] and it uses the same technique.
System Services in the Windows systems are similar to system calls on UNIX or UNIX-like OS. They run in kernel mode and execute the critical features of the system. They are exported by the kernel executable, NTOSKRNL.EXE. The number of System Services on Windows XP counts up to 1,300. Windows 2000, XP, 2003

Server have user mode subsystems to enable not only Win32 programs but also POSIX or OS/2 programs to be source level compatible and the system may seem very sophisticated. Actually all subsystems eventually call the System Services. For example, Win32's *OpenFile* function and POSIX's *open* differ in its parameters but after some processing, they both call the System Service *NtOpenFile* in order to open the file and obtain a handle to it. Each System Service have an individual Service ID. The System Services are organised in a table called the System Service Descriptor Table(SSDT) which uses the Service ID as indexes and the entry corresponding the Service ID contains the addresses of the actual functions. This is identical to the system call table in Linux. As the System Services must run in kernel mode, it is necessary for the call to the System Services from user mode to switch to kernel mode. This is done by filling the EAX register with the appropriate Service ID and raising a software interrupt, int 0x2E, on Windows NT and 2000 or executing the SYSENTER instruction on Windows XP and 2003 Server. The call t the System Services from kernel mode is simply a jump to the address. This technique is similar to the system call table hooking adopted by Mark Russinovich's "Filemon for Linux" [17] or Intel's "Vtune Performance Analyzer" [18].

The interception is achieved by overwriting the addresses in the SSDT to the programmer defined code. This overwriting is similar to that of the IAT patching. It is possible to intercept all calls to the System Services. The System Services are the critical functions of the OS and intercepting them will enable the programmer to understand and modify the action of the OS at a deeper level than user mode techniques. Figure 3.2 is a diagram of intercepting the *ZwCreateFile*.

The drawback of this technique is the amount of coding necessary. A new function has to be defined for each System Service to intercept. It is clear that the newly defined function has to call the original function in order to let the OS run properly. This is true to all the functions resulting in a large amount of coding if the number of System Services to intercept is large.

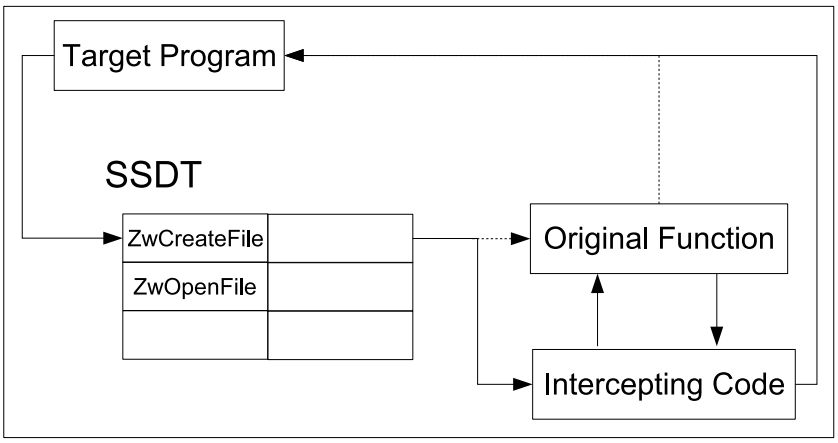

Figure 3.2: Interception of ZwCreateFile. The dotted line indicates the original execution flow and the normal line indicates the flow after the interception is applied.

Filter Drivers

This technique is very common among anti-virus softwares. A filter driver is a driver program which operates between the software and the device driver or between two drivers. It captures I/O requests in the form of I/O Request Packets (IRPs), which are eventually sent to device drivers. When the IRP reaches the programmer's filter driver, it may be modified to fulfill the programmer's needs and passed down to the next driver or passed down without any modification. Anti-virus softwares capture file I/Os and match the actual I/O done with their virus database in order to detect viruses. If the I/O done matches any of the viruses, it blocks the I/O and warns the user about the virus. If it does not match any, it simply passes the IRP down to the next driver for the I/O to be completed. This technique is very effective as all I/O request go through the filter driver and, thus, are checked.

The disadvantage of this technique is that only IRPs can be intercepted and thus, the interception is limited to I/O requests. As all requests are checked, the system is prone to slow down and may degrade in overall performance.

Chapter 4

IDS using SYSENTER

As described in the previous chapter, many works have been done on intercepting functions and several techniques exist on Windows systems. Our work will introduce a new intercepting technique using the SYSENTER instruction. It is fairly easy to implement and also useful.

4.1 The SYSENTER instruction

First of all, we will discuss the SYSENTER instruction. This instruction is new compared to the other traditional instructions in Intel's IA-32 architecture. It was added to the IA-32 architecture in 1997. The first CPU to support this instruction is the Pentium II processor. AMD's IA-32 CPUs also support a very similar instruction called SYSCALL. The SYSENTER is used for a user mode program to switch to kernel mode and execute privileged instructions such as system calls or System Services. The SYSENTER instruction uses the following three Machine Specific Registers(MSR). These registers are defined inside the CPU and each MSR has an independent address.

- SYSENTER_CS_MSR $\cdots$ 32-bit segment selector for the privilege code segment.
- SYSENTER_EIP_MSR $\cdots$ 32-bit offset into first instruction of the privilege code.

- SYSENTER_ESP_MSR $\cdots$ 32-bit stack pointer for the privilege code.

When the instruction is executed, the following steps are taken by the CPU.

1. Loads the SYSENTER_CS_MSR into the CS register (segment selector).
2. Loads the SYSENTER_EIP_MSR into the EIP register (instruction pointer).
3. Adds 8 to the SYSENTER_CS_MSR and loads it into the SS register.
4. Loads the SYSENTER_ESP_MSR into the ESP register (stack pointer).
5. Switches to privilege level 0.
6. If the VM flag in the EFLAGS register is set, it is cleared.
7. Begins the execution.

More detail on the SYSENTER instruction can be found in the Intel CPU manual [19].
The Windows systems have adopted this instruction since Windows XP. Prior to Windows XP, Windows NT (e.g. Windows NT 3.51, NT 4, 2000) used software interruptions (i.e. int 2E) to switch to kernel mode. It was necessary for the OS to execute several instructions in order to achieve the same result as executing SYSENTER. New versions of the Linux kernel have also adopted the feature.

4.2 Interception using SYSENTER

In our IDS, the SYSENTER_EIP_MSR will be modified to intercept calls to the System Services. The modification is done by executing the *WRMSR* instruction. This instruction writes the value in EDX:EAX to the address of the MSR specified by ECX. The address for SYSENTER_EIP_MSR is 176h. EDX:EAX constructs a 64 bit address but on 32 bit Windows XP the EDX was always 0 which implies that only the EAX register indicates the 32 bit address of the first instruction to be executed. Figure 4.1 shows an example of an assembly code to modify the SYSENTER_EIP_MSR. Our system is basically the same.

```
  mov ecx, 176h    // Loads the address of SYSENTER_EIP_MSR for reading.
  rdmsr            // Reads the MSR.
  mov oldEIP, eax  // Stores the real SYSENTER_EIP_MSR.
  cli              // Clears interrupts.
  mov ecx, 176h    // Sets the address for SYSENTER_EIP_MSR.
  xor edx, edx     // Sets EDX to 0.
  mov eax, stub    // Sets the Instruction Pointer of the next instruction.
  wrmsr            // Modifies the MSR.
  sti              // Sets interrupts.
   ⋮
stub:              // This part is executed when interception is done.
   ⋮               // The logging is done here.
   ⋮               // (The EAX register holds the Service ID)
  jmp [oldEIP]     // Jumps to the real SYSENTER_EIP_MSR.
```

Figure 4.1: Sample code to modify the SYSENTER_EIP_MSR. This part of the code was written in inline assembly, thus // is used for comments.

The intercepting code under the label *stub* will be executed when a user mode program calls a System Service. In our system, when the intercepting code starts executing, it takes logs of the Service ID stored in the EAX register. The advantage of this technique is that it is very easy to add or delete the ID to be logged. The technique which uses System Service Descriptor Table Patching described in Section 3.2 can also be used to log the Service IDs but each addition of a new ID needs a new function. In our technique, only the following 2 lines are needed to be added in order to log a new ID. It is apparent that our technique is easier to implement.

```
cmp eax, (Service ID to log)
jmp log
```

4.3 Detection Program

The code in the previous section needs to be executed in kernel mode. This implies that the code needs to be stored in the kernel's address space. On Windows systems, kernel mode operation can be achieved by implementing a device driver. Although user mode device drivers exist, most device drivers reside in the kernel's address space and run in kernel mode. We have implemented a kernel mode device driver which injects our interception code. After the code is injected, the SYSENTER instruction will make the control jump to the code injected by our device driver. The code is executed and the control will jump to the original address to continue the execution of the System Service. The control will switch to our interception code only when the SYSENTER instruction is executed and thus, the interception occurs only when an "user" mode program calls a System Service. Calls made directly to the System Services from kernel mode (e.g. other device drivers) do not execute SYSENTER and can not be intercepted. This is a drawback but a significant proportion of viruses run in user mode and our system has the possibility of being effective against them.
Together with the device driver, a user mode program to load and start the driver is needed. In our implementation, the user mode program will not only insert the driver but will also do the detection. We could have implemented the detection inside the kernel but this is dangerous as a fault inside the kernel will create a BSOD and stop the computer. This means that each bug inside the driver may create a BSOD. The device driver, when loaded, creates a Device Object and a Symbolic Link. A Device Object is a object inside the Windows system and is created under the */Device* directory. This directory is identical to the */dev* directory in Linux. The Symbolic Link is a link to the Device Object and is used by user programs to gain access to the device driver. User mode programs are not permitted to access */Device*. The user program opens the Symbolic Link and reads or writes to it as if it were a normal file. The device driver will receive the request as an IRP and is responsible for it to be processed. In our case, when our user mode program requests a read operation to the device driver, the driver will receive the request as an IRP and return the System Service trace logged. Other requests such as write

requests will be ignored. The log inside the device driver will be cleared. After receiving the logged data from the driver, the user program will use the data to detect anomalies. Figure 4.2 shows a diagram of how are system operates. For the moment, our program will write the logged data to a text file. The text file contains the Service IDs in the order they were called. The reading from the device driver is done by spawning a thread and reading at a defined interval (e.g. every 3 seconds). Figure 4.3 is a screenshot of our user mode program.

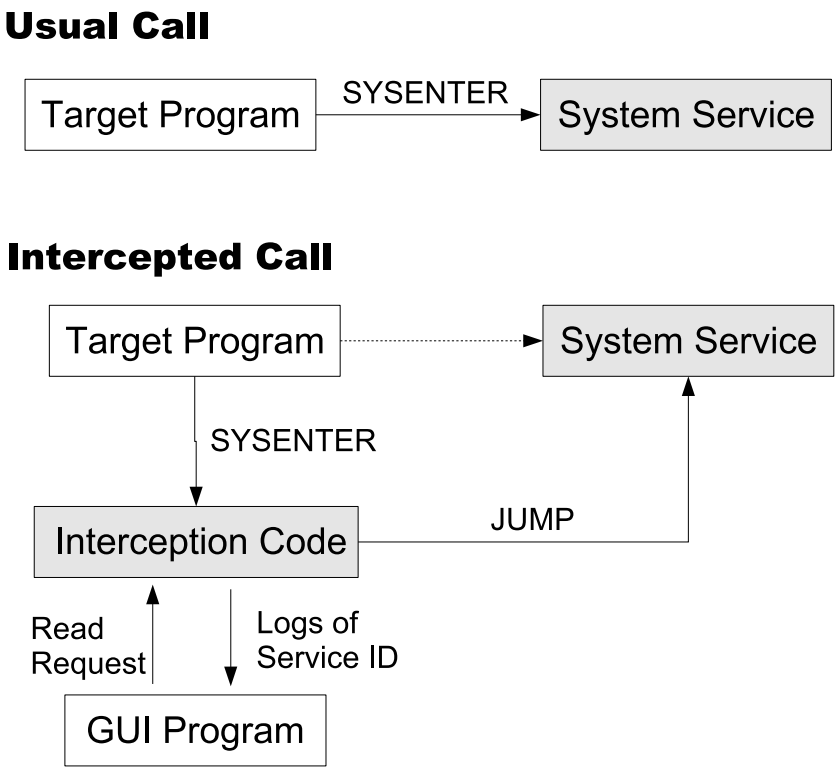

Figure 4.2: This is a diagram of our interception code. The top half is the diagram of an usual call to the System Service. The bottom half is the diagram after the interception code has been injected. The gray boxes indicate the parts which run in kernel mode.

The detection is not yet done due to the fact that a through study on the nature of viruses or other malicious codes is still needed and, thus, we do not have good

System Service patterns to fulfill our requirements.

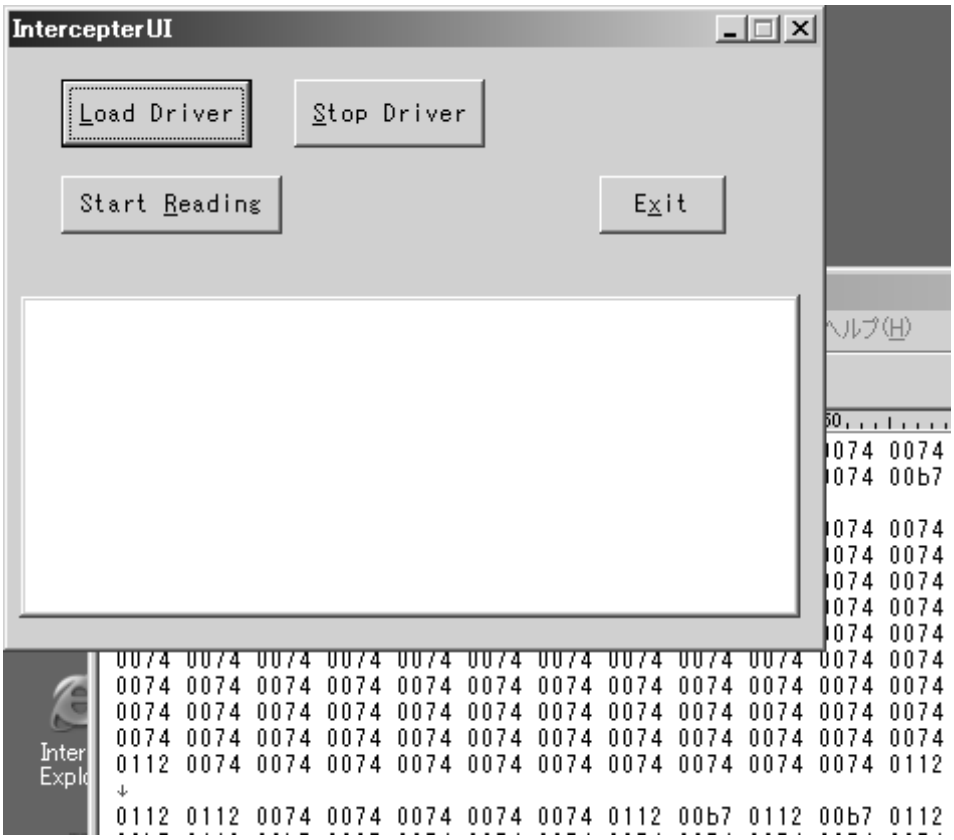

Figure 4.3: User mode GUI of our program. The window below is a text editor showing the log file created by our program.

4.4 System Service IDs to check

The important part of an IDS which uses system call traces is defining the pattern. Although many patterns are possible, there are patterns which are strange compared to patterns without any anomalies. For example, viruses which spread through e-mails (e.g. WORM_BAGLE.AT or WORM_NETSKY.Z) usually collect e-mail addresses by searching text files inside the host. This searching may yield a System Service trace with repeated calls to the System Services, *NtOpenFile*(opens a file) and *NtReadFile*(reads a file). Similarly, this kind of viruses has a built-

in SMTP program to send infected e-mails to the e-mail addresses searched. In Windows systems, sending data over the network will open a handle to the device driver of the ethernet driver and write data into it. Accordingly, this will create a call to the System Service, *NtOpenFile* and *NtWriteFile*(writes to a file). As a result, the propagation via e-mail will yield a log with many calls to *NtOpenFile* and *NtWriteFile* in a short time.

4.5 Results

We have run our program on Windows XP SP2 operating on VMWare 4.0.0 build-4460. The reason for the use of VMWare is for security reasons. With the use of VMWare, the virus can be run on the virtual machine without affecting the host machine. Our program will then collect the traces. After the log is taken, the virtual machine can be switched off and the virus will be gone. We used the snapshot feature and returned to the state when the machine was not infected. The remote debugging of our device driver was also done using VMWare.
So far, we have collected traces of some viruses such as WORM_BAGLE.AT and WORM_NETSKY.Z. They are shown together with traces of normal operation such as launching Internet Explorer. Figure 4.4 shows unique parts of the logs created by our program. For the moment, our program logs calls to *NtCreateProcess(002F)*, *NtCreateProcessEx(0030)*, *NtCreateThread(0035)*, *NtOpenFile(0074)*, *NtWriteFile(0112)*, *NtReadFile(00B7)*, and *NtDeleteKey(003F)*. Each 4 digit hex value above and in figure 4.4 indicates the Service ID of the System Service.
Although we have not yet collected many logs, significant differences can be seen among the logs. The log of the normal operations are more or less mixed up with different calls occurring singly. In the case of the logs under infection, the same System Service tend to be called one after another. When the system was infected by WORM_NETSKY.Z, the *NtReadFile(00b7)* is called many times in a short period. The same can be noticed in the case of WORM_BAGLE.AT which calls *NtOpenFile(0074)* similarly. The detection may use this fact. We have collected other viruses' logs such as VBS_LOVELETTER.A, WORM_SASSER.B, WORM_MSBLAST.E and WORM_MUMU.B. They all have unique logs and strange

traces were seen.

Normal Operation(Starting Internet Explorer)

0112 0112 00b7 00b7 0074 0074 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 0074 00b7 0074 00b7 0074 00b7 0074 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 00b7 00b7 00b7 0074 00b7 0074 00b7 0074 00b7 00b7 00b7 00b7 00b7 00b7 00b7 0035 0074 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074 00b7 0074

Infected by WORM_BAGLE.AT

00b7 00b7 00b7 0112 0112 00b7 00b7 00b7 0112 00b7 0112 00b7 0112 00b7 0112 0112 00b7 00b7 0112 0112 00b7 00b7 00b7 0112 00b7 0112 00b7 0112 00b7 0112 0112 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 0074 00b7

Infected by WORM_NETSKY.Z

0112 0112 00b7 00b7 00b7 00b7 0112 00b7 0112 00b7 0112 00b7 0112 00b7 0112 00b7 0112 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 0074 0074 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 00b7 0112 00b7 0112 00b7 0112 00b7 0112 00b7 0112

Figure 4.4: Logs created by our program. Each 4 digit hex value indicates the Service ID.

Chapter 5

Future Work

Our program is in the middle of development and a lot has to be done.
First of all, the patterns of the Service IDs have to be worked on. The logs from the viruses look promising but more is needed. Additionally, more logs on normal operations have to be collected to compare them with logs of viruses. After collecting enough information, we will construct a pattern to detect anomalies.
The usage of other information is one way to make our program more effective. Currently our program only logs the Service IDs but other values such as Process IDs, parameters passed on to the System Services and return addresses may be useful. Using these new data, ideas and methods of previous works on Linux or UNIX described in chapter 2 may be ported to our program.
Another field which has to be worked on is the recovery. If a malicious code writer knows about the mechanism of the technique, they may rewrite it back to the normal state and the interception will be taken off. Thus, they may execute their malicious codes. without any detection done. For example, in the case of our work, the malicious code writer may rewrite the SYSENTER_EIP_MSR to its original value. One idea to prevent this is to run a separate thread which periodically checks the value of the SYSENTER_EIP_MSR and if it was modified, rewrite it again to jump to our code.

Chapter 6

Conclusion

We have worked on a new technique to intercept calls to the System Services. It uses the rewriting of the SYSENTER_EIP_MSR to force the SYSENTER instruction to jump to our code. The SYSENTER instruction is made when a program is operating in user mode and makes call to the System Service. Another jump to the original code is made after our code is executed to continue the normal execution of the System Service.

Many ideas on intercepting the execution of programs have been introduced and many have been described in this paper. Although one of the other techniques, System Service Descriptor Table Patching, may also be used for the same purpose as our program, it needs much coding to intercept large numbers of System Services. Our program, on the other hand, is simple as it only compares the value of EAX with the Service IDs to be logged immediately after the SYSENTER is executed.

Our program is far from complete and more data is needed to make efficient detections but our simple experiment using existing viruses yielded promising results with unusual traces.

References

[1] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, p. 120. IEEE Computer Society, 1996.

[2] J. Allen, A. Christie, W. Fithen, J. McHugh, J. Pickel, and E. Stoner. State of the practice of intrusion detection technologies. Technical Report CMU/SEI-99TR-028, 2000.

[3] Debin Gao, Michael K. Reiter, and Dawn Xiaodong Song. On Gray-Box Program Tracking for Anomaly Detection. In *USENIX Security Symposium*, pp. 103–118, 2004.

[4] Henry Hanping Feng, Oleg M. Kolesnikov, Prahlad Fogla, Wenke Lee, and Weibo Gong. Anomaly Detection Using Call Stack Information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, p. 62. IEEE Computer Society, 2003.

[5] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *SP '01: Proceedings of the IEEE Symposium on Security and Privacy*, p. 144. IEEE Computer Society, 2001.

[6] Jan Krzysztof Rutkowski. Execution path analysis: finding kernel based rootkits. *Phrack Magazine # 59*, July 2002.

[7] Jan Krzysztof Rutkowski. Advanced Windows 2000 Rootkits Detection(Execution Path Analysis). *Black Hat USA 2003*, July 2003.

[8] MSDN Library. *SetWindowsHookEx*. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/hooks/hookreference/hookfunctions/setwindowshookex.asp.

[9] Matt Pietrek. Under The Hood. *Microsoft Systems Journal*, September 1997. http://www.microsoft.com/msj/0997/hood0997.aspx.

[10] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. pp. 135–144, July 1999. http://www.research.microsoft.com/sn/detours/.

[11] Matt Pietrek. *Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format*, February 2002. http://www.msdn.microsoft.com/msdnmag/issues/02/02/PE/default.aspx.

[12] Matt Pietrek. *Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format, Part 2*, March 2002. http://www.msdn.microsoft.com/msdnmag/issues/02/03/PE2/default.aspx.

[13] Anton Bassov. *Process-wide API spying - an ultimate hack*. The Code Project, March 2004. http://www.codeproject.com/system/api_spying_hack.asp.

[14] Mark Russinovich and Bryce Cogswell. Windows NT System-Call Hooking. *Dr. Dobbs Journal*, January 1997.

[15] Roberto Battistoni, Emanuele Gabrielli, and Luigi V. Mancini. A Host Intrusion Prevention System for Windows Operating Systems. In *ESORICS*, pp. 352–368, 2004.

[16] M.Schmid, F. Hill, and A. K. Ghosh. Moderating the Execution of Applications on Win32 Platforms.

[17] Mark Russinovich. *Filemon for Linux*. http://www.sysinternals.com/linux/utilities/filemon.shtml.

[18] Intel(R). *Intel VTune Performance Analyzer*. http://www.intel.com/software/products/vtune/.

[19] Intel(R). *IA-32 Intel(R) Architecture Software Develope's Manual Volume 2B: Instruction Set Reference, N-Z*, 2004.