INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

Rootkits - Detection and prevention

André Jorge Marques de Almeida

Dissertação para obtenção do Grau de Mestre em

Engenharia Informática e de Computadores

Júri

Presidente:	Professor Jose Carlos Martins Delgado
Orientador:	Professor Carlos Nuno da Cruz Ribeiro
Vogal:	Professor Miguel Pupo Correia

Setembro 2008

Agradecimentos

Tenho a agradecer ao meu Orientador, o Professor Carlos Ribeiro, pela coragem em alinhar nesta minha ideia arrojada de explorar um tema tão obscuro, invulgar e complexo como este. Por outro lado, apoiou-me e ajudou-me a estruturar a dissertação e a definir os objectivos deste trabalho, algo que me preocupava desde início.

Grande parte desta dissertação foi estudada e escrita no Tagus Park juntamente com outros amigos meus destinados ao mesmo: São eles o Henrique Costa (que teve a paciência rara de ler esta dissertação e me indicar aspectos a melhorar), o Nuno Monteiro e o João Mendes, que vendo as suas férias limitadas por causa da dissertação, ajudaram-me a tornar mais leve a execução de um trabalho desta dimensão.

Devo agradecimentos ao Daniel Almeida, ao Sérgio Almeida e ao Telmo Rodrigues pelas constantes dicas e ideias que me forneceram durante várias discussões que tivemos sobre estes temas, e agradeço incondicionalmente ao meu irmão, Filipe Almeida, por dispensar um pouco do seu precioso tempo para ler esta dissertação e me indicar vários pontos a melhorar, não só ao nível da qualidade da escrita mas também do conteúdo.

Seria quase criminoso, abandonar esta secção sem agradecer ao meu Pai e à minha Mãe. Quero agradecer aos meus pais por terem feito os possíveis ao seu alcance para me ajudarem. Uma vez que não dominavam o assunto, ajudaram-me em pormenores de escrita e de construção frásica.

Lisboa, November 23, 2008

André Jorge Marques de Almeida

Abstract

This thesis pretends to explore the underworld of the complex and unwanted software that subtly modifies the behavior of the operating system without users or administrators notice. This type of software is named "Rootkit". This thesis does not pretend to promote the creation of these type of attacks, but instead, help building defenses. No defense is successful if we don't study first the point of view of the attacker. It is precisely in this way that this work is presented: For each defense, many attack vectors possibly used by the attacker are presented to the reader, creating room for understanding what follows. After that, without neglecting the other side, the evidences left by the rootkits are studied and analyzed, in an attempt to prevent their respective attack in the future.

This essay starts outlining some Rootkit techniques commonly used in Windows operating system, with the objective of understanding where and how these methods can be applied in Linux. "Hooking" techniques are a powerful method not only for those who attack but also for those who defend. Thus, their relevance in this work. Hooking is a technique that intercepts the execution flow of an application, or even Kernel, and allows the attacker to obtain important data or modify information.

It is interesting to notice the multiple places of an operating system sensible to "Hooking". Either in User-mode or in privileged mode, there are crucial zones sensible to slight changes, creating a different global behaviour of the system, helping the attacker covering his tracks. This way, the system will provide false information for an application that requests some state of the operating system.

Fortunately, knowing the inner-workings of these subversive programs, we can detect their presence if they are installed. Typically, it is always possible to detect traces of intrusion, but depending on the technological advance of the Rootkit, this can be an extremely difficult task.

This work shows the efficiency and the dangerousness of the technique "Hardware Breakpoint Hooking", and also presents a method for detecting and preventing this same technique (but not infallible). Additionally, code modification attacks in user-mode are studied, to which are proposed a memory scanning tool. Finally, a proactive detection mechanism is presented which scans many sensitive places of the operating system during its operation.

Resumo

Esta dissertação pretende explorar o submundo do complexo software indesejado que modifica subtilmente o comportamento do sistema operativo sem que o utilizador ou administrador se aperceba. A esse tipo de software dá-se o nome de "Rootkit". O objectivo desta dissertação não é o de fomentar a criação deste tipo de ataques, mas antes pelo contrário, ajudar a criar defesas. Nenhuma defesa é bem conseguida se não estudarmos primeiro o ponto de vista do atacante. É precisamente desta forma que este trabalho é apresentado: Para cada defesa, são primeiro apresentados ao leitor as várias vertentes que podem ser utilizadas pelo atacante, criando assim espaço para a compreensão do que vem a seguir. Posteriormente, sem descurar o lado oposto, são apresentadas as alternativas e estudadas todas as evidências deixadas pelos rootkits, numa tentativa de as aproveitar para que o seu respectivo ataque possa ser prevenido no futuro.

Esta dissertação começa por enunciar várias técnicas de Rootkits vulgarmente utilizadas no sistema operativo Windows, com o objectivo de perceber onde e como é que estas podem ser aplicadas no sistema operativo Linux. As técnicas de "hooking" são tranversais a todo este trabalho e são um método poderoso tanto para quem ataca como para quem defende, daí a sua relevância nesta dissertação. Hooking não é mais do que uma técnica que permite interceptar o fluxo de execução de uma aplicação ou até mesmo do Kernel, permitindo a quem a utiliza, obter dados importantes ou modificar essa mesma informação.

É interessante notar os múltiplos pontos sensíveis do sistema operativo onde se podem instalar "Hooks". Tanto em modo utilizador como em modo privilegiado, existem várias zonas cruciais no sistema operativo que podem ser sujeitas a ligeiras alterações, originando um comportamento global do sistema diferente, que irá favorecer o atacante. Deste modo, o sistema irá fornecer informação falsa para uma aplicação que queira saber como se encontra o estado do sistema.

Felizmente, conhecendo o funcionamento destes programas subversivos podemos detectar a sua presença caso estes estejam instalados. Tipicamente, é sempre possível detectar vestígios de intrusão no sistema, mas dependendo do avanço tecnológico do Rootkit, pode ser extremamente difícil de o conseguir.

Este trabalho demonstra a eficácia e a perigosidade da técnica "Hardware Breakpoint Hooking", e apresenta um método de detecção e prevenção dessa mesma técnica (não infalível). Adicionalmente, são estudados os ataques que implicam modificação de código de aplicações em modo utilizador, aos quais

é apresentada uma ferramenta de rastreio de memória. Por fim, é apresentado um sistema de detecção de rootkits proactivo que inspecciona vários pontos fundamentais do sistema operativo, durante a sua operação.

Palavras Chave
Keywords

Palavras Chave

Segurança

Prevenção

Detecção

Sistema

Intrusão

Subversão

Keywords

Security

Prevention

Detection

System

Intrusion

Subversion

Index

List of Figures

List of Tables

1 Introduction

Your computer can be doing absolutely anything to anybody anywhere, and you'll never know it
– Rick Cook

This chapter acts as a prelude for this thesis beginning with a short motivational introduction, followed by a description of the goals of this work and finalizing with the roadmap of this thesis.

1.1 Motivation

The term rootkit originated in the mid-nineties. The idea was based on a set of tools (kit) whose purpose would be to subvert the system in order to change some of its original behavior. Usually these tools would maintain covert root access to a system and hide the intruder's presence. This is why the resulting term is rootkit.

A rootkit tries to remain undetected by using techniques to hide parts of the system to a User-land program. It may hide files on the hard drive, running processes, active network connections or any other element that may reveal unauthorized activity.

The complexity of a rootkit depends on the technique it uses to achieve its stealth to avoid detection. The application of these techniques requires deep knowledge of the operating system and advanced programming skills. Depending on how advanced the rootkit is, it is important to understand memory layout, kernel objects, file and network drivers, privilege levels; some of them are not documented.

Typically, a User-land rootkit is much simpler than a Kernel-land rootkit. On the other hand, entering the kernel is a whole new world of endless possibilities. For a security expert that knows the depths of the operating system, accessing the memory without constraints will make him find many different ways to subvert it in order to achieve stealthiness.

Rootkits are constantly evolving and they will become a serious threat in the future, if they aren't yet. Every year, new pioneering techniques are demonstrated at security conferences around the globe, and we are starting to see anti-rootkit technologies implanted in anti-virus solutions.

The latest advance has been in virtualization techniques (Microsoft, 2006), which by taking advantage of virtualization technology, rootkits manage to avoid mainstream detection algorithms as they

take the place of the virtual machine supervisor. This allows them to monitor and intercept all hardware calls initiated by the guest operating system. The main purpose of this thesis is to identify current rootkit technologies, and provide solutions (or mitigating techniques) for them, discussing the pros and cons.

1.2 Goals

This document pretends to explore some advanced hiding technologies used in rootkits along with solutions to detect and prevent these abuses.

Some hooking techniques used on windows operating system to intercept application function calls at runtime seem to be forgotten on Linux. This article intends to prove that they are possible and also deserve as much attention from the security perspective as the other known hiding techniques.

The system call invocation mechanism contains many weak places that rootkits can explore in order to subvert the original system's behavior. The system call table and the interrupt descriptor table belong to the most common attack vectors used from rootkits. This work intends also to detect and/or prevent these attacks with the final goal of improving the security of the Linux operating system.

1.3 Structure of the Document

Chapter 2 starts with the state of the art in what concerns to rootkit technologies; chapter 3 shows different methods for accessing the memory address space of a process, and ilustrates different ways of hooking a function call, terminating with solutions to test the integrity of the running applications; chapter 4 introduces the old-age art of system call redirection, making also reference to interrupt descriptor table redirection and presenting solutions, once again, for its detection and prevention.

Finally, this work ends in chapter 5 with a discussion of the results taken from this study along with future work and conclusions.

2 State of the Art

By playing the part of an attacker, we are always at an advantage. As the attacker we must think of only one thing that a defender didn't consider.

– Greg Hoglund and James Butler

2.1 Introduction

This section presents the state of the art in what concerns to rootkit technologies. To ease the understanding of the following chapters, this one gives particular relevance to windows operating system since part of this thesis resides in the exploration of existing windows rootkit techniques on Linux.

This chapter starts with a classification of malware followed by a brief introduction of the general inner working mechanisms of a rootkit; Next, some hiding techniques are uncovered and state of the art defenses are described for each hiding method; finally, a comparative study shows the strengths and weaknesses of some existing anti-rootkit software and the conclusions of this chapter.

2.2 General Malware Classification

The following classification was made by *Joanna Rutkowska*, a Polish security expert known for her research of stealth malware and contributions concerning improvement of Windows Vista security. Rutkowska's paper (Rutkowska, 2006) makes use of this taxonomy to explain different rootkit techniques and their respective counter measure. Before delving further into the details, it is important to understand this classification. Instead of classifying malware in categories such as Virus, Worms, Spyware, Backdoors, etc...; Rutkowska decided to create a ≪big picture≫ of malware which will certainly help creating our defenses against these threats. Figure 2.1 shows each type of malware.

Malware Type 0 Type 0 is malware which does not modify the system in any way, meaning that the other applications don't change their behavior. Malware inside this category is not considered a rootkit since it does not use any kind of system subversion technique to hide itself and remain undetected.

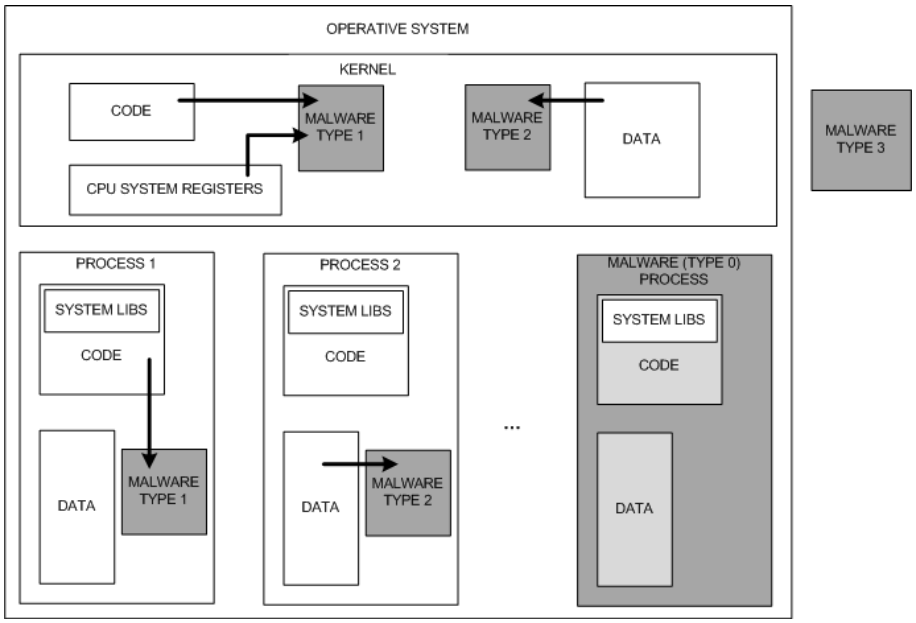

Figure 2.1: Malware classification

Malware Type 1 This type of malware modifies sections that are not generally changed by legit applications, such as code sections within executable files and BIOS code.

Since these areas should never change, a good approach for detecting this malware is in integrity checkers. Integrity checkers are covered later in 2.5.3.

Malware Type 2 Type 2 malware involves modification of regions that should be changed, typically configuration files, registry data and data sections of running processes and Kernel. The attackers challenge here is to identify dynamic regions of the system in order that he can take advantage of. Since these areas are expected to be modified over time, it is extremely hard to detect every system change that can be potentially harmful. Integrity checkers are unsuccessful in detecting this category of malware, instead, we need to identify all dynamic sections of running applications and Kernel, and understand them, in order to determine any potentially installed malware.

Despite being the perfect solution, this task is not practical to achieve. This would require the creation of an extensive list with all the sensitive dynamic places, together with their corresponding verification method.

However, it's not possible to determine all those sensitive places (not even by Microsoft). Software is constantly evolving, with regular upgrades being distributed to the public which provide new veiled methods for a rootkit to hide.

Malware Type 3 This type of malware is the latest advance in rootkit technology. This kind of rootkit, known as Hypervisor rootkit, does not change a single bit of the operating system. Instead, it

acts from the outside as a host machine making use of the virtualization features provided by the processor, intercepting all hardware calls made by the guest OS.

Detection of this technique is a heated discussion among nowadays security researchers. Most (if not all) methods presented detect the presence of a virtual machine but cannot tell if the virtual machine is legitimate, or if it is a rootkit. In the future most enterprise solutions will be based on virtualization technology, which is why these detection methods are not enough.

2.3 How do Rootkits work

2.3.1 Application level

Binary kits are a set of tools designed to replace, without authorization by the systems owner, the system binaries in order to change the original information flow from Kernel to user applications. Typically, the information concealed is the information revealing the presence of unauthorized activity, so the user will not suspect of an intrusion. These types of attacks are mostly used in Linux, where the code is open-source and the system tools are easy to alter. In later versions of Windows, system files are protected (for instance, Windows XP uses Windows File Protection), and thus, these tools are harder to be subverted.

2.3.2 Library level

Library level rootkits are those which intercept library calls, by hooking or replacing the libraries.

Library replacement is usually associated with library kits. Library kits attempt to replace system libraries in order to intercept important functions. For example, in Linux, the T0rn 8 rootkit (Miller, 2000) replaces the "libproc.a" library, which is used to find information from */proc* file system to get access to process information from the Kernel, which can be gathered by simple system tools like */bin/ps* and *top*. Another alternative is to modify the glibc/libc main system library in order to filter the information exchanged between user and Kernel-land.

Once again, in Windows, system libraries are system files, which are generally protected by the operating system, so it is more viable to make the library interception by entering the address space of the processes and hooking the desired functions. Hooking is covered later in 2.4.1.

2.3.3 Kernel Level

Kernel level rootkits use some way to get into Kernel space and directly add and/or replace code to the operating systems core. In Windows, most Kernel rootkits make use of device drivers, while under

Linux, they use Loadable Kernel Modules (LKM). LKM rootkits were for a long time the favorite approach for the attackers who wanted to have control at Kernel level in Linux systems. An LKM rootkit does not require much knowledge to be implemented, the addresses of the system calls are accessible via the sys_call_table array and it's easy to redirect system call functions to point to our code by making use of this array. However, since Kernel 2.6 that the sys_call_table array is not exported which means that an attacker will have additional work searching for a reference to it. The system owner can defeat LKM rootkits by simply disabling Kernel module loading support before compiling the Kernel. Nevertheless, it is not mandatory the use of these Kernel extensions to access the system core, as explained later in 2.4.3.

2.4 Hiding techniques

2.4.1 Hooking

One of the techniques a rootkit may employ to shift the execution flow of the operating system is hooking.

The main goal of hooking is to alter the information returned by the system, or provide the system with altered information from the calling application.

Due to system flexibility and backward compatibility, there are many ways a system can be hooked. Hooks can be installed in user land or in Kernel-land and they can be considered malware type 1 or 2, depending on which technique is used. All the methods require certain knowledge of the function being intercepted.

2.4.1.1 What to hook?

The potential interests of a rootkit can be the following: Hide files, registry information, processes, network connections, drivers, etc. All of this can be accomplished with hooking.

For instance, in Windows, a user land rootkit may want to hide itself by intercepting the correct system API call that reveals its presence.
A good target would be TaskMgr.exe, the windows task manager. What the rootkit would have to do was enter the process address space, and intercept the call to the *NtQuerySystemInformation* function in ntdll.dll, by hooking it and filtering the response from the system.
Ntdll.dll is a system support library which provides dispatch stubs to windows executive system services that will pass control to SSDT (system service dispatch table), in Kernel-land, where the actual work is performed.

The only problem arising from this method is that while the windows task manager gets subverted, any other application will be able to detect the rootkit executable, meaning the hook is only installed for the process TaskMgr.exe. For the hook to be system wide, there are two options: Hook all processes, which is not a very stealthy option, or go deeper into the system and install it at Kernel-land. To accomplish this, it is important to understand the execution flow of a process enumeration request to Kernel, find out where to hook, and how to hook.

It is possible to hook one of multiple functions to successfully hide our process, but if we are working in user-land, a good choice is *NtQuerySystemInformation*, since this is the deepest we can go before entering the Kernel. In Kernel-land, the system call *ZwQuerySystemInformation* in the SSDT table would be a good place to hook.

2.4.1.2 How to hook?

I will enumerate some hooking techniques focusing on Windows operating system, with a brief description for each one:

Inline Hooking This technique, also known as trampolining, patches the first bytes of the function to hook with a *jmp* instruction pointing to our function. Overwritten bytes are backed up, and executed inside our intercepting function in a different place, right before calling the original function (trampoline). We can change the information returned by the original function or provide different arguments to it. Microsoft's research group has released a library called Detours (Corporation, n.d.), which is a handy utility to ease the use of hooking.

Function pointer table hooking This technique outlines any hook based on dispatch table modification. The idea is to change a function pointer from a table, making it point to our function instead. Potential tables to place hooks are:

1. System Service Descriptor Table (SSDT) – The system call table. Due to its system wide impact, SSDT hooking is not only used in rootkits, but also in anti-rootkits. A system abuse can be predicted by analyzing what system calls are being used.
2. Interrupt Descriptor Table (IDT) – Allows interception of system interrupts. Since each processor has its own IDT, a rootkit will have to hook this table into all processors.
3. IRP tables – Device drivers communicate via IRPs (Input/Output Request Packets). Each packet type is handled by a proper function, which is saved in the IRP table. IRP table hooking aims to intercept driver requests such as reads, writes and queries. Since drivers are extremely low level, IRP hooking is a good place to hide a rootkit.

4. Import Address Table (IAT) – Table containing pointers to all functions inside each module, within a process.

Hardware Breakpoint Hook This is an unusual technique based in the use of debug registers. Intel x86 CPUs have special registers intended for debug use only. By storing special values into these registers, the CPU can execute an INT 1 (interrupt 1) instruction just before any read, write, or execute attempt to a specific memory location. Through the help of structured exception handler (SEH), a routine is specified to handle the exception raised by interrupt 1. Once in this routine, we can change the extended instruction pointer register (EIP) to point to our function, thus changing the natural execution flow of the thread. When inside our function, if we want to execute the original (hooked) function, the hardware breakpoint has to be removed before, and restored after the call, to avoid entering in an endless recursive loop.

This method slows down the execution speed of the process being hooked because of the interrupt cost. On the bright side (from the attackers perspective), there is not a single byte changed in the code, so the code sections remain untouched.

2.4.2 Direct Kernel Object Manipulation

Windows Kernel stores important information in Kernel objects for book-keeping and reporting. These objects are no more than Kernel structures, holding what the system believes exists on the system. Direct Kernel Object Manipulation (DKOM) works by taking advantage of this fact.

The correct way to access a Kernel object is by using the Object Manager. Object Manager eases the access to an object, such as creation, deletion and protection. DKOM accesses the object directly, bypassing the Object Manager and thus all access checks to the object.

A rootkit that uses DKOM will subvert these objects and force the system to believe it is in a different state. The strongest countermeasure of using DKOM relies on the fact that the attacker must know how the structure is built, and this requires hours of debugging. Successfully exploitation of DKOM can result in hiding processes, device drivers and network ports. There are other system changes the attacker may find useful, such as privilege elevation of a process.

This technique is considered malware type 2, and is extremely difficult to detect, since all the manipulated sections are dynamic.

2.4.3 Raw access to the physical memory

It is possible to access Kernel memory without the need of drivers (Windows) or Kernel modules (Linux) by making use of the physical memory device.

In Windows, the *\Device\PhysicalMemory* section object provides access to physical memory from the user mode, specifically from the Administrators group, however, in Windows XP 64-bit, Windows 2003 Server SP1, and Windows Vista, all user mode access to this device has been blocked.

For translation between virtual memory to physical memory, a good memory layout understanding is required, and it is important to keep in mind that the memory model varies between windows versions.

W32/FanBot.A@mm (Florio, 2005) is a well written worm that uses this technique to perform the DKOM technique. In what concerns to Linux, since nearly the beginning that some character devices, called */dev/mem* and */dev/kmem*, are available to privileged applications. Despite */dev/kmem* not being a physical address, it contains the whole Kernel memory which can be manipulated by a rootkit. SucKIT rootkit makes use of this device to intercept the system call table. Additionally, */dev/mem* device provides raw access to any physical page. Once again, it's important to understand the UNIX memory model.

2.4.4 Layered Filter Drivers

Another alternative for intercepting the natural execution flow of a Windows system is by using filter drivers. Layered filter drivers can extend the original features of a driver without having to rewrite it from scratch. More importantly, there is no need to understand the complexity of the hardware.

This technique is a flexible solution for extending driver functionality that can be used for good and evil. Many virus scanners attach a layered filter driver on the top of the file system driver, for scanning files as soon as they are accessed and before they are passed to the user land application or get executed.

On the other hand, a rootkit can take advantage of this for stealth purposes by intercepting and modifying the information coming in and out from the lower-level hardware, changing for instance: file data, file enumeration and network connections. A good example of a rootkit using this method is KLOG (Clandestiny, n.d.). KLOG is a smart approach for a keylogger: Works by attaching a filter driver to the keyboard driver, and monitoring all the keystrokes typed.

2.4.5 DLL/Code Injection

Dynamic-link library (DLL) injection works on any operating system supporting shared libraries, but we will focus on Windows. Despite not being seen as a rootkit, it can be used to drop a relatively stealthy peace of code running into another process. There are many ways to force a DLL to be loaded on a remote process and nowadays this is becoming a very common technique for hiding malware. This is why DLL Injection is getting attention from the security community and its detection and prevention is getting studied with great results.

Code injection has some advantages over DLL Injection since it does not insert a new module in the target process. Instead, it injects just the necessary executable code into the target process, and then starts a new thread at the beginning of the code. It is a stealthier technique, however, one should be careful since the imported function addresses will not be correctly resolved. This means we will have to manually load every DLL whose function we want to use with *LoadLibrary()*, and then get their respective address with *GetProcAddress()*. *LoadLibrary* and *GetProcAddress* function pointers are correct because they reside in *Kernel32.dll*, which has the same base address on all processes.

In DLL Injection we don't have this problem because every function address is loaded when the portable executable file (DLL) loads in the target process. Note that, in both techniques, a new thread does not have to be necessarily created, an existing thread can be hijacked, but this can be dangerous unless the attacker knows that the thread is safe to be hijacked. Note that once we are inside the target process's address space, we can hijack library calls with hooking.

There are several ways of injecting DLL/code into another process, most of them are easily detected by the majority of anti-virus proactive defense solutions.

2.5 Detecting and Preventing

At the beginning of 2005, host based detection and prevention systems (HIDS and HIPS) concerning rootkit technologies started to become popular, and many companies launched their products in a response to this threat: F-Secure, Kaspersky, Microsoft... etc.

First of all, it's important to clarify the difference between detection and prevention:
Detection assumes that the system may already be infected, and searches for signs of known techniques, while prevention analyzes the application's execution flow in order to avoid hostile activities, or checks what applications are about to run (anti-virus products). No matter which side we choose, both represent added value towards best-practices on host based security.

2.5.1 Signature based detection

Signature based detection works like a fingerprint. A sequence of bytes from a file is compared with another sequence of bytes which are known to belong to a malicious program. This concept is easy to understand, and is the way anti-virus products have been working for years. The weakness of this method is that it is ineffective against new and unknown malware. Besides, since this technique is usually employed on the filesystem, it can easily be eluded by a rootkit trick, for example, by hooking the filesystem driver.

For true effectiveness against a rootkit, Kernel memory should also be scanned for signatures (besides filesystem). Since polymorphic obfuscation is rarely a rootkit concern, a Kernel memory scan can reveal great results towards rootkit detection, no matter how advanced the rootkit is.

2.5.2 Heuristic based prevention

Also known as behavior based prevention, heuristic based prevention is becoming a popular approach for preventing rootkits and general malware, employed in anti-virus solutions. It works by analyzing execution paths, and determining through the heuristic, if a certain behavior is abusive. Comparing to signature based detection, this one has much more probability for false positives, but on the other hand, can be quite effective against new threats.

For example, to prevent injection of some kind into processes, Kaspersky Anti-Virus (*Kaspersky Lab*, n.d.) hooks some functions in the system call table which must be used for general injection techniques. Once these functions are called, Kaspersky analyzes the arguments, checks if they can lead to an unwanted operation, and alerts the user indicating what program is trying to be intrusive and what action it is trying to do. Then the user may choose to allow, deny, or terminate the suspected application.

2.5.3 Integrity Check detection

Integrity checking is the most effective detection approach for malware type 1.

The main purpose of integrity checking is to verify if certain regions of filesystem and memory are identical to a known trusted baseline. Digital signatures can make this task easier. Of course this baseline should not be forged or easily bypassed, and this is where many integrity checkers fail. Integrity checking will surely avoid some sophisticated rootkit techniques from taking place. The analyzed regions should be all sections susceptible to subversion, which means code sections in memory (Kernel-land and User-land memory), and files, usually system files. The best tools that use this method are Tripwire and System Virginity Checker, which are covered in the next chapter.

2.5.4 Crossview based detection

This is a very popular rootkit detection approach employed by many anti-rootkit software. Crossview based detection (X-View) compares a “high level” view with a “low level” view of the system. If the “high level” view does not show the same contents as the “low level”, this means something is being hidden, and the system was subverted. X-View can be used to detect hidden files, processes, registry keys, network connections, etc.

The “high level” view of the system is obtained using the common API functions, provided by the operating system. The “low level” view should be obtained without being affected by any rootkit trick and this is certainly the hardest part when designing X-View. Obtaining the “low level” view depends on what we are scanning. For instance, to implement X-View in a file system scan, a good approach for obtaining the “low level” view would be to access the raw disk file sectors, and parsing them according to the NTFS layout. Nevertheless, a smart rootkit would hook the function that reads the file sectors, and provide to the anti-rootkit, again, the wrong information about the file system. This means that a reliable X-View anti-rootkit can be very hard to implement.

2.5.5 Detecting DKOM

There is no general method for detecting DKOM attack, and since Kernel objects reside in dynamic sections, integrity checkers do not help here. The attacked Kernel object must be studied in order to understand how we can verify the inconsistency. For example, process hiding works in the following way: Every process has an associated structure called EPROCESS, which is a Kernel object. Windows Kernel maintains a doubly linked list of all EPROCESS structures belonging to every process running in the system. If a user-mode application requests a list of the running processes, a Kernel function traverses this list and returns process information to the requesting application. It’s easy to understand that if we remove an entry from this doubly linked list, the process will be invisible to user-land applications. But despite the process being hidden, we want it to run, so the threads (ETHREAD structure) belonging to our hidden process are still in the waiting dispatcher list of the NT scheduler. This means there are still evidences we can hardly cover, because if we remove the threads from the dispatcher list our hidden process will stop running. Based on this fact, Joanna Rutkowska created her tool called Klister. Klister runs the list of threads in the dispatcher list, and searches for any associated process not present in the doubly linked list of EPROCESS’s.

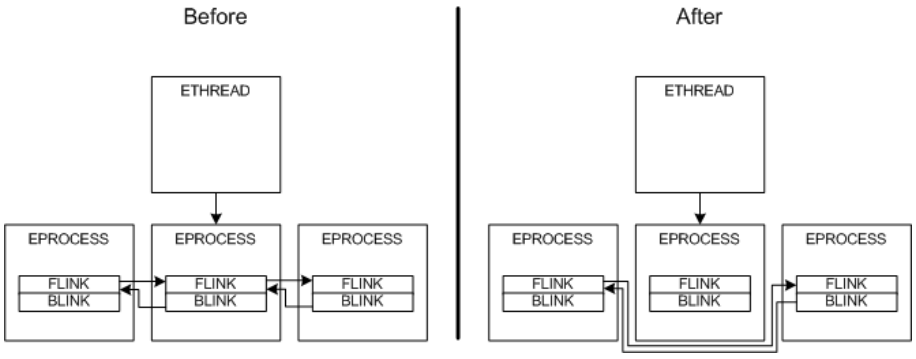

Figure 2.2: Illustration of a DKOM attack before and after hiding a process.

2.5.6 Detecting Hooks

Hooking is a very popular technique for rootkits, but it's hard to be sure when a certain hook is legitimate. Microsoft uses a feature called "hot patching" which allows the patching of system binaries even if they are running in executable memory. This technique is particularly handy for reducing the number of reboots required when updating system components.

Inline Hooks Inline hooks cause code modification, which is easily detected by integrity scanners. Alternatively, if we find a *jmp* instruction in the beginning of a function, we are in the presence of an inline hook. However, integrity scan is more reliable because an attacker can find obfuscated techniques to execute a jump instruction not in the beginning of the hooked function but some instructions later (for example, using NOPs).

Table Hooks In what concerns to function table redirection, these tables generally point to some place where we can confirm it is legitimate. For example, SSDT functions always point to ntkrnlpa.exe module. An anti-virus may use this hooking approach to implement a heuristic scanner by changing some function pointers to its own module (for instance, Kaspersky proactive defense mechanism redirects some system service functions to point to *kl1.sys* module). If these tables point to some unknown and suspicious location, there is a strong indication that something might be wrong.

Hardware Breakpoint Hooks Since hardware breakpoint hooks use a debugging technique, we must first confirm that a program is not being legitimately debugged. If not, we need to read the debug registers for every thread context within a process and check if they are properly configured to call interrupt 1 when an address is reached (executed, read or written). Care should be taken when accessing the thread's context: Using a high level API call such as GetThreadContext is dangerous because this function may also be hijacked. The perfect solution would be to access the thread's context directly from Kernel memory by making use of a driver.

PatchFinder 2 Technique Joanna Rutkowska created a tool called Patchfinder 2 (Rutkowska, 2004) which uses a sophisticated detection technique based on runtime execution path profiling. This method consists in the idea that a rootkit adds new code somewhere near a routine's execution path, in order to subvert it. Based on this fact, the number of instructions executed before and after a certain routine is hooked will be significantly different (much greater in the hooked routine than in the clean one). Patchfinder 2 takes advantage of the single-step feature provided by x86 processors, where an Interrupt Service Routine (ISR) can be specified to execute after each instruction is consumed. The only thing this ISR does is to increment the variable that performs the counting. Rutkowska also suggests a smart improvement for future work which consists in analyzing the code flow instead of counting only the number of instructions. It's important to

realize that this technique is prone to false positives: the number of executed instructions during a system call may fluctuate, but have a stable peak. To reduce false positives, Patchfinder 2 creates a histogram for calculating the peak, and watches for shifts on this peak that may indicate a hook was installed in the meanwhile.

2.6 Anti-Rootkit Software Analysis

The following table shows the features implemented by eight anti-rootkit programs.

Table 2.1: Detection software - Comparative study

		F-Secure Blacklight	SysInternals Rootkit Revealer	Icesword	GMER	Rootkit Hook Analyzer	System Virginity Verifier	Mcafee Rootkit Detective	Rootkit Unhooker
Filter Drivers					X				
DKOM (Proc Hide)		X		X					
Hooking	SSDT			X	X	X	X	X	X
	IDT				X	X	X		
	SYSENTER				X	X	X		X
	Inline			X	X	X	X		X
	IAT			X	X	X	X	X	X
	IRP				X		X	X	
	HWBP								
X-View		X	X		X			X	X
Integrity Check							X		

Crossview and integrity checking are two detection mechanisms that cannot tell which rootkit technique was employed. They can just tell that something is being hidden, or some part of memory is not as it should be, respectively. For example, a rootkit may use an inline hook to intercept a function. Crossview detection will only show the alert if the hook is hiding some information that can be compared with a crossview analysis.

As we can see through this table, hardware breakpoint hook detection is not much of a concern nowadays, nevertheless, it is as powerful as any other type of hook.
Layered filter drivers are easy to detect, but hard to determine if they are malware. In this case, a deep crossview analysis may be helpful (if the filter driver is hiding something important such as files, processes or network connections). Alternatively, a memory signature scan can also be helpful if the rootkit's signature is already known.

2.6.1 System Virginity Verifier

SVV (Rutkowska, n.d.) is an integrity checking tool created by Joanna Rutkowska for windows 2000/XP/2003 that verifies in-memory code sections along with many other read-only system com-

ponents which are usually altered by various stealth malware (type 1) and supports disinfection.
SVV also takes additional care with false positives, because many tools use techniques which involve code modification, which can be considered rootkit (for instance, system monitoring tools and firewalls).

2.6.2 Tripwire

Tripwire (*Tripwire*, n.d.) is another integrity verification tool available for all UNIX systems, whose purpose is to identify and alert on file changes. On the first run, tripwire scans the file system and stores in a database, cryptographic hashes for each file. On a later date, new hashes are obtained for each file and these are compared against the information previously stored in the database.

2.6.3 VICE

VICE (Butler, 2004) is a freeware tool written by James Butler designed to detect most types of hooks. It works by installing a device driver which scans both User-land and Kernel-land for hooks. Supported hooks are: Inline, IAT, SSDT and IRP. The biggest problem about VICE, as with many hooking detection tools, is in its large number of false positives, from legitimate windows hooks.

2.6.4 Hijacking anti-rootkit software

It is not hard to implement a rootkit with the ability to bypass a known anti-rootkit system. By knowing details of the anti-rootkit software, i.e., executable file name, description, properties or other information about the application, a rootkit can turn off its defenses and show a clean view of the system, only to the scanning application. Even worse, a smarter rootkit may hook certain functions or patch code sections of the anti-rootkit software, and make it completely sterile.

To mitigate this threat, a scanning application may perform the scanning with the help of another process through the injection of a function or a DLL. With DLL injection, it's important to hide the injected module because a rootkit could also scan the Process Environment Block structure (PEB) of all processes and traverse the doubly linked list of modules in order to find the anti-rootkit module.

Trojan.Linkoptimizer is a rootkit that attacks anti-rootkits. However, this application chooses to prevent the execution of known anti-malware programs, which will certainly alert the user that the system may be infected.

2.7 Summary

All the presented detection techniques have their strengths and weaknesses, but where one fails, the other succeeds. This chapter has shown the general techniques for Windows rootkits along with the typical approaches for detecting them, not forgetting to reference the Linux operating system when needed. Most of the techniques covered here have been in use for years, and when a new hiding method becomes public, defenses against it are immediately studied and implemented in anti-rootkit software. It is interesting to watch this as an endless game, where both players are always relatively balanced. In fact, no one will ever win or lose. A comparative study at the end of the chapter shows the top anti-rootkits, along with their differences.

3 User-Land

No detection algorithm is complete of foolproof. The art of detection is just that - an art.

– Greg Hoglund and James Butler

3.1 Introduction

The previous chapter contained the explanation of three general methods used for hooking a function on Windows. Although not being widely used on Linux rootkits, these techniques also deserve attention since they can be used to achieve the same objectives as in Windows. The purpose of this chapter is precisely to explore these attacks on Linux and study what can be done to detect and prevent these threats.

By having access to the address space of a process, the execution flow can be manipulated in many ways. The way the application sees the information and its behavior can be changed without users consent. Since hooking is based in memory manipulation, this chapter begins by introducing three methods that can be used to access the memory of a process. Once this is introduced, room is created to understand how the implementations of hooking can be done.

Lets look at the following motivational scenario: An attacker penetrates a system where tripwire is running, he is able to elevate his privileges. The attacker pretends to discover the password of the users of that machine but he does not want to waste undetermined time trying to crack the shadows in */etc/shadow*, so he uses another approach: Access the ssh daemon address space at runtime and hook the function *auth_password* which authenticates the user's password at login. When the attacker hooks this function, he can access the arguments and steal the password. He can also verify the return value of the function which returns zero on failure, and different from zero on success so he can avoid stealing wrong passwords. The passwords can then be saved to a place where tripwire does not monitor such as */tmp*. The attacker just has to wait for the trojaned version of the ssh daemon to do its job and check the file in */tmp* for passwords from time to time.

The figure 3.1 does not focus any particular hook but serves to show the potential of one. In this case, the hook is set not to hide information that reveals the presence of the attacker, but to steal information that the attacker should not have access to. Tripwire will be quiet and will not report anything

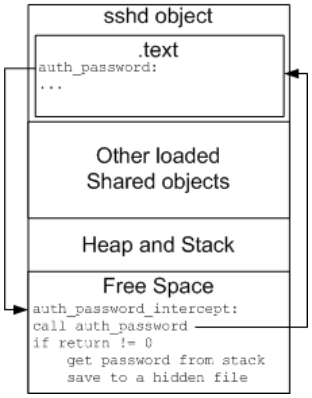

Figure 3.1: Hooking the ssh daemon at runtime

since the disk image of the ssh daemon will be untouched.

3.2 Accessing the address space of a process

When we talk about hooking, it is implicit that there must be any means to access the process memory. In Windows operating systems, microsoft provides a strong API for manipulation of other processes. Lets take a look at some examples:

VirtualAlloc Allocate space on a specific process.

VirtualProtect Change the protection of a range of commited pages in the virtual address space.

WriteProcessMemory Write data to a certain region of memory in a specified process.

ReadProcessMemory The same as above, but for reading.

CreateRemoteThread Creates a thread in a specific process starting at an address. Linux does not have a similar function.

Windows User-land rootkits take advantage of these (and other) API functions to achieve stealthiness (MSDN, 1999). Some proactive defense systems intercept the execution flow triggered by these functions somewhere in the system (generally the system calls are ultimately invoked), and search for known behavior patterns in the sequence of the calls and input arguments in order to detect abuses. In Linux, things are different. This section shows three ways to access the address space of processes:

Using LD_PRELOAD By using this environment variable, it is possible to instruct the loader to load additional libraries before all the others that were defined at compile time. Note that this access

cannot be made when an application is already running because it takes place when the dynamic loader does its job, when the process is executed. In fact, since the access is made from a shared object, we can access the memory of the process because we actually make part of it. For the attacker, this technique has the disadvantage of modifying an environment variable, possibly revealing an unwanted shared library being loaded to a careful administrator.

Using ptrace system call The ptrace (*ptrace Linux Man Page*, 1999) system call is widely used by debuggers to examine and control the execution flow of an application. This can be achieved by changing the process registers and directly reading and modifying the address space. With this function call we can hook a function and intercept important information at any moment while the process is running, contrary to the LD_PRELOAD technique, which only works when the process is run. This system call, in a way, is equivalent to some functions seen above in the API for windows.

Direct access to the process memory from a Kernel module On Intel x86 CPU, there is a special register called *CR3* that specifies a base address of a page table in physical memory. Each process has its own entry in the page-table directory, so when accessing the process memory from a Kernel module, one has to be sure that the CR3 register is pointing to the correct page-table entry. Since the CR3 register only changes when the current process changes, this technique can be hard to implement. Nevertheless, there are tricks that can make this easier which consist in waiting for the target process to be the "current". For example, if the Kernel module is intercepting a system call, it can check to see if the current process is the target and then access its memory.

3.3 Hooking techniques

This section presents three techniques for hooking a function in a process. The first one is inline hooking, primarily introduced by Microsoft Detours (Corporation, n.d.), and we will see how it can be applied in Linux. The second is PLT injection, presented by the security expert Silvio Cesare which consists in function pointer table modification, and finally, hardware breakpoint hooking, which has the advantage of not changing read-only memory segments nor function pointer tables.

3.3.1 Function hooking via code section modification

Under Linux, inline hooking may be the simplest way to hook a function. As described in the state of the art chapter, inline hooking works by replacing the first bytes of a function with an unconditional jump, forcing the instruction pointer to jump to the hijacking function.

Figure 3.2 shows the bytes before the replacement in red. The `HijackFunction` can control the input and/or output data flow of the function which is being intercepted. When the `Function` is

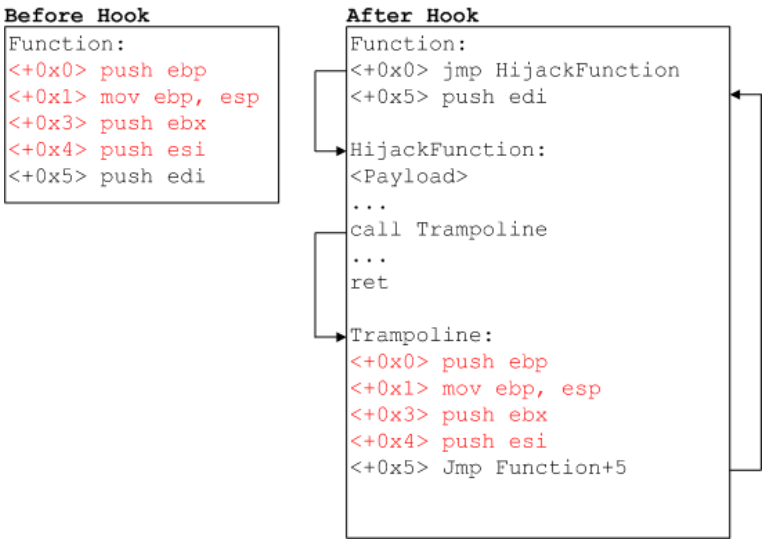

Figure 3.2: Inline function hooking

invoked, the new bytes force the instruction pointer to jump to the `HijackFunction`. Inside this function, the attacker has control over the arguments and is able to modify them. It is important not to forget to call the `Function` using the trampoline, which is a function that executes the old bytes that were replaced and then jumps to the following instruction, keeping the call to the function, consistent. The returning value can also be modified. Note that this hook implies the modification of read-only segments of memory, more precisely, *.text* sections. Before writing to memory, it is important to change the allowable accesses of the region using the function `mprotect` in order to avoid a segmentation fault for writing to non-writable position in memory. The memory segment should be made writable and after the byte code modification, the permissions of the segment should be restored.

It is easy to understand that one way to detect this attack is by searching the beginning of all functions for a `jmp` opcode. This method can be safe from false positives because *gcc* compiler never puts a `jmp` in such place. In Windows, there are some anti-rootkit software that run all the functions in memory and search for a `jmp` opcode. This detection scheme is unreliable because the attacker may not put the `jmp` opcode right at the beginning, but instead, after some useless instructions, just for the sake of not being detected this way.

We could use `nop` opcodes to achieve the same result but it would ease the job of a rootkit scanner because it wouldn't also be stealth to put such opcodes at the beginning of hooked functions. In the figure 3.3 the code before the `jmp` instruction does not do anything. The only usefullness of it is to avoid putting the jump at the beginning, this way, deviating the attention from the rootkit detectors. A rootkit detector that previously knows a rootkit that applies this technique, could use this code as a signature to detect unwanted hooks.

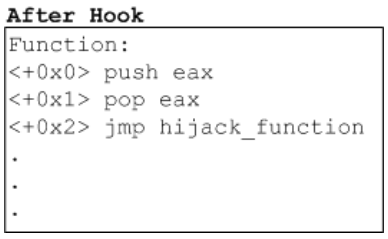

Figure 3.3: Inline function hooking (variant)

Since inline hooking relies in code section modification, and because these sections are read-only, a memory integrity check should be enough to detect that something might be wrong, as explained in section 3.4.1.

3.3.2 PLT Injection

PLT Injection is a hooking technique by *Silvio Cesare* similar to windows *import address table hooking* outlined in 3 (state of the art), except that this one only works for functions inside shared libraries (besides having a different file format, of course). Hooking techniques relying on function pointer table modification are not as accurate as inline hooking because the application may load the library itself using library APIs.

PLT stands for *Procedure Linkage Table* which is a section of the elf file (*ELF Specification*, 1995) format intended to redirect position-independent function calls to absolute locations. Function calls between executables and shared objects cannot be resolved in compile time because the compiler is unable to guess what will be every shared object function's offset inside the process's memory.

For every shared function, the PLT table redirects the flow to the PLT(GOT) table which does not contain the actual function addresses when the application starts. The first call to a shared function will jump to the dynamic linker which will resolve the real address of the function and replace it in the PLT(GOT) table, for the subsequent calls. This method of resolving the requested symbol when it is first referenced is called lazy loading and generally improves overall application performance, because the dynamic linking process does not cause overhead with unused symbols.

The idea behind this method is to replace the function pointer in the *.got.plt* table of the elf executable belonging to the process that contains the function to be intercepted, so that when the application invokes the desired function, the instruction pointer travels to our hooking function instead of the original one.

Despite this technique being old - almost ten years to be precise - it still can be used because the ELF file format is unchanged. On windows, IAT hooking is widely used on many User-mode rootkits, but

surprisingly for Linux, I did not find any rootkits taking advantage of PLT Injection. Instead, I found game cheats using this method to take control over the game.

Since this work does not go much deeper on this topic, I suggest reading the *Silvio Cesare's* article (Cesare, 1999) for those who are interested in the details.

3.3.3 Hardware breakpoint hooking

Once again, this topic has also been covered in the state of the art chapter 2.4.1.2, but here we will go deep into details and study how it can be applied on Linux x86 architecture.

This hooking method relies in the manipulation of debug registers. On the x86 architecture, there are eight debug registers available, from DR0 to DR7 which can only be modified in privilege level zero. The modification of these registers in another privilege level causes a general protection exception. The following table outlines the purpose of each debug register.

Table 3.1: Description of the debug registers

DR0 - DR3	The four breakpoints. These registers store the linear address of at most four breakpoints.
DR4	Reserved.
DR5	Reserved.
DR6	Helps the debugger determine which of the debug conditions have occurred.
DR7	Enable or disable breakpoints (from DR0 to DR3) and specify the debug conditions.

Note that the DR6 and the DR7 register have a specific format that must be considered when assigned. I suggest reading the INTEL manual (Intel, 2008) to understand the flags and fields of each one. The proof of concept of the following implementation is in the Appendix A.

First, we need a way to get inside a process address space. In my proof of concept, I used the LD_PRELOAD environment variable to ease this task. To the LD_PRELOAD variable was added a shared library which will be controlling the hook. Libraries should export initialization and cleanup routines using the `__attribute__((constructor))` and `__attribute__((destructor))` function attributes, respectively, so when the shared library loads, the constructor is called and here we can invoke the `fork` function, letting the main program run. This way we can get a process for us and attach to the child using ptrace's argument `PTRACE_ATTACH`. It is necessary to have a separate process because we need to control the execution flow of the target process with debugging mechanisms. To manipulate the debug registers of the target process we can use, once again, the `ptrace` system call. When the Kernel interrupts the execution to give processor time to another process, it saves the debug registers in the user segment of the application's memory. With the PTRACE_PEEKUSER and PTRACE_POKEUSER `ptrace` arguments, it is possible to read and write these registers, respectively.

We are going to need a function to perform the interception, in the memory of the target process. Because we used the LD_PRELOAD technique, the functions in our library object will exist in the memory of the target process, so we can take advantage of this and create a function for this purpose.

Nevertheless, it is also possible to do this without using the LD_PRELOAD variable at all and injecting the interception code somewhere in free space of the target process, using ptrace's argument PTRACE_POKEDATA. Despite being harder to implement, this one brings the advantage of hooking the function at any time during the execution of the process.

Finaly, we need the address of the function we want to intercept. We can find it with the help of GDB (*The GNU Project Debugger*, 2008) and then hard code the value in the code, or else we may use a more elegant way and find the symbol's address by parsing the ELF file on disk and locating the address in the symbol table.

Right after attaching to the target process using `ptrace`, we set the hardware breakpoint. Setting the breakpoint takes two steps: (1) Assign the address of the breakpoint to one of the first four debug registers (DR0 to DR3). (2) Change DR7 register according to specification in order to enable the breakpoint, so the CPU knows which registers are enabled and which debug conditions are defined - stop on execution, in this case.

To perform this hook, another system call is required. The attacker needs to know when the process stops due to the breakpoint, so here we need to get familiar with `sys_wait4` system call. Lets get into the details of this system call by taking a look at the following functions:

```
pid_t sys_wait4(pid_t pid, int *status, int options, struct rusage *rusage);
pid_t wait(int *status);
```

The `pid` argument is the pid of the child that the parent process is waiting on. `status` is a pointer to an integer that the Kernel fills with information revealing why the child process stopped. The `options` argument defines different behaviours to the system call and the last one, the `rusage`, provide resource usage information about the child. These last two arguments are not relevant for this study. The `wait` function is a libc wrapper function that simplifies the call to `sys_wait4`. The call `wait(&status)` is equivalent to:

```
sys_wait4(-1, &status, 0, NULL);
```

The value of the `pid` being `-1`, means that the parent is waiting for any child process to stop. Using the following macros to analyze the `status` variable, we can understand the reason why the process stopped:

`WIFSTOPPED(status)` - returns true if the child process stopped by delivery of a signal.
`WSTOPSIG(status)` - returns the number of the signal which caused the child to stop.

Note that there are more macros available for other purposes, take a look at the man pages (*waitpid system call*, 1997) for more information. When `wait` returns, it's important to verify two things to make sure that the traced process stopped due to a hardware breakpoint: (1) status argument indicates that the process received a SIGTRAP. (2) DR6 register of the traced process indicates that the breakpoint occurred.
The following steps illustrate the implementation of a hardware breakpoint hook:

1. Attach ptrace to the process with PTRACE_ATTACH argument.
2. Set one of the DR0 - DR3 registers to the address of the function we want to intercept.
3. Assign register DR7 according to specification, enabling the breakpoint defined in step 1 to break on execution.
4. Continue the execution of the traced process until the `wait` function returns and reports that the traced process stopped because of a SIGTRAP. At this moment, the breakpoint was reached meaning that the instruction pointer is pointing to the beginning of the function.
5. Using `ptrace` (PTRACE_SETREGS argument), write to the instruction pointer (EIP) register the address of the function that will be intercepted.
6. Continue the execution of the traced process

After step 6, the instruction pointer will continue executing the code in the hijacking function, but if we want to call the original one, we have to be careful because the breakpoint is still set. One may think about disabling the breakpoint while executing the intercepting function, but we cannot be sure when it is finished because it runs on a separate process. In this case, there can be used one of two alternatives:

- Use `ptrace` with PTRACE_SINGLESTEP on every "even" break, disabling the breakpoint before, and enabling it after. An "odd" break will occur when some function decides to call the target function. The "even" breaks will occur when the intercepting function invokes the target function.
- Change the DR register containing the breakpoint address to the address of the second instruction of the function to be intercepted, and when a break occurs, switch the it back to the previous one (the first instruction).

The code in the Appendix A uses the first alternative. The figure 3.4 clarifies the inner-workings of this hook.

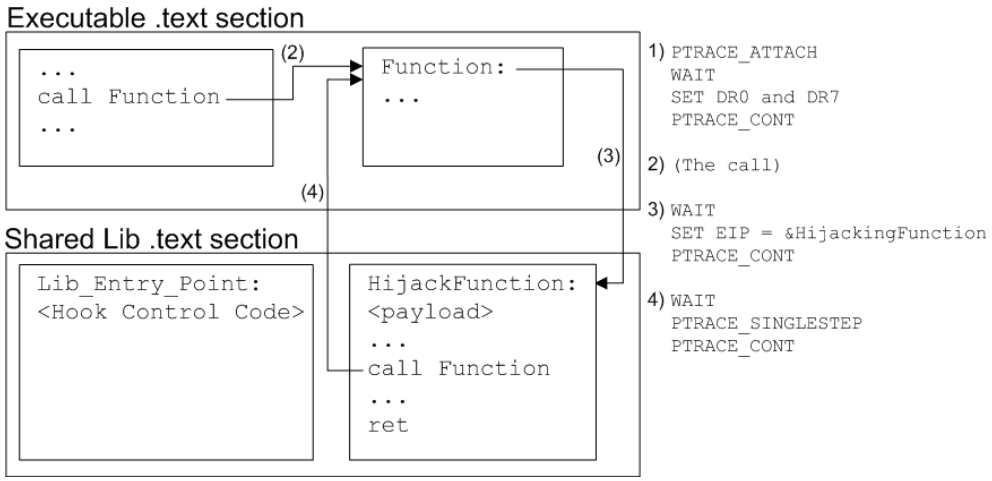

Figure 3.4: Ilustration of the Hardware Breakpoint Hooking Technique

3.4 Defenses

This section shows the implementation of two detection mechanisms each one for a specific hooking method. The first one is designed to detect inline hooking attacks. The goal is to build a program that scans the code sections in both memory segments and in the disk file (not only for the executable but also for all the dynamically loaded libraries), and compare if these two versions are identical. The second mechanism aims to detect and prevent hardware breakpoint hooking on-the-fly with the help of a Linux Kernel module.

3.4.1 Detecting program byte code changes

Integrity checking in physical storage is well known because of tools such as Tripwire (*Tripwire*, n.d.). These tools are extremely useful when it comes to rootkit detection because many malware replaces essential system tools of the operating system directly in the hard disk. On the other hand, to achieve its purpose, malware must be present in memory segments and not necessarily in disk. The disk modification is only for a matter of persistence and also to make the installation of the rootkit simpler, but it's not mandatory. On Windows, today, there are several anti-rootkit applications that verify the integrity of the memory of running applications. As said earlier, searching for a *jmp* opcode at the beginning of each function is unreliable because the attacker may not put a *jmp* precisely at the beginning of the function. Code sections should remain unmodified until the program terminates which means that there is another method for detecting inline hooks - and other types of memory incongruity - which is: checking the integrity of process code sections. These defenses seem to be much of a concern on Windows operating system, but not on Linux, probably because this type of attacks, regardless of being possible,

are rarely used. It's important to keep in mind that in a running application, normally there are several segments that contain code sections because the executable has is own code section, but the other shared libraries - also known as dynamic shared objects (DSO) - also have their own.

Next follows my implementation of a memory integrity scanner for Linux. Note that the tool I present here only scans for code sections (malware type 1, see section 2.2), which means that a rootkit may also find ways to hide information by changing other sections in the memory segments.

This implementation can be resumed as follows: For each running process, compare the *.text* (code) section present in memory with the version inside the executable on disk, do the same for each loaded shared object within the same process. If we are testing the integrity of something, we need a trusted baseline to compare with. Actually, we can't say that the disk image can be "trusted", because the attacker is able to change it unless there is a disk integrity checker present. So, to bring this application to a real scenario, there should be a database with checksums of all important processes and libraries, as tripwire does, but for this demonstration we will consider that the disk image is trusted.

3.4.1.1 Understanding memory mapping

If we want to find where a determined section of the application is inside a memory segment, it is important to understand how the Kernel maps an executable and their respective libraries in memory.

There are three types of ELF objects (Haungs, 1998):

- Relocatable file - an object file that contains compiled code suitable for linking with other object files to build an executable or a shared object file. These files have a '.o' extension.
- Executable file - Holds the program code after being linked with all its object files and is ready for execution. Has no extension.
- Shared object file - Also known as dynamic shared objects (DSO) or dynamic libraries, these objects can be used in two ways: (1) The linker combine this object with other shared object or relocatable files to produce another object file. (2) The shared object is linked at runtime with the application (and not at compile time).

When the Kernel loads the executable into memory, it has to know where to copy the executable, which parts it needs to copy and how to organize them in memory. The same must be done for each shared object that the executable requires to run properly. The ELF structure contains two important tables: The program header table and the section header table.

First, lets look at the section header table. The *readelf* tool shows this information:

```
readelf -S /bin/cat
There are 27 section headers, starting at offset 0x6678:
Section Headers:
[Nr] Name      Type      Addr        Off       Size    ES Flg Lk Inf Al
...
[11] .init     PROGBITS  080489dc    0009dc    000030  00 AX  0  0   4
[12] .plt      PROGBITS  08048a0c    000a0c    0002b0  04 AX  0  0   4
[13] .text     PROGBITS  08048cc0    000cc0    0048dc  00 AX  0  0   16
[14] .fini     PROGBITS  0804d59c    00559c    00001c  00 AX  0  0   4
[15] .rodata   PROGBITS  0804d5c0    0055c0    000dfc  00 A   0  0   32
...
```

The previous example shows the section header table of the cat command. Each section contains useful information when linking the program: program's code, data, initialized and uninitialized variables relocation information and other. When the program is executed, the dynamic loader does not look for this table, but instead it reads the program header table. The program header table contains information about segments of memory which are virtually contiguous page frames that the Kernel allocates for the memory of the application. These segments, known as VMA (virtual memory areas), are made of one or more sections so that the Kernel knows which sections belong to each VMA. The following image shows the parallel between the sections and memory segments.

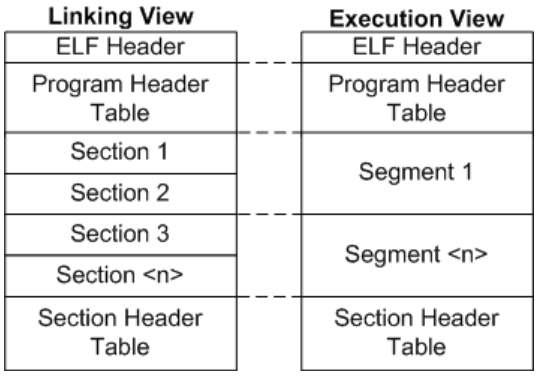

Figure 3.5: ELF file structure

The figure 3.5 taken from ELF specification (*ELF Specification*, 1995) shows on the left (Linking View) the elf file structure as the linker sees it and on the right (Execution View), how the Kernel sees an application and how it maps the segments in memory with the help of the program header table.

3.4.1.2 Using /proc/<pid>/maps

The */proc* directory contains virtual files which provide useful system information about the running Linux Kernel. Because the */proc* is not a real file system, no storage space is consumed but only a limited amount of memory. Nowadays, this pseudo file system is getting more attention from rootkit authors

and can also be subverted, not only to change the way the applications view the state of the Kernel but also to guarantee that an unprivileged user can regain unauthorized system control. Assuming the */proc* device is clean and we can trust it, we can view the memory mapping of an executable and respective library files by accessing the */proc/<process_pid>/maps* file.

```
cat /proc/self/maps
08048000-0804f000 r-xp 00000000 08:01 155058 /bin/cat
0804f000-08050000 rw-p 00006000 08:01 155058 /bin/cat
08050000-08071000 rw-p 08050000 00:00 0      [heap]
b7dd1000-b7dd2000 rw-p b7dd1000 00:00 0
b7dd2000-b7f1b000 r-xp 00000000 08:01 16765  /lib/tls/i686/cmov/libc-2.7.so
b7f1b000-b7f1c000 r--p 00149000 08:01 16765  /lib/tls/i686/cmov/libc-2.7.so
b7f1c000-b7f1e000 rw-p 0014a000 08:01 16765  /lib/tls/i686/cmov/libc-2.7.so
b7f1e000-b7f21000 rw-p b7f1e000 00:00 0
b7f2d000-b7f2f000 rw-p b7f2d000 00:00 0
b7f2f000-b7f30000 r-xp b7f2f000 00:00 0      [vdso]
b7f30000-b7f4a000 r-xp 00000000 08:01 293779 /lib/ld-2.7.so
b7f4a000-b7f4c000 rw-p 00019000 08:01 293779 /lib/ld-2.7.so
bfaee000-bfb03000 rw-p bffeb000 00:00 0      [stack]
```

Here, *cat* prints its own memory map. This file contains much of the information needed. We can see the information for each segment of memory, in particular, each starting and ending addresses, their access permissions and the path of the file from which the memory was mapped. We are able to see where the heap and the stack are located. [vdso] stands for *virtual dynamic shared object* and it refers to a virtual shared object exposed by the Kernel on all processes. Unless any application has changed it, segments with *"r-xp"* permissions are not writable and therefore should contain code sections.

3.4.1.3 Finding *.text* section offset in memory

Now lets take a look at the program header table. All the information we need to determine the offset and the size of the *.text* section in the object file is inside the respective ELF's section header table.

```
readelf -l /bin/cat
Elf file type is EXEC (Executable file)
Entry point 0x8048cc0
There are 7 program headers, starting at offset 52

Program Headers:
Type      Offset    VirtAddr   PhysAddr   FileSiz MemSize Flg Align
PHDR      0x000034  0x08048034 0x08048034 0x000e0 0x000e0 R-E 0x4
INTERP    0x000114  0x08048114 0x08048114 0x00013 0x00013 R-- 0x1
[Requesting program interpreter:  /lib/ld-linux.so.2]
LOAD      0x000000  0x08048000 0x08048000 0x063c0 0x063c0 R-E 0x1000
LOAD      0x0063c0  0x0804f3c0 0x0804f3c0 0x001dc 0x00364 RW- 0x1000
DYNAMIC   0x0063d4  0x0804f3d4 0x0804f3d4 0x000d0 0x000d0 RW- 0x4
```

```
NOTE      0x000128  0x08048128 0x08048128 0x00020 0x00020 R-- 0x4
GNU_STACK 0x000000  0x00000000 0x00000000 0x00000 0x00000 RW- 0x4

Section to Segment mapping:
Segment Sections...
00
01        .interp
02        .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr
.gnu.version .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini
.rodata .eh_frame
03        .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
04        .dynamic
05        .note.ABI-tag
```

There are many types of segments but the one that is important here is the `LOAD` type. The `LOAD` type means that the Kernel must read at `Offset` and copy all bytes to a memory segment starting at `VirtAddr` until it reaches `Offset` + `FileSiz`. This is valid not only for executable objects but also for shared libraries. The first `LOAD` is intended to copy all the read-only and executable memory. This segment contains the `.text` section which is the one we want to find in memory. The second `LOAD` segment contains all the dynamic data, and therefore, is readable and writable. For the example of the `cat` command, since the Kernel is supposed to copy all the bytes since the beginning of the file (`Offset` = 0x000000) until 0x063c0 (`FileSiz`), which is an address far beyond the ending offset of the code section (see the section table), we can conclude that this `LOAD` segment contains the .text section. Since there is only one segment that is simultaneously read-only and executable, this means that a 'r-x' segment in the maps file will contain the code section of an object.

Now we only need to know where exactly the code section begins and ends in memory. Continuing with the example of the *cat* command, if we take a look again at the *maps* file, the segment of memory reserved for the `cat` executable which is read-only and executable, contains the code section. We can conclude that the beginning of the text section in memory is exactly at where this segment begins, plus the offset of the code section in the executable file. To get the ending offset we obviously just need to sum the size of the section. In practice, the code section begins at *0x08048000* + *0xcc0* position and ends at *0x08048000* + *0xcc0* + *0x48dc*.

We know where the code sections begins and ends in the memory image and in the corresponding file, so we are able to build our code integrity checker. The following pseudo-code outlines the memory scan which is implemented in the Appendix B.

```
for each process
    access /proc/<process_pid>/maps
    for each "r-xp" block
```

```
get info on the .text section for the object (in disk)
compare disk and memory image
```

Once again, accessing the memory of another process turns out to be easy with the help of ptrace. By providing PTRACE_PEEKTEXT as the first argument, ptrace reads a word at the location specified by *addr* (third argument).

3.4.2 Detecting and preventing Hardware Breakpoint Hooking

As described earlier in this chapter, hardware breakpoint hooking relies in the use of two system calls: `sys_ptrace` and `sys_wait4`. If we are trying to detect these hooks, it would be a good idea to analyze the sequence of calls to these important system functions and understand the information flow that passes through them. This way, maybe we could detect when some application is trying to redirect the instruction pointer register with hardware breakpoint hooking and we could even block this attempt. System call interception, even when used for good purposes, is seriously discouraged mostly because it can cause problems in SMP systems. These details are discussed in the "Results and discussion" chapter 5. But if we want to be watching the data that goes in and out of these system calls, there does not seem to be a better choice unless we actually change the system call inside the Kernel code and recompile it, thus, creating a new Kernel image.

3.4.2.1 Finding sys_call_table in Linux Kernel 2.6

Because the `sys_call_table` symbol was being targeted in most cases for malicious use, this symbol is no longer exported in Linux Kernel 2.6, which means that we have to find a way to get it. Actually, there is no clean way to do this.

In conversations with a friend, I was told that one could search the Kernel memory for the three addresses of the first system calls consecutively - `sys_exit`, `sys_fork` and `sys_read` - with the help of the */proc/kallsyms* device and it would point to the beginning of the table. In fact, it works but this method turned out to be unreliable because it presupposes that the system call table is untouched. Besides that, this technique scans the whole Kernel memory which can be slow, although this is not a big problem since this scan is made only once.

In Phrack issue 0x3a, the article "Linux on-the-fly Kernel patching without LKM" shows a clever, but dirty way to get the system call table using the interrupt descriptor table (IDT). Actually, the article goes a little further, explaining how to get the system call table from User-mode, with the help of the */dev/kmem* device, but we can easily reproduce this technique in a Linux Kernel module because in Kernel-land, memory access has no boundaries. We are actually using a rootkit technique to perform our defenses! It may sound weird when we think about it, but if we look at the majority of the proactive

defense software in Microsoft Windows for example, most of the protections are made with system service descriptor table (the "system call table" for Windows) hooks.

To avoid being eluded by some sort of malware, and since the system call table is always at a fixed position during system operation, for security reasons, the module should save the address in a configuration file.

3.4.2.2 Formulating a pattern for detection

I divided the detection in two parts, which can be understood as two states of an automaton. First, we have to detect that a process has stopped due to a hardware breakpoint trap, and that another tracing process is receiving that information. After this, we need to detect that do tracing process is trying to change the instruction pointer right after this trap.

In the subsection 3.3.3, we saw that the system call `sys_wait4` provides a `status` argument. By intercepting this system call, we have access to the information that passes through it and we can understand what the application is doing with the traced process. Again, by doing the same verification as the attacker did to the `status` variable we can determine if it stopped due to a hardware breakpoint trap even before the attacker's application.

With the following three verifications, we can make sure that the child process has stopped due to a hardware breakpoint:

- `WIFSTOPPED(status)` returns true
- `WSTOPSIG(status)` returns SIGTRAP, meaning that the child received a breakpoint trap
- Inspect the DR6 register of the child and test if it reveals that there was a breakpoint trap due to one of the debug registers (DR0 - DR3)

Since we are intercepting this particular system call to catch the breakpoint traps, we have scope for all processes. At this point, the extended instruction pointer register (`EIP`) of the traced process should be saved for later use. We can say that we are in the first state of our detection algorithm, now we need to look if the parent process tries to change the `EIP` value of the child. Here, we use the same method as for the first part, but this time we intercept the `ptrace` system call. For the tracing process to change the `EIP` register of the child, it has to do the following call to `ptrace`:

```
ptrace(PTRACE_SETREGS, pid, NULL, &regs);
```

The `&regs` variable is the address of a `user_regs_struct` structure in the calling process. This structure contains the values of all general purpose registers of the process at that instant. With this

call, `ptrace` will replace all these registers of the child process with the ones inside the structure. If we intercept the call and compare the `EIP` register inside the `&regs` structure with the previous one that we saved when the trap occurred, we can detect if the execution flow is being controlled with this debugging mechanism.

At this state, we have detected the hooking attempt. I wrote "attempt" because we can still avoid the hook from taking place by returning `-1` to the application that called `ptrace` from User-land.

When do we return to the state zero of this detection automaton (when no process has been trapped)? We can assume that when the parent process orders the child to continue with the execution (`PTRACE_CONT` as first argument of `ptrace`), the process will continue and if there occurs another hardware breakpoint, our hooked `sys_wait4` will catch it, and transit to state one.

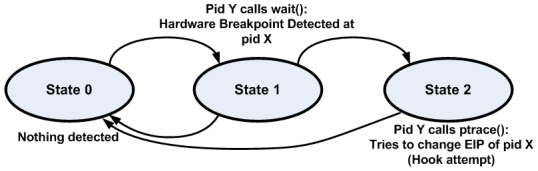

Figure 3.6: The three states of this hooking detection

The code of the proof of concept is in the Appendix C.

3.5 Summary

After understanding how the access to the memory of a process works, we are able to figure how the installation of hooks can be done. Three methods of hooking in User-land are depicted in this chapter: The first one is called *inline hooking* and relies in code section modification. It is widely used in Windows rootkits and this chapter studies its applicability under Linux. The second one was presented by *Silvio Cesare* and is called *PLT Injection*. *PLT Injection* is somehow similar to Windows IAT hooking (outlined in 3), but it is slightly more complex. The last hooking technique described in this chapter is *hardware breakpoint hooking*. It's important to realize that the distinction between code sections and the other parts of the process memory is related to the General Malware Classification section in 2.2.

The second half of the chapter focuses on the defenses where two detection methods are presented The strengths and weaknesses of these two methods are discussed later in the chapter "Results and discussion" at 5.

4 Kernel-Land

A hidden process is particularly threatening because it represents code running on your system that you are completely unaware of.

– Greg Hoglund and James Butler

4.1 Introduction

The last chapter focused on user mode rootkit attacks. Once we are aware of the details of the attack, most of the user mode techniques are relatively easy to defeat since they can be better controlled from Kernel-mode.

If the intruder gets into Kernel-mode, challenges are higher for those who seek defenses and it gets harder to ensure that the system has not been compromised. There is no doubt that a good integrity checker for the physical memory is extremely effective against rootkits that replace simple system utilities such as *ls* or *netstat*. But when the attack is performed at Kernel-mode, integrity scanning must be made at another level, concerning dynamic and non-dynamic Kernel memory. For example the *ls* command, which returns the list of the contents of a directory, delivers control to Kernel invoking the `sys_getdents`. The rootkit can modify this system call in order to hide files or even processes (of the directory listed is */proc*). Kernel rootkits are installed directly into Kernel memory area modifying important Kernel structures, function pointer tables, or patching the Kernel code, in order to filter the data that the attacker pretends to conceal from the administrator.

The current chapter covers the attack vectors used nowadays for intercepting system calls and other subverting approaches. Also describes the implementation of two applications intended to detect the threats.

4.2 Attacking the Kernel

This section explains several methods that can be used to intercept the execution flow of a system call. The figure 4.1 shows the path of an invocation to a system call from the moment that a User-land application calls, until it reaches the depths of the Kernel. The system call in the figure is `sys_getdents` (get

directory entries), and by taking a first glance at the image we can see that there are many places where the interception can be installed in order to provide what the attacker wants: Control of the call.

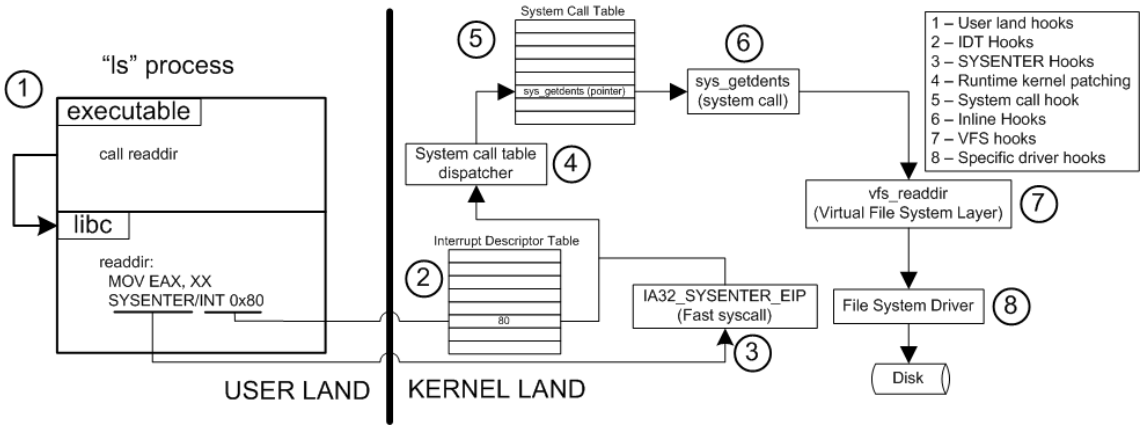

Figure 4.1: Linux Kernel Attacks - The Big Picture

Next follows the explanation of each technique that can be used to redirect the system call table.

4.2.1 Simple system call redirection

Once the system call table is found, an entry in the table can be replaced with the address of another function. Before Kernel Version 2.6, when the `sys_call_table` variable was exported, almost every rootkit used this method to hide its presence.

```
saved_syscall = sys_call_table[SYSCALL_NUMBER];
sys_call_table[SYSCALL_NUMBER] = new_syscall;
```

Today, Kernel developers hardened this task because the table is not directly available to the Kernel module, so the only challenge is to find where it is in Kernel memory. After that, it is a matter of pointer replacement. This is the most simple way of hooking a system call, and also the easiest method to detect. Nevertheless, today, rootkits tend to use other methods.

4.2.2 Inline hooking Kernel functions

Kernel inline hooking works in the same way as the inline hooking technique presented in 3.3.1, but this time, we are applying it at Kernel-land. This method can be used not only for system calls but also for any other Kernel function that the attacker might find useful to intercept. The first bytes of the function are replaced with an unconditional jump instruction forcing the program counter to branch to a new function. The new function should call the original function with the help of a trampoline,

and if needed, providing modified arguments. Return value can be changed as well. By doing this, the attacker is modifying the Kernel code directly into memory and leaving traces which are handy for an anti-rootkit. Here, the way a detector should work is similar to the one presented for the same hooking technique at user land in chapter 3: The Kernel memory should be verified for integrity with a trusted baseline.

4.2.3 Hooking interrupt 0x80 / Sysenter instruction

First, it is important to understand the role of the interrupt descriptor table (IDT). This table is a data structure designed to implement an interrupt vector table in the x86 architecture. There are 256 interrupts supported which can be triggered with one of three types of interrupts: Software interrupts, hardware interrupts and processor exceptions. The first 32 entries are reserved for processor exceptions, and any 16 of the remaining entries can be used for hardware interrupts. The rest are available for software interrupts.

Each entry in this table holds, between other security related information, the address of the interrupt handler (or Interrupt Service Routine - ISR) that processes the interruption. When an interrupt is triggered, the processor multiplies the interrupt number by the length of each entry in the table (8 bytes), and then adds the offset of the beginning of the table, finding the correct address of the interrupt handler. (*The i386 Interrupt Descriptor Table*, 2007)

The interrupt number 0x80 is a software interrupt designed to dispatch the intended system call to be executed for the User-mode application, this way, calling the `system_call` function in Kernel code. The `system_call` function inspects the contents of the `eax` register and invokes the respective system call routine indexed by the `eax` value (the number of the system call) in the system call table.

The location of the IDT is maintained by a 48 bits special register named *IDTR* which can be stored and loaded using the `sidt` (store interrupt descriptor table) and `lidt` (load interrupt descriptor table) instructions, respectively. An attacker can replace the interrupt service routine pointer for interrupt 0x80 (in the table) with another interrupt handler that fits the needs of the attacker, hiding information for certain system calls. A rootkit can also make a copy of the table and change the "official" base address of the IDT, to point to a new table, by modifying the IDTR register. This can be stealth because the attacker can hide modifications made to the interrupt table if the anti-rootkit scanner scans only for modifications of the original table. Another alternative is to patch the `system_call` function, which is located in the Kernel code, via inline hooking.

Although, newer platforms such as Windows XP, 2003, vista and recent versions of Linux Kernel 2.6 use another method to call the system services. The occurrence of an interruption causes the CPU to load one interrupt gate and one segment descriptor from memory to know what interrupt han-

dler to call. This incurs in a considerable overhead that is aggravated with high frequency of system service calls made by user mode applications. Because of performance reasons, *INTEL* and *AMD* simultaneously and independently developed their versions of a fast system call in alternative to the interrupt 0x80 mechanism. *INTEL* came up with the `SYSENTER` instruction (Garg, 2006) and *AMD* with `SYSCALL`. Both are very similar in what concerns to its functionality, but in this work I will focus on the `SYSENTER` fast system call. When the `SYSENTER` instruction is invoked, processor passes control to the "`IA32_SYSENTER_EIP`" which is a register in one of the Model-Specific Registers (MSRs). In Linux Kernel, `IA32_SYSENTER_EIP` contains the address of the function `sysenter_entry`, which will process the system call as explained earlier in this subsection.

If we want to control the execution flow triggered by the `SYSENTER` instruction, we need to manipulate the `IA32_SYSENTER_EIP` register, but, since this register is privileged, we are only able to read and write to it from Ring Zero, using the `rdmsr` and `wrmsr` instructions.

The following assembly code ilustrates how this hook can be installed:

```
movl $0x176, ecx //location of the IA32_SYSENTER_EIP register
rdmsr //read register value
movl eax, orig_sysenter //store in orig_sysenter
movl new_sysenter, eax //store into eax register the new SYSENTER routine
wrmsr //save it to IA32_SYSENTER_EIP
```

After the execution of this code, if `SYSENTER` was not already hijacked, `orig_sysenter` should be pointing to `sysenter_entry` Kernel function. It is important not to forget to restore the register when unloading the rootkit or the system will crash because every system call uses the `SYSENTER` instruction and it is pointing to an invalid address. When inside the `new_sysenter` function, one is able to check and modify the arguments in the stack which have all the information about the invoked system call from User-land. After the payload, the `new_sysenter` function must jump to the `orig_sysenter`, this way, keeping the original system behaviour. Yet, I never saw a Linux rootkit using this technique.

Detecting a `SYSENTER` hook is extremely easy: Just read the value of `IA32_SYSENTER_EIP` and check if it has the address of `sysenter_entry` function. If not, some malware should be controlling the execution flow.

4.2.4 Runtime Kernel patching

Runtime Kernel patching is similar to inline hooking in the sense that it is based in Kernel memory modification. In 1998, Silvio Cesare presented several methods (Cesare, 1998) for extracting and modifying sensitive information from the Kernel through the */dev/kmem* device. Three years later, in 2001, *sd* and *devik* published an article in the phrack magazine showing how to subvert the Kernel into using another system call table (sd & devik, 2001), instead of the original one.

As explained in the second chapter, nowadays, this is not much of a concern because the */dev/kmem* device is read-only in the recent versions of the Kernel. However, if the Linux Kernel is compiled with LKM support, it is still possible to use the same method to force the Kernel to use another system call table created by the rootkit, possibly eluding a weak anti-rootkit if it only searches for modifications in the original table.

Because Kernel no longer exports the `sys_call_table` symbol, this method seems to be the best way to find its address. The `system_call` function takes the system call number and uses it to index into the `sys_call_table` vector to find the specific system call needed by the user land application. Somewhere after the beginning of that function, resides the call to the actual table. We are going to search for the following code:

```
call *sys_call_table(,%eax,4)
```

Which correspond to bytes "`ff 14 85 XX XX XX XX`" where XX bytes belong to the actual `sys_call_table` address, the one we are searching. The following steps resume the technique:

1. Find IDT base address with `sidt` instruction.
2. Seek to the entry corresponding to `int 0x80` in the table.
3. Get the interrupt handler address of `system_call` from the table entry.
4. Search for the three consecutive bytes: `ff 14 85`.

Now, the attacker can change the four bytes after the pattern, placing the address of a new system call table, which he can happily modify. Since this method relies in byte code searching, it is not guaranteed that will work between Kernel versions, but I have tested it with many Linux 2.6 Kernels and it seems to be quite portable. Probably because it's unlikely that the system call mechanics are changed.

4.3 Detecting the attacks

Rootkits can be hard to detect, especially when they operate in Kernel-land because at this level they are able to alter functions used by all running applications, including anti-rootkit software. The current section describes what actions can be taken in order to protect the Kernel from the attacks shown in the previous section.

4.3.1 Kernel Integrity Checking

Generally, integrity checkers are more concerned about finding suspicious activities by searching in the physical storage. Physical storage is, indeed, important to analyze and in an infected system it may

contain evidences of an intrusion, however, malware only needs to survive in memory. If the attacker concerns enough about steathiness, he should avoid the file system altogether. This means that the consistency of the running Kernel memory also should receive attention and this is what this subsection is all about.

Fingerprinting sensitive places

Fingerprinting works like a Kernel memory "photograph" of sensitive places, taken at the time that the system was clean. Later in time, the administrator can compare the current Kernel memory image with the fingerprint taken previously, and check for system modifications.

This method can lead to false positives if the administrator does not make a fingerprint after every relevant system change. Note that Kernel fingerprint detection is only sensitive to system changes, which means that if the fingerprint is taken when a rootkit is active, it will not detect any system change (unless the rootkit is removed, of course). The fingerprint should be saved in a safe place - preferably encrypted and/or signed - where the administrator is sure that the attacker cannot reach, in order to prevent him from taking another fingerprint of the subverted Kernel and replace the original one.

A good fingerprint should be based in the following: System call table, Interrupt Descriptor Table, *IA32_SYSENTER_EIP* address, IDTR register, Kernel Symbols (first bytes of the functions) and other important function pointers such as the */proc* virtual file system lookup function (which will detect the *adore-ng* rootkit, for example).

Zeppoo anti-rootkit (Evron, 2006) implements this detection method pretty well.

Verifying the integrity of important dynamic regions in Kernel memory

In what concerns to dynamic Kernel memory, this is somehow similar to fingerprinting but with the difference that the last one blindly saves the information without checking its validity.

Here, a good anti-rootkit should inspect the correctness of all regions where important information is stored in the system, in other words, looking for kooks. These are the type of integrity checks that are used in the next subsection with the "Proactive Detection Approach". Continue reading for deeper details.

4.3.2 A Proactive Detection Approach

In this subsection, I present a proactive detection approach based in the analysis of certain parts of the Kernel before the invocation of a system call. Note that this is different from a Proactive Defense system - as the Kaspersky Anti-Rootkit Software for Windows - because this method does not prevent

the malware from being installed, but it detects that it is in Kernel memory. The code of the proof of concept is in the Appendix D.

Because the system call table is usually targeted by rootkits and depending on each system call that is being called, the malware may be subverting different parts of the Kernel, it should be a good idea to check specific locations of the system for consistency, when each system call is invoked. Being able of executing code after a User-land request to the Kernel and before the execution of a system call, gives us the possibility to scan certain parts of the system when it is more convenient, hence the word "proactive".

The basic idea behind this method is to clone the system call table, and replace every function pointer to point to stub functions, where some verifications to the system can be performed. When the checks are done, the actual function call is invoked with the help of the cloned table, which is still the same as the original table before the installation of the defense system.

The figure 4.2 illustrates the implementation of this defense system.

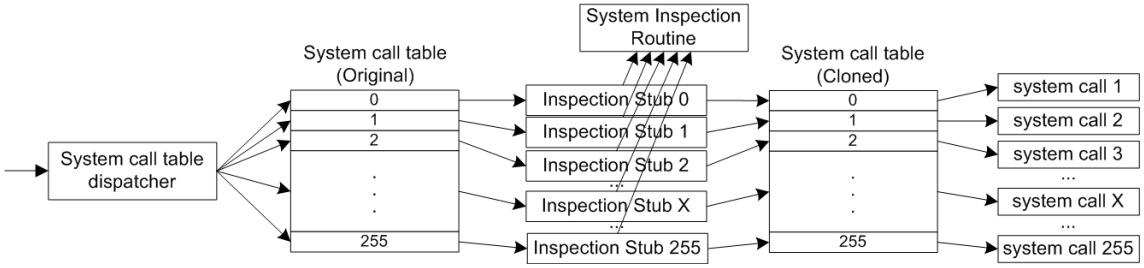

Figure 4.2: Kernel Proactive Detection System

By looking at the figure 4.2, the following pertinent question might outcome: "Why not redirect every entry in the system call table to the same *Inspection Stub*?". The reason for this relies in the fact that with a unique *Inspection Stub*, we would only have one way to identify the destination system call - reading the value of `EAX` register - which is completely unreliable if a rootkit intercepts the system call and changes it.

Since system calls are invoked so many times during system operation, it is important to reduce - as much as possible - the overhead created by the checks of this detection system. Relevant checks to be made in the "*System Inspection Routine*" are described:

Check the system call table entry When a certain system call is invoked, we can check if the corresponding entry in the original system call table is correct, meaning that it should contain the address of the respective *Inspection Stub*.

Check Call Stack Note that when the *Inspection Stub* is executed, the call stack should contain always the same functions. This leaves us with a method for detecting another possible incongruity: Check if the call stack is something similar to the following:

```
[<e0a8b2cd>] Inspection_Stub+0xd/0x20 [Inside module code]
[<c01040f2>] sysenter_past_esp+0x6b/0xa9 [Inside Kernel code]
```

If the call stack shows more than these two functions, probably there is rootkit code being executed.

Nevertheless, for performance reasons, the default Kernel configuration comes without frame pointers which means that we won't be able to have good stack traces having the `EBP` register as a general purpose register. Selecting the "Compile the Kernel with frame pointers" option (if present) in the Kernel configuration menu, sets the *CONFIG_FRAME_POINTER* flag enabling the `EBP` register as the frame pointer. Actually, the frame pointer has no other use unless to provide a correct stack trace, so developers decided to remove the use of it in order to have a smaller and slightly faster Linux Kernel image for default instalations.

So, for this check to work, one has to enable this option before compiling the Kernel.

Check first bytes of the system call function and the system call table dispatcher function Here, one of three alternatives can be taken: (1) Check for an unconditional jump at the beginning of each function to detect a possible inline hook. (2) Test the integrity of the first bytes with a trusted database. This database should have previously gathered this information. (3) Check if the first bytes are unchanged with the help of the *vmlinux* file

Check IA32_SYSENTER_EIP If the system uses the `SYSENTER` fast call, it is important to check if the `IA32_SYSENTER_EIP` register has the address of the correct Kernel function, which is *sysenter_past_eip*.

Check "int 0x80" In case the system is using the old method (with the Interrupt Descriptor Table), test if entry 0x80 in the IDT has the address of the correct interrupt service routine, which is the `system_call` function.

Check for patched Kernel code As stated above in 4.2.4, we can check if the correct system call table is being used by inspecting the byte codes in the `system_call` function.

Check Virtual file system If the system call is related to disk input/output, the defense system may check some relevant function pointers of the virtual file system (Brown, 1999).

When a check in this system detects that something is wrong, it cannot stop the system call from being executed because it will get the system unstable. Instead, it should alert the administrator and give as much details as possible so he or her can understand what is wrong and where.

With this proactive detection system running, we can defend the system against most of the attacks stated in section 4.2.

Table 4.1: Correspondence between each system check and Kernel attack

Check to perform	**Attack to detect**
System Call Table Pointers	Simple Redirection
Verify CallStack	IDT/Sysenter/Simple Redirection
Byte Code	Inline Hooking
Sysenter	Sysenter Hook
int 0x80	IDT Hook
Patched Code	Runtime Patching
VFS	Other Lower level hooks

The table 4.1 relates each check described above with the respective attack that it intends to detect.

4.4 Summary

Several ways of attacking the Linux Kernel were explained in this chapter concerning the system call invocation mechanism. Almost every hiding technique ends up redundantly in hooking, this is why the term is having so much prominence in this thesis. Also, most of the communications between User-mode and the Kernel-mode are exchanged with the help of the system call table, resulting in a common attack vector for the attackers. The second half of this chapter covered different ways of detecting these attacks, having them concentrated into an IDS which was built under the scope of this work.

5 Results and discussion

More-advanced rootkit techniques and their detection are being developed as you read these words.

– Greg Hoglund and James Butler

5.1 Introduction

We need to retrieve the profits of this study with an analysis for each new attack and defense technique presented in this thesis. Generally speaking, each analysis has the goal of understanding the benefits and the drawbacks of each method. In most cases, these methods won't have problems with false negatives because they are designed to detect specific attacks of rootkits. On the other hand, false positives can be a disadvantage since there can be system changes because of the installation of specific software or system upgrades.

5.2 Process code integrity scanner

Section 3.4.1 described the implementation of an application memory integrity scanner, specifically for code sections. The tool is a simple application which scans the memory of all running processes on the system and compares the code sections in memory with the versions in the ELF executable and in the shared objects. I designed it to prove that the idea I had was effective against inline hooking and other code modification attacks. This subsection intends to analyze the execution performance and disadvantages of this detection technique.

5.2.1 Execution

Unlike other intrusive detection tools which are installed on the system, this one is different: It only checks the system when executed and takes about two to three minutes to complete, depending on the factors such as processor, disk access and memory bus speed. Next follows a snippet of the execution of the tool, detecting byte modification in a process.

```
(...)
[05:04:50] Scanning pid 5435...  OK
```

```
[05:04:51] Scanning pid 5438...  OK
[05:04:52] Scanning pid 5683...  OK
[05:04:54] Scanning pid 5736...  OK
[05:04:56] Scanning pid 5738...  OK
[05:04:56] Scanning pid 5758...  OK
[05:04:58] Scanning pid 5765...  OK
[05:04:58] Scanning pid 5977...
WARNING - Offset:  0x08048484:  0x83e58955 (DISK) != 0xe9aaaaaa (MEM)!
WARNING - Offset:  0x08048488:  0x04c708ec (DISK) != 0x04c708aa (MEM)!
Object ``/home/andre/thesis/inline/inline'' modified in memory!
Failed
```

The previous example detected a simple inline hook in a running process. Both bytes in disk and memory are reported as also the position where the modification occurred so that the administrator can check which function was changed (if there is enough debugging information about the process).

Clearly, the way the detection tool is used is not the most handful. An administrator would prefer the tool to be proactive, meaning that it scans the memory, periodically, when more convenient and without blindly wasting system resources.

5.2.2 Problems

Not all processes can be scanned: *Init* (PID 1) simply cannot be attached to ptrace. Processes that are already being debugged also cannot be attached. This can lead to a defense mechanism from the attackers point of view, because a simple attach to ptrace can leave the process immune to scans.

Scanning *.text* sections might not be enough. It is possible that there are other read-only sections sensitive to other attacks beyond code sections that should be scanned for integrity.

Also, there could be legitimate applications changing their own code sections at runtime leading to false positives. Fortunately, this is uncommon. The only way to minimize false positives is to provide a built in exclusion option in the tool to ignore certain offsets of specific applications causing the false positives.

The use of the *ptrace* system call causes an overhead that would not occur if the scanning was made through a Linux Kernel Module. Unlike this tool which runs at User-level, at Kernel-level memory can be accessed directly without suffering from the slowdown caused by the *interrupt 0x80* or the *sysenter* instruction. Note that *CR3* register must be correctly assigned in order to access the address space of the process to be scanned, as described in 3.2.

5.2.3 Other approaches

A more interesting approach would be to hook the *sys_execve* and the *sys_fork* system calls, which are the only way to bring a new process to memory, and inspect every application when loaded. After the first inspection, periodic checks should be also performed from time to time.

All the information about the permissions of the sections of the loaded process image are in the program header table, in the ELF file. The tool could use this information to find where these sections are located so that it can check the integrity.

5.3 Hardware breakpoint hooking attack

The technique is described in 3.3.3. Here we will see an example of the attack in action, and discuss the traces it leaves to a forensic analysis.

5.3.1 Execution

The following simple C application was used to illustrate the execution of this attack:

```
//function to be intercepted
int simple(int value){
  printf("Simple:  argument!  %d\n", value);
  return 5;
}

int main(){
  printf("Simple() returned:  %d\n", simple(10));
  printf("Running simple again:\n");
  printf("Simple() returned:  %d\n", simple(11));
  return 0;
}
```

Function *simple()* will be fully hijacked, meaning that its parameters and return value are modified. The following function code is the function that will be jumped to, when the breakpoint occurs in the attacker's process. In other words: *EIP* register is changed to address of *hijack_simple*.

```
//prototype of the function to be intercepted
typedef int (*simple_func_t) (int);
simple_func_t simple_func;

//hijacking function
int hijack_simple(int value) {
    int new_value = 31338;
```

```
    simple_func = (simple_func_t)HIJACK_ADDR;
    simple_func(new_value); //call the original
    return 31337;
}
```

Normal execution of the application

```
PID: 7239 wait 2 seconds...
Simple:  argument!  10
Simple() returned:  5
Running simple again:
Simple:  argument!  11
Simple() returned:  5
```

Execution of the application with hardware breakpoint hooking

```
PID: 7239 wait 2 seconds...
Simple:  argument!  31338
Simple() returned:  31337
Running simple again:
Simple:  argument!  31338
Simple() returned:  31337
```

5.3.2 Analysis of stealth

Unlike to *inline hooking* or *PLT injection* (see 3.3), the code sections and other function pointer tables in the process remain untouched. On the other hand, debug registers are modified, which is one of the evidences left by this technique. Nevertheless, the fact that a process got its debug registers modified does not mean that it is being attacked because it can be under a debug operation using hardware breakpoints. Also, there must be another process to perform the control of the hook itself (to modify the debug registers, wait for the process to stop at the breakpoint address and change the *EIP* value), which is yet another clue that something might be wrong with the system.

Note that the method presented here uses the *ptrace* system call but it is also possible to implement it at Kernel level through the help of an LKM, which is much more complex but stealthier.

5.4 Preventing hardware breakpoint hooking

The current approach to hardware breakpoint prevention is able to stop a hook attempt from taking place by making the attacker's code sterile and without harming the target application. This section

uses the example given in the previous section to show the effectiveness of this approach.

5.4.1 Execution

Execution of the application with hardware breakpoint hooking enabled and the prevention system active

```
PID: 5165 wait 2 seconds...
PTRACE_SETREGS: Operation not permitted
Simple:  argument!  10
Simple() returned:  5
Running simple again:
Simple:  argument!  11
Simple() returned:  5
```

The *PTRACE_SETREGS* error is printed to *stderr* from the process controlling the hook and reveals that it could not set the *EIP* register on the target process because the system call was blocked by the prevention system. As a result, we can see that this prevention method avoids any change in the execution flow if this attack is attempted.

Since the Kernel module uses the `printk` function to log its activity, the following line is logged to */var/log/messages* which can be fetched with the *dmesg* command:

```
WARNING: PID: 5173 - HARDWARE BREAKPOINT HOOK ATTEMPT on process 5174!
```

Both attacker's process and target process are identified and debug registers used to perform the attack are restored as soon as the LKM detects the hook attempt.

5.4.2 Problems

The refered prevention method is only effective if the attack is performed using `sys_ptrace` and the `sys_wait4` (or one of its derivatives), which is what will be probably used if it is made from User-land. This defense can also work if a dumb rootkit uses these system calls from kernel land, which is not so unlikely to happen as seen by the poorly written malware nowadays. At Kernel-land, the attack is harder to control because the intruder has the same privileges as the defense system and a smarter approach to this hook will certainly not use the above system calls, thus, bypassing this defense.

The pattern behavior used in this prevention system is unlikely to cause false positives because it's a very specific automaton that should not occur in legitimate applications. The detection only takes place when the instruction pointer is modified after the breakpoint trap, which means that there must already be some execution flow manipulation. In what concerns to the use of hooking for good purposes: for example, if an application uses a hooking method to avoid having to restart after an update, that method

will not be hardware breakpoint hooking because it is slower (cost of an interruption) and consumes resources that a software hooking technique won't need.

5.5 Kernel Proactive Detection Approach

In the same manner as the hardware breakpoint defense tool in the previous section, this implementation is also a Linux Kernel module that logs information to */var/log/messages*. On the other hand, this system does not prevent the malicious activity from reaching its goals, leaving just an alert to the administrator. Analysis of the accuracy, performance and drawbacks of this implementation are dissected in this section.

5.5.1 Execution

Next follows a snippet of the logged file containing the output information printed by the detection tool when loaded into Kernel.

```
SCT Defender Loaded!...
Interrupt 80 OK
Sysenter instruction OK
SCT Defender:  sys_call_table at c0308500
SCT Defender:  verifying sys_call_table integrity...
Done.
System_call jump OK
SCT Defender:  Active.
```

When the module is loaded, the following checks are performed before the installation of the defense system:

- Check Interrupt Service Routine 80 address in the interrupt descriptor table.
- Check `sysenter` instruction for hooks.
- Check the whole system call table.
- Check for inline hooks on every system call table.
- Check the system call jump in the *system_call* function in Kernel code.

Due to the extreme speed of these verifications, it is acceptable to perform them when the module is loaded because it does not incur in a significant overhead. The same is not so true when the IDS is installed and running.

Let's see the scenario of a system call table entry redirection. Figure 5.1 illustrates the attack of the `sys_getuid32` system call hook along with this IDS installed. To ease the understanding, rootkit information is displayed in gray color.

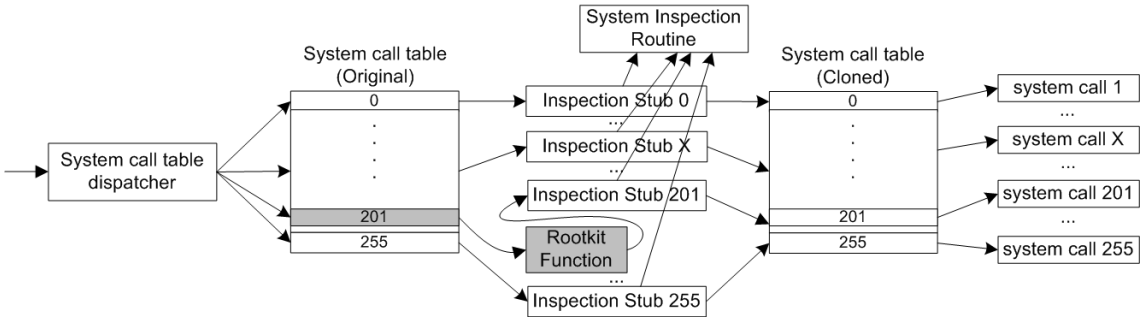

Figure 5.1: Detecting system call redirection attack

Two checks will detect this system change as soon as the system call is invoked by any User-land application: (1) The entry in the system table has now the address of the attackers's function (the one that hooks the system call). (2) The call stack is modified, showing again, that same function. The following *warning* lines are printed to */var/log/messages*.

```
WARNING! Syscall 201 hijacked!
WARNING! 0xe0a750a8 should be inside kernel code!
WARNING! Call stack too big, something is wrong!
```

The first line is the check (1) alerting that the system call entry number 201 was modified. The remaining two lines are related to the call stack check. Let's see how the stack looks like at this time:

```
[<e0a858bd>] syscall_stub201+0xd/0x20 [sctdefender]
[<e0a750a8>] hijack_getuid32+0x18/0x20 [rootkit]
[<c01040f2>] sysenter_past_esp+0x6b/0xa9
```

The line in red reveals the presence of a function being invoked before the `syscall_stub201` (which belongs to the IDS). The defense system alerts because the second function in the call stack should be `sysenter_past_esp` (pointing to the Kernel code section). Also, the call stack has three functions of size instead of two.

For a careful attacker, the last check can be avoided: In the `hijack_getuid32` routine, instead of invoking the system call directly, one can `jmp` to it after restoring the stack as it was, before the execution flow reaches the `hijack_getuid32` function.

5.5.2 Performance Tests

Each system check, along with the redirection of the system call itself contributes to an overall delay of the total time that the system call takes. The present subsection studies the total overhead produced by the proposed defense system. Table 5.1 illustrates the estimated overhead in percentage relatively to the original system call time.

Table 5.1: Performance tests

	Average time (ms)	Percentage
Total system call time (no IDS)	520	0,0
No checks	535	2,8
All checks	1126	116,5
SCT	558	7,3
Call Stack	619	19,0
Sysenter	804	54,6
int 80	606	16,5
system_call jmp code	578	11,2
Syscall Inline check	565	3,6
All checks except sysenter	752	44,6

It is crucial to analyze each check to understand which are consuming more time. The results in the previous table were grabbed from the following test conditions: Measure the time taken during five million invocations to the system call *sys_time*. The "Average time (ms)" column is the average time taken in twenty of these tests. "Percentage" column helps to understand the overhead produced by these checks towards a system with no IDS installed. The tests were performed in an Intel Pentium M 760 Processor (2MB L2 Cache, 2.0 Ghz) with 1.3GB of RAM.

We can clearly see that the `sysenter` check is consuming considerable time. The reason for this is in the `rdmsr` instruction. Because the remaining verifications are based on simple byte comparison in memory (except for the call stack check which is slightly slower), they are faster and do not represent a significant overhead in a system call.

We can minimize this overhead by taking the following into consideration: Unlike the `sysenter` and `int 0x80` checks, the remaining verifications are unique for each system call, which means that we can execute the first ones with less frequency, for instance, after each ten system calls.

5.5.3 Problems

If this mechanism is implemented as described in chapter 4, there is one easy way around it. Imagine the following attack: The attacker hooks a function in the system call table. When a User-land application invokes the system call, the rootkit code gets executed and runs the original system call. But instead of

using the address in the system call table before the hook, it finds the right address of the system call by searching the kernel, using the */proc/kallsyms* device of simply searching in the *System.map* file.

In this case, the `System Inspection Routine` won't get called and no checks to the kernel will be made. To solve this problem, there should be periodic calls to this routine, each one intended to scan a different system call. However, this solution has the disadvantage of not being entirely "proactive detection", but also "passive detection" because it performs periodic scans on self-demand.

On SMP systems this implementation may cause problems. For start, there is an IDT for each processor, which is not being taken into consideration on this implementation. Also, the moment when the whole system call table is cloned and then fully replaced can cause race conditioning problems with multiple processors; This is why system call redirection is strongly discouraged by Kernel developers.

Finally, for the call stack check to work the *CONFIG_FRAME_POINTER* flag must be enabled when the Kernel is compiled (which is not by default). Revisit 4.3.2 to understand the use of this flag.

5.6 Summary

The present chapter analyzed every implementation of this thesis measuring the advantages and disadvantages from the defender's point of view. It begins showing that the code integrity scanner is effective for any attack involving code modification in the memory of a process. Also, the results of implementations concerning hardware breakpoint methods were analyzed, illustrating the attack itself (in User-mode) and the respective prevention system. Finally, despite some issues, the Kernel defense mechanism presented in chapter 4 was tested and proven to be reliable not only in terms of performance but also in effectiveness. Though, it is not infallible.

As we could see, we cannot say that these detection methods are foolproof.

6 Conclusions

The fact that a product claims to provide some level of protection does not necessarily mean it actually does.

– Greg Hoglund and James Butler

In chapter 3, I provided information about Linux hooking techniques focused on user land hooks, but in the case of the hardware breakpoint hooking detection, with some defenses from Kernel-land. User-land hooks are relatively easy to detect and prevent, but the same is not so true for Kernel-land hooking. The Intel x86 debugging mechanism is powerful and can be used and abused for stealth purposes. As we could realize, hooking is a sophisticated technology for execution flow control that is used not only by malware but also by many anti-virus applications and host-based intrusion detection and prevention systems.

Detection techniques do not prevent the system from being attacked in the first place. Generally, they only indicate what is wrong with the system and where is the incongruity. Some of them only alert when the detection tool is executed (because of a scheduled scan or on administrator's demand). The approach presented in chapter 4 is permanently running in background as a Linux Kernel module, and logs suspicious system changes when detected. Although, it still does not prevent the attacks from taking place because the invocations to the system calls cannot be stopped. If they are, the system will be left in an inconsistent state. Generally, the smartest thing to do when the administrator receives the alert of an intrusion, is to reinstall the entire system. The administrator can trust his or her personal anti-rootkit tools for removing the threat, but can never be sure that the system is clean simply because rootkits are a constantly growing threat. There are always new and different ways of hiding resources in the system and some of them have never been disclosed to the public.

The results obtained ensure that the goals initially outlined in the first chapter were reached for the studied hiding techniques. Although, we are far from stating that the work is done. In fact, a good researcher may say that it will never be done: The reason concerns the constant evolution of rootkits. Tests performed with the IDS's against rootkits implementing the respective hiding technologies had successful results, and helped to understand failures in the solution that attackers could exploit.

It's interesting to watch the evolution of rootkits through the time: At the beginning, a rootkit was no more than a set of tools, designed to replace certain system utilities. Nowadays, most rootkits come in the form of a driver that enters Kernel and subverts some part of it. In the future, rootkits may

find new stealthy places, attacking not the operating system, but the hardware such as the processor's microcode, or other microchips in the computer. There is already a hardware based anti-rootkit card called "Copilot", which is a PCI card. This card is plugged on the host machine and has its own CPU. The idea is to work independently from the potentially infected operating system, and scan physical memory through DMA for signs of a rootkit via Kernel memory integrity checks and techniques to detect Kernel hooks.

We can conclude that this is like the cat-and-mouse game: At some instant, rootkit authors may be winning, but then anti-rootkit security specialists strike back, and the cycle repeats itself. This is somehow similar to online game cheating: cheaters bypass the game protections, and then anti-cheat software updates, and catches cheaters. This competition ends up being healthy for technology advancements since this brings motivation to the security scenario.

Finally, despite of these methods not being foolproof, this does not mean that they don't reserve attention from security specialists. Let's consider the analogy of a bank: Guards don't just lock the doors; there are motion sensors, cameras and other surveillance systems that do not guarantee total security but it is another valuable step towards a perfectly safe system.

6.1 Future Work

In some aspects, this research is incomplete not only because of the subject's obscurity - making it hard study - but also because there is always new information on the topic coming out. Next follows a list of extensions to the implementations and new research vectors on this thesis.

- Improve the User-land memory scanner in order to scan all read-only segments and not only code sections of applications. Also there should be possible to exclude certain process regions, to mitigate false positives.
- Research the defenses against stronger implementations of the hardware breakpoint attack such as in Kernel-land. A good starting point would be the phrack paper (halfdead, 2008).
- Create a more secure and sophisticated alert mechanism for the proactive detection system. The current implementation only logs the alerts to */var/log/messages* through the `printk` routine.
- For performance reasons, slower checks in the proactive detection system, such as the *SYSENTER*, should have lower checking rates, ie, the check should only take place at each X invocations to a system call. The X value should be configurable.
- To mitigate the problem described in 5.5.3, periodic scans should be performed in the proactive detection system (with a configurable time interval).

- In the proactive detection system: Create specific checks for each system call. The 4.3.2 subsection suggests the VFS layer check when a *sys_getdents* system call is triggered. Besides this one, other verifications could be implemented, this way, improving the overall trustworthiness of the anti-rootkit software.
- Study direct attacks to the IDS - Analyze possible attack vectors directly to the defense system and ways to prevent them.

Bibliography

Brown, N. (1999). *The linux virtual file-system layer.* (http://cgi.cse.unsw.edu.au/~neilb/oss/linux-commentary/vfs.html (visited on 2008/09/29))

Butler, J. (2004). *Vice - catch the hookers!* (http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-butler/bh-us-04-butler.pdf (visited on 2008/02/10))

Cesare, S. (1998). *Runtime kernel kmem patching.* (http://vx.netlux.org/lib/vsc07.html (visited on 2008/09/29))

Cesare, S. (1999). *Plt injection.* (http://vx.netlux.org/lib/vsc06.html (visited on 2008/08/04))

Clandestiny. (n.d.). *A filter driver example using a kernel key logger key logger.* (http://www.rootkit.com/newsread.php?newsid=187 (visited on 2008/01/11))

Corporation, M. (n.d.). *Detours.* (http://research.microsoft.com/sn/detours (visited on 2008/01/11))

Elf specification. (1995). (www.x86.org/ftp/manuals/tools/elf.pdf (visited on 2008/08/20))

Evron, G. (2006). *Zeppoo: Decent rootkit detection for linux.* (http://blogs.securiteam.com/index.php/archives/433 (visited on 2008/09/29))

Florio, E. (2005, December). *When malware meets rootkits.* (http://www.symantec.com/avcenter/reference/when.malware.meets.rootkits.pdf (visited on 2008/01/10))

Garg, M. (2006). *Sysenter - fast transition to system call entry point.* (http://manugarg.googlepages.com/systemcallinlinux2_6.html (visited on 2008/09/29))

The gnu project debugger. (2008). (http://www.gnu.org/software/gdb/ (visited on 2008/08/17))

halfdead. (2008). *Mistifying the debugger.* (http://www.phrack.com/issues.html?issue=65&id=8 (visited on 2008/09/29))

Haungs, M. L. (1998). *The executable and linking format.* (http://www.cs.ucdavis.edu/~haungs/paper/node10.html (visited on 2008/09/29))

The i386 interrupt descriptor table. (2007). (http://www.acm.uiuc.edu/sigops/roll_your_own/i386/idt.html (visited on 2008/09/29))

Intel. (2008). *Intel ia-32 manual.* (http://download.intel.com/design/processor/manuals/253668.pdf (visited on 2008/08/07))

Kaspersky lab. (n.d.). (www.kaspersky.com (visited on 2008/05/24))

Microsoft. (2006). *Virtualization.* (http://www.microsoft.com/whdc/system/platform/virtual/CPUVirtExt.mspx (visited on 2008/09/29))

Miller, T. (2000, November). *Analysis of the t0rn rootkit.* (http://www.securityfocus.com/infocus/1230 (visited on 2008/05/21))

MSDN. (1999). *Memory management functions.* (http://msdn.microsoft.com/en-us/library/aa366781(VS.85).aspx (visited on 2008/09/29))

ptrace linux man page. (1999). (http://www.linuxmanpages.com/man2/ptrace.2.php (visited on 2008/08/05))

Rutkowska, J. (n.d.). *System virginity verifier.* (http://www.invisiblethings.org/papers/hitb05_virginity_verifier.ppt (visited on 2008/03/04))

Rutkowska, J. (2004, January). *Detecting windows server compromises with patchfinder 2.* (http://www.invisiblethings.org/papers/rootkits_detection_with_patchfinder2.pdf (visited on 2008/04/15))

Rutkowska, J. (2006, November). *Malware taxonomy.* (http://www.invisiblethings.org/papers/malware-taxonomy.pdf (visited on 2008/02/16))

sd, & devik. (2001). *Linux on-the-fly kernel patching without lkm.* (http://www.phrack.com/issues.html?issue=58&id=7 (visited on 2008/09/29))

Tripwire. (n.d.). (http://sourceforge.net/projects/tripwire (visited on 2008/04/17))

waitpid system call. (1997). (http://www.linuxmanpages.com/man2/wait4.2.php (visited on 2008/08/17))

I Appendices

Appendix A

Hardware breakpoint hooking proof of concept

— Begin of “test.c” —

```
/* Basic application to be intercepted with the hardware breakpoint method
 *
 * Compile:
 * gcc -o test test.c
 *
 * Author: Andre Almeida
 */

#include "stdio.h"
#include <signal.h>

int simple(int value){
  printf("Simple: argument! %d\n", value);
  return 5;
}

int main(){
  int i;
  printf("PID: %d wait 2 seconds...\n", getpid());

  sleep(1);
  printf("Simple() returned: %d\n", simple(10));

  printf("Running simple again:\n");
  printf("Simple() returned: %d\n", simple(11));

  return 0;
}
```

— End of “test.c” —

— Begin of “hijack.c” —

```
/*
 * Hardware breakpoint hooking proof of concept
 *
 * Compile:
 * gcc -fPIC -c hijack.c -o hijack.o
 * gcc -shared -o hijack.so hijack.o
 * LD_PRELOAD="./hijack.so" ./test
```

```
 *
 * Author: Andre Almeida
 */

#include <sys/ptrace.h>
#include <wait.h>
#include <sys/types.h>
#include <linux/user.h>
#include <stdio.h>
#include <stdlib.h>

// The beginning of the function to intercept
// Compile the test.c application and check in GDB the
// address of the symbol: simple_function
#define HIJACK_ADDR	0x08048414

//Enable DR1 and Break on instruction execution only
#define ENABLE_DR0	0x00000001

#if !defined(offsetof)
#   define offsetof(type,memb) ((size_t)&((type*)0)->memb)
#endif

typedef int (*simple_func_t) (int);
simple_func_t simple_func;

int set_breakpoint(pid_t pid){
	if(ptrace(PTRACE_POKEUSER, pid, offsetof(struct user, u_debugreg[0]),
		HIJACK_ADDR) != 0){
		perror("PTRACE_POKEUSER");
		return 0;
	}
	if(ptrace(PTRACE_POKEUSER, pid, offsetof(struct user, u_debugreg[7]),
		ENABLE_DR0) != 0){
		perror("PTRACE_POKEUSER");
		return 0;
	}
	return 1;
}

int clear_breakpoint(pid_t pid){
	if(ptrace(PTRACE_POKEUSER, pid, offsetof(struct user, u_debugreg[0]),
		0x00000000) != 0){
		perror("PTRACE_POKEUSER");
		return 0;
	}
```

```
        if(ptrace(PTRACE_POKEUSER, pid, offsetof(struct user, u_debugreg[7]),
                0x00000000) != 0){
                perror("PTRACE_POKEUSER");
                return 0;
        }
        return 1;
}

int hijack_simple(int value) {
        int new_value = 31338;
        simple_func = (simple_func_t)HIJACK_ADDR;
        printf("Executing hijack_simple parameter: %d, new parameter: %d\n",
                        value, new_value);
        simple_func(new_value);
        return 31337;
}

pid_t sigtrap_wait(){
        pid_t waitret;
        int status;
        while(1){
                if((waitret = wait(&status)) == -1) {// Wait for child's signal
                        perror("wait");
                        break;
                }
                if(WIFSTOPPED(status)){
                        if(WSTOPSIG(status) == SIGTRAP){ /* it is a breakpoint trap */
                                return waitret;
                        }else
                                continue;
                }else
                        continue;
        }
        return waitret;
}

void __attribute__((constructor))
startup()
{
        int status;
        pid_t pid, testpid;
        struct user_regs_struct regs;
        switch (pid = fork()) {
        case -1:
                perror("fork");
        break;
```

```
case 0: /* child process starts */

break;
default:
        if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) != 0)
                perror("PTRACE_ATTACH");

        if(wait(&status) == -1) // Wait for child's signal
                break;

        //Change the debug registers
        if(!set_breakpoint(pid))
                break;

        if (ptrace(PTRACE_CONT, pid, 0, 0) != 0)
                                perror("PTRACE_CONT");

        while(1) {
                if(sigtrap_wait() == -1) {// Wait for child's signal
                        break;
                }

                // changing EIP
                if (ptrace(PTRACE_GETREGS, pid, NULL, &regs) != 0){
                        perror("PTRACE_GETREGS");
                        break;
                }

                regs.eip = (int)hijack_simple;
                if (ptrace(PTRACE_SETREGS, pid, NULL, &regs) != 0){
                        perror("PTRACE_SETREGS");
                        break;
                }

                if (ptrace(PTRACE_CONT, pid, 0, 0) != 0){
                        perror("PTRACE_CONT");
                        break;
                }

                if(sigtrap_wait() == -1) {// Wait for child's signal
                        break;
                }

                /* clear breakpoint while we execute the instruction */
                if(!clear_breakpoint(pid))
                        return;
```

```
                        if (ptrace(PTRACE_SINGLESTEP, pid, 0, 0) != 0){
                                perror("PTRACE_STEP");
                                break;
                        }

                        /* restore breakpoint */
                        if(!set_breakpoint(pid))
                                return;

                        if (ptrace(PTRACE_CONT, pid, 0, 0) != 0){
                                perror("PTRACE_CONT");
                                break;
                        }
                        //continuing the cycle to maintain the hook active
                }
                exit(0); /* exit process because return runs the executable
                                        (this forked process) */
        }
}
```

— End of "hijack.c" —

Appendix B

User-Land Memory Integrity Scanner

— Begin of "memscan.c" —

```
/* Memory integrity scanner
 *
 * Compile:
 * gcc -o memscan memscan.c
 *
 * Author: Andre Almeida
 *
 */

#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <errno.h>
#include <time.h>
```

```
//elf includes
#include <link.h>
#include <elf.h>
#include <sys/mman.h>

//ptrace includes
#include <sys/ptrace.h>

//dir includes
#include <dirent.h>
#include <ctype.h>

#define HEXSIZE        8
#define MAX_PATH_LEN     128
#define PROCKALLSYMS "/proc/kallsyms"
#define MAPS_REFIX       "/proc/"
#define MAPS_SUFFIX      "/maps"

#ifdef __FreeBSD__
        #define ELF(x) Elf_##x
#else
        #define ELF(x) ElfW(x)
#endif

void* mapfile(char* path, unsigned long* size, int *fd_no)
{
        struct stat s;
        FILE* f;

        f = fopen(path, "r");
        fstat(fileno(f), &s);
        *size = s.st_size;
        *fd_no = fileno(f);
        return mmap(NULL, s.st_size, PROT_READ, MAP_PRIVATE, *fd_no, 0);
}

int memscan_pid(void* base, unsigned long begin_addr, pid_t pid)
{
        unsigned int i, ec, symcount, count, res = 1;
        char* codePos;
        unsigned long* bytecode;
        unsigned long peekbyte;
        ELF(Ehdr)* e = base;
        ELF(Shdr)* shdr = (ELF(Shdr)*)((char*)e + e->e_shoff);
        ELF(Sym)* sym = 0;
        Elf32_Word nameIndex;
```

```
Elf32_Word strtab_size;
char* strtab = 0;
count = 0;

for(i = 0; i < e->e_shnum; i++) {
        if(shdr->sh_type == SHT_SYMTAB) {
                sym = (ELF(Sym)*)((char*)e + shdr->sh_offset);
                symcount = shdr->sh_size / sizeof(ELF(Sym));
        }
        else if(shdr->sh_type == SHT_STRTAB){
                if(count == 1){
                        strtab = (char*)((char*)e + shdr->sh_offset);
                        strtab_size = (Elf32_Word)((char*)e + shdr->sh_size);
                        break;
                }
                count++;
        }
        shdr++;
}
shdr = (ELF(Shdr)*)((char*)e + e->e_shoff);

for(i = 0; i < e->e_shnum; i++) {
        if(!strcmp(".text", &strtab[shdr->sh_name])){

                codePos = (char*)e + shdr->sh_offset;
                for(ec = 0; ec < shdr->sh_size/4; ec+=1){
                        bytecode = (unsigned long*) codePos+ec;

                        peekbyte = ptrace(PTRACE_PEEKTEXT,
                                        pid, begin_addr + shdr->sh_offset + ec*4, 0);
                        if(errno == ESRCH){ //are we tracing a non-existent process?
                                perror("PTRACE_PEEKTEXT");
                                return 0;
                        }

                        if(*bytecode != peekbyte){
                                printf("\nWARNING - Offset: 0x%.8x: 0x%.8x (DISK) != 0x%.8x (MEM)!",
                                        begin_addr + shdr->sh_offset + ec*4,
                                  *bytecode, peekbyte);
                                res = 0;
                        }
                }
                break;
        }
        shdr++;
}
```

```
        return res;
}

int memscan(pid_t pid, unsigned long begin_addr, unsigned long end_addr, char* file_path){
        unsigned long size;
        Elf32_Addr offset;
        void* handle;
        int fd_no, res = 0;
        if((handle = (void*)mapfile(file_path, &size, &fd_no)) == MAP_FAILED){
                perror("mmap");
                res = 0;
        }else{
                res = memscan_pid(handle, begin_addr, pid);
                munmap(handle, size);
                close(fd_no);
        }

        return res;
}

unsigned long proc_verify(pid_t pid){
        char addresses[HEXSIZE*2+1];
        char *end_addr_str;
        char permissions[MAX_PATH_LEN+1];
        char current_file_path[MAX_PATH_LEN+1];
        char maps_path[32];
        unsigned long begin_addr, end_addr;
        int res = 1, retres = 1;
        FILE *mapfile;

        sprintf(maps_path, "%s%d%s", MAPS_REFIX, pid, MAPS_SUFFIX);
        mapfile = fopen(maps_path, "r");

        while(fscanf(mapfile,"%s %s %*s %*s %*s", addresses, permissions) != -1){
                while(fread(current_file_path, 1, 1, mapfile) != 0){
                        current_file_path[1] = '\0';

                        if(strcmp(current_file_path, "\n") == 0){
                                current_file_path[0] = '\0';
                                break;
                        }

                        if(strcmp(current_file_path, "/") == 0){
                                //This line has a slash, the executable exists, so rewind and fscanf
                                fseek(mapfile, -1, SEEK_CUR);
                                fscanf(mapfile,"%s\n", current_file_path);
```

```
                        break;
                }
        }
        if(strcmp(permissions, "r-xp") == 0 && current_file_path[0] != '\0') {
                addresses[HEXSIZE] = '\0';
                begin_addr = strtoul(addresses, NULL, 16);
                end_addr = strtoul(addresses + HEXSIZE + 1, NULL, 16);

                //Do the scan
                res = memscan(pid, begin_addr, end_addr, current_file_path);
                if(!res){
                        printf("\nObject \"%s\" modified in memory!", current_file_path);
                }
                if(!res && retres)
                        retres = 0;

        }
    }

    fclose(mapfile);

  return retres;
}

int pid_scan(pid_t pid){
        int status, res = 0;
        if (ptrace(PTRACE_ATTACH, pid, NULL, NULL) != 0){
                perror("\nPTRACE_ATTACH");
                return res;
        }

        wait(&status);

        res = proc_verify(pid);

        if (ptrace(PTRACE_DETACH, pid, NULL, NULL) != 0){
                perror("\nPTRACE_DETACH");
                res = 0;
        }

        return res;
}

void scan_tasks(){
        DIR * dir, *rdir;
        int pid;
```

```
        struct dirent * entry, *rentry;
        char file[20];
        char directory[64];
        char rfile[64];
        time_t mytime;
        struct tm *brokentime;

        if((dir = opendir("/proc")) == NULL){
                perror("opendir /proc :");
                return;
        }
        else{
                while((entry = readdir (dir)) != NULL){
                        if(isdigit(entry->d_name[0]) != 0){
                                pid = atoi(entry->d_name);
                                if(pid != 1 && pid != getpid()){
                                        time(&mytime);
                                        brokentime = (struct tm*)localtime(&mytime);

                                        printf("[%.2d:%.2d:%.2d] Scanning pid %d... ", brokentime->tm_hour,
                                                brokentime->tm_min, brokentime->tm_sec ,pid);
                                        fflush(stdout);
                                        if(pid_scan(pid))
                                                printf("OK\n");
                                        else
                                                printf("\nFailed\n");
                                }
                        }
                }
        }
}

int main(int argc, char **argv){
        pid_t pid;
        if(argc == 2){
                pid = atoi(argv[1]);
                printf("Scanning pid %d... ", pid);
                fflush(stdout);
                if(pid_scan(pid))
                        printf("OK\n");
                else
                        printf("\nFailed\n");
        }else
                scan_tasks();
        return 0;
}
```

— End of “memscan.c” —

Appendix C

HWBP detection and prevention module

— End of “ptracemod.c” —

```
/*
 * Hardware breakpoint hooking detection and prevention module.
 *
 * Author: Andre Almeida
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/syscalls.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <linux/kallsyms.h>
#include <linux/errno.h>

#include <asm/i387.h>

#include "ptrace.h"

#define AUTHOR "Andre Almeida"
#define VERSION "v1.0"

#define ARCH 0
#define PROCKALLSYMS "/proc/kallsyms"
#define MOD_NAME "HWBP Defender"
#define HEXSIZE 8
#define KMEMBASE 0xc0000000


/* wait macros */
#  define __WAIT_INT(status) (status)
#define __WIFSTOPPED(status) (((status) & 0xff) == 0x7f)
#define __WSTOPSIG(status) __WEXITSTATUS(status)
#define __WEXITSTATUS(status) (((status) & 0xff00) >> 8)
# define WIFSTOPPED(status) __WIFSTOPPED(__WAIT_INT(status))
# define WSTOPSIG(status) __WSTOPSIG(__WAIT_INT(status))
/* end of wait macros */

unsigned long bp_addr;
```

```
int dr7_set;
pid_t watch_pid = -1, target_pid = -1;
static void **sys_call_table; /* kernel sys_call_table */

ulong get_eip(struct task_struct*);

/* code to find the system call table from SuckIT Rootkit */
struct idtr {
        ushort  limit;
        ulong   base;
} __attribute__ ((packed));
struct idt {
        ushort  off1;
        ushort  sel;
        u_char   none, flags;
        ushort  off2;
} __attribute__ ((packed));

/* from SuckIT */
void *memmem(char *s1, int l1, char *s2, int l2) {
        if (!l2)
                return s1;
        while (l1 >= l2)
        {
                l1--;
                if (!memcmp(s1,s2,l2)){
                        return s1;
                                }
                s1++;
        }
        return(NULL);
}

/* from SuckIT */
ulong   get_sct(ulong ep) {
        #define SCLEN 512
        char code[SCLEN];
        char *p;
        ulong r;
        memcpy(&code, (void *)ep, SCLEN);
        p = (char *) memmem(code, SCLEN, "\xff\x14\x85", 3);
        if (!p)
                return 0;
        r =  *(ulong *) (p + 3);
        return r;
}
```

```
/* from SuckIT */
/*  even if the kernel uses the sysenter fast call to process system calls,
 *  this will work too because the kernel code for the int 80 stub is there! */
static u_long locate_sys_call_table(void) {
        struct idtr idtr;
        struct idt idt80;
        ulong old80, sct, offp;
        asm ("sidt %0" : "=m" (idtr));
        offp = idtr.base + (0x80 * sizeof(idt80));
        memcpy(&idt80, (void *)offp, sizeof(idt80));
        old80 = idt80.off1 | (idt80.off2 << 16);
        sct = get_sct(old80);
        return(sct);
}

/* get task_struct of a process by pid  */
struct task_struct *get_task_from_pid(pid_t pid){
        struct task_struct *task;

        for_each_process(task)
                if (task->pid == pid)
                        return task;
        return NULL;
}

/* sys_waitpid */
static asmlinkage long (*orig_wait4) (pid_t pid, int __user *status,
        int options, struct rusage __user *ru);
static asmlinkage long our_wait4(pid_t pid, int __user *status,
        int options, struct rusage __user *ru){
        struct task_struct *task;
        pid_t retpid;
        ulong eip;
        retpid = orig_wait4(pid, status, options, ru);
        eip = 0;

        if(WIFSTOPPED(*status)){
                if(WSTOPSIG(*status) == SIGTRAP){ /* it is a breakpoint trap */
                        if(retpid != -1){ //wait4 did not return error?
                                if((task = get_task_from_pid(retpid)) != NULL){
                                        eip = get_eip(task);

                                        if((task->thread.debugreg[6] & 0x00000001) == 0x00000001 ||
                                                (task->thread.debugreg[6] & 0x00000002) == 0x00000002 ||
                                                (task->thread.debugreg[6] & 0x00000004) == 0x00000004 ||
```

```
                                        (task->thread.debugreg[6] & 0x00000008) == 0x00000008){
                                        /*  Hardware Breakpoint triggered! */
                                        /* state one! */
                                        bp_addr = eip;
                                        watch_pid = current->pid;
                                        target_pid = retpid;
                                        printk("HWBP triggered on pid %d, by pid %d\n",
                                                retpid, current->pid);
                                }
                        }
                }
        }
        return retpid;
}

/* sys_ptrace */
static asmlinkage long (*orig_ptrace) (enum __ptrace_request request, pid_t pid,
                void *addr, void *data);
static asmlinkage long our_ptrace(enum __ptrace_request request, pid_t pid,
                void *addr, void *data){
        struct user_regs_struct *regs;
        struct task_struct *task;
        int abort = 0, ret = -1;
        ulong eip;

        if(request == PTRACE_SETREGS){
                regs = (struct user_regs_struct*)data;
                if(watch_pid == current->pid && target_pid == pid) {
                        if((task = get_task_from_pid(pid)) != NULL){
                                eip = get_eip(task);
                                if(eip == bp_addr && regs->eip != bp_addr){
                                        /* state two! warning! */
                                        abort = 1;
                                        task->thread.debugreg[0] = 0x0;
                                        task->thread.debugreg[7] = 0x0;
                                        printk("WARNING: PID: %d - HWBP ATTEMPT on process %d!\n",
                                                current->pid, pid);
                                }
                        }
                }
        }

        /* return to state zero */
        if(request == PTRACE_CONT){
                if(watch_pid == current->pid){
```

```
                        watch_pid = -1;
                        target_pid = -1;
                }
        }

        if(!abort)
                ret = orig_ptrace(request, pid, addr, data);
        return ret;
}

void sys_call_hook(void){
        /*  hook sys_ptrace system call */
        orig_ptrace= sys_call_table[__NR_ptrace];
        sys_call_table[__NR_ptrace] = our_ptrace;

        /*  hook sys_wait4 system call */
        orig_wait4 = sys_call_table[__NR_wait4];
        sys_call_table[__NR_wait4] = our_wait4;
}

/* unhooks the syscalls */
void sys_call_unhook(void){
        sys_call_table[__NR_ptrace] = orig_ptrace;
        sys_call_table[__NR_wait4] = orig_wait4;
}

void init_vars(void){
        bp_addr = -1;
        dr7_set = 0;
}

/* begin kernel code of ptrace.c
  * this code is needed to find the EIP of an address */
static inline int get_stack_long(struct task_struct *task, int offset)
{
        unsigned char *stack;

        stack = (unsigned char *)task->thread.esp0 - sizeof(struct pt_regs);
        stack += offset;
        return (*((int *)stack));
}

static unsigned long my_getreg(struct task_struct *child,
        unsigned long regno)
{
        unsigned long retval = ~0UL;
```

```
        switch (regno >> 2) {
                case GS:
                        retval = child->thread.gs;
                        break;
                case DS:
                case ES:
                case FS:
                case SS:
                case CS:
                        retval = 0xffff;
                        /* fall through */
                default:
                        if (regno > FS*4)
                                regno -= 1*4;
                        retval &= get_stack_long(child, regno);
        }
        return retval;
}
/* end kernel code of ptrace.c */

ulong get_eip(struct task_struct *task){
        ulong eip_offset, eip;
        eip_offset = offsetof(struct user_regs_struct, eip);
        eip = my_getreg(task, eip_offset);
        return eip;
}

int init_module(void){
        u_long sct_addr;
        printk("HWBP defender: Started.\n");

        sct_addr = locate_sys_call_table();
    if (!sct_addr) {
            printk("Cannot find sys_call_table. Aborting\n");
            return 0;
    }
    sys_call_table = (void *)sct_addr;

        printk("sys_call_table @ %p\n", sys_call_table);

        init_vars();

        /* hooks the syscalls */
        sys_call_hook();
  return 0;
```

```
}

void cleanup_module(void){
        /* dont forget to unhooks the syscalls */
        sys_call_unhook();
        printk("HWBP defender: Stopped.\n");
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR(AUTHOR);
MODULE_DESCRIPTION(VERSION);
```

Appendix D

Proactive Detection System

— Begin of "sctdefender.c" —

```
/*
 * Author: Andre Almeida
 * Special thanks to Daniel Almeida
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/unistd.h>
#include <linux/syscalls.h>
#include <linux/proc_fs.h>
#include <asm/uaccess.h>
#include <linux/kallsyms.h>
#include "main.h"

#define AUTHOR  "Andre Almeida"
#define VERSION "Syscall defender"

/* pointers the our sct tables  */
void **my_sys_call_table; /* our copy of the original sys_call_table */
void **sys_call_table; /* kernel sys_call_table */
void **correct_sys_call_table; /* correct kernel sys_call_table
        after proactive defense system installed */

typedef int (*readdir_t)(struct file *, void *, filldir_t);
static ulong find_int80_handler(void);

unsigned long *kernel_syscall_jmp;
unsigned long stubfunc_shift = 0;
```

```
unsigned long kernel_code_begin, kernel_code_end, sysenter_entry_addr, system_call_addr;

struct stack_frame {
        struct stack_frame *next_frame;
        unsigned long return_address;
};

/* get's symbols from /proc/kallsyms USERSPACE */
unsigned long get_kernel_symbol(char *symbol_name){
  int fd,i;
  char values[HEXSIZE+1];
  char type[3+1];
  char name[KSYM_NAME_LEN+1]; /* max kernel symbol name length */

  mm_segment_t old_fs = get_fs();
  set_fs(KERNEL_DS);
  fd = sys_open(PROCKALLSYMS, O_RDONLY, 0);

  if (fd >= 0) {
    /* string parsing */
    while(sys_read(fd, values,HEXSIZE) == HEXSIZE){
      sys_read(fd, type,3);
      for(i=0; i<KSYM_NAME_LEN; i++){
        sys_read(fd,name+i,1);
        if( name[i] == '\n'){
          name[i] = '\0';
          break;
        }
      }
      /* there's a match */
      if(!strcmp(symbol_name,name)){
        sys_close(fd);
        set_fs(old_fs);
        return simple_strtoul(values,NULL,16);
      }
      memset(values,'\0',HEXSIZE+1);
      memset(type,'\0',4);
      memset(name,'\0',KSYM_NAME_LEN+1);
    }
    sys_close(fd);
  }
  set_fs(old_fs);

  /* symbol not found */
  return 0;
}
```

```
int verify_sysenter(void) {
        unsigned long sysenter_addr;
        asm("movl $0x176, %%ecx;"
                "rdmsr;"
                "movl %%eax, %0;"
                : "=r" (sysenter_addr) : );

                return (sysenter_addr == sysenter_entry_addr);
}

static int verify_int80(void){
        ulong int80_addr = find_int80_handler();
        return (int80_addr == system_call_addr);
}

static int verify_syscall_jmp(void){
        return (*kernel_syscall_jmp == sys_call_table);
}

int verify_table_entry(int sys_call_number){
  return (*(sys_call_table + sys_call_number) == *(correct_sys_call_table + sys_call_number));
}

int verify_inline(int sys_call_number){
        char *memp;
        memp = *(correct_sys_call_table + sys_call_number);
        return(*memp != 'E9');
}

/* from SuckIT */
struct idtr {
       ushort  limit;
       ulong   base;
} __attribute__ ((packed));
struct idt {
       ushort  off1;
       ushort  sel;
       u_char   none, flags;
       ushort  off2;
} __attribute__ ((packed));

/* from SuckIT */
void *memmem(char *s1, int l1, char *s2, int l2) {
       if (!l2)
               return s1;
```

```
        while (l1 >= l2)
        {
                l1--;
                if (!memcmp(s1,s2,l2)){
                        return s1;
                                }
                s1++;
        }
        return(NULL);
}

static ulong find_int80_handler(void) {
        struct idtr idtr;
        struct idt idt80;
        ulong old80, offp;
        asm ("sidt %0" : "=m" (idtr));
        offp = idtr.base + (0x80 * sizeof(idt80));
        memcpy(&idt80, (void *)offp, sizeof(idt80));
        old80 = idt80.off1 | (idt80.off2 << 16);
        return old80;
}

/* from SuckIT */
ulong   get_sct(ulong ep) {
        #define SCLEN 512
        char code[SCLEN];
        char *p;
        ulong r;
        memcpy(&code, (void *)ep, SCLEN);
        p = (char *) memmem(code, SCLEN, "\xff\x14\x85", 3);
        if (!p)
                return 0;
            kernel_syscall_jmp = ep + (p + 3) - code;
        r =  *(ulong *) (p + 3);
        return r;
}

/* from SuckIT */
/*  even if the kernel uses the sysenter fast call to process system calls,
        this will work too because the kernel code for the int 80 stub is there! */
static u_long locate_sys_call_table(void) {
        ulong sct;
        sct = get_sct(find_int80_handler());
        return(sct);
}
```

```
int inside_kernel_code(unsigned long addr){
        if(addr >= kernel_code_begin && addr <= kernel_code_end)
                return 1;
        else
                return 0;
}

int find_kernel_code_limits(void){
        kernel_code_begin = get_kernel_symbol("_stext");
        kernel_code_end = get_kernel_symbol("_etext");
        if(kernel_code_begin == 0 || kernel_code_end == 0)
                return 0;
        else
                return 1;
}

static inline int my_valid_stack_ptr(struct thread_info *tinfo, void *p, unsigned size)
{
        return  p > (void *)tinfo &&
                p <= (void *)tinfo + THREAD_SIZE - size;
}

static inline unsigned long print_context_stack(struct thread_info *tinfo,
                                unsigned long ebp,
                                int sys_call_number)
{
#ifdef  CONFIG_FRAME_POINTER
        int stack_depth = 0;
        unsigned long addr_shift;
        struct stack_frame *frame = (struct stack_frame *)ebp;
        while (my_valid_stack_ptr(tinfo, frame, sizeof(*frame))) {
                struct stack_frame *next;
                unsigned long addr;

                addr = frame->return_address;
                next = frame->next_frame;

                switch(stack_depth){
                        case 0:
                                addr_shift = addr - (unsigned long)*(correct_sys_call_table +
                                        sys_call_number);
                                if(stubfunc_shift == 0)
                                        stubfunc_shift = addr_shift;
                                else if(addr_shift != stubfunc_shift)
                                        printk("WARNING! Difference between: 0x%x and 0x%x!\n",
                                                stubfunc_shift, addr_shift);
```

```
                        break;
                        case 1:
                                if(!inside_kernel_code(addr))
                                        printk("WARNING! 0x%x should be inside kernel code!\n", addr);
                        break;
                        default:
                                printk("WARNING! Call stack too big, something is wrong!\n");
                        break;
                }

                if (next <= frame)
                        break;
                frame = next;
                stack_depth++;
        }
#endif
        return ebp;
}

void check_stack(unsigned long stack_pointer, unsigned long start_ebp, unsigned long sys_call_number){
        unsigned long ebp = 0;
        unsigned long stack;
        stack = stack_pointer;
        ebp = start_ebp;

        while (1) {
                struct thread_info *context;
                context = (struct thread_info *)
                        ((unsigned long)stack & (~(THREAD_SIZE - 1)));
                ebp = print_context_stack(context, ebp, sys_call_number);
                stack = (unsigned long)context->previous_esp;
                if (!stack)
                        break;
        }
}

/*  here is where the system gets inspected */
void syscheck(unsigned long addr, int sys_call_number,
        unsigned long stack_pointer, unsigned long start_ebp){

  if(!verify_table_entry(sys_call_number))
        printk("WARNING! Syscall %d hijacked!\n", sys_call_number);
#ifdef  CONFIG_FRAME_POINTER
  check_stack(stack_pointer, start_ebp, sys_call_number);
#endif
```

```
  if(!verify_sysenter())
        printk("WARNING: IA32_SYSENTER_EIP hijacked!\n");

  if(!verify_int80())
        printk("WARNING: int80 handler hijacked!\n");

  if(!verify_syscall_jmp())
        printk("WARNING: system call jump modified!\n");

  if(!verify_inline(sys_call_number))
        printk("WARNING: jmp at the beginning of the syscall number %d!\n", sys_call_number);
}

unsigned long process_call(int call_number){
        unsigned long jmp_syscall, stack_pointer, start_ebp;
        asm ("movl %%ebp, %0" : "=r" (start_ebp) : );

        jmp_syscall = (unsigned long)*(my_sys_call_table + call_number);
        asm("movl %%esp, %0" : "=r" (stack_pointer) : );

        //syscall verification goes here!
        syscheck(jmp_syscall, call_number, stack_pointer, start_ebp);
        return jmp_syscall;
}

void sys_call_unhook(void){
  //restore to the original table
  int i;
  for(i = 0; i < SCT_SIZE; i++){
        sys_call_table[i] = my_sys_call_table[i];
  }
}

void clone_table(void){
        my_sys_call_table = kmalloc(SYS_CALL_TABLE_SIZE, GFP_KERNEL);
        memcpy(my_sys_call_table, sys_call_table, SYS_CALL_TABLE_SIZE);
}

int sys_call_table_scan(void){
        int i, valid = 1;
        for(i = 0; i < SCT_SIZE; i++){
                if(!inside_kernel_code((unsigned long)*(sys_call_table + i))){
                        printk("Function %d not inside kernel code (0x%.8x) !\n",
                                i, (unsigned long)*(sys_call_table + i));
                        valid = 0;
                }
```

```
        }
        return valid;
}

int init_module(void){
        u_long sct_addr;
        int i;
        printk("SCT Defender Loaded!...\n");

        sysenter_entry_addr = get_kernel_symbol("sysenter_entry");
        system_call_addr = get_kernel_symbol("system_call");

        if(!verify_int80())
                printk("WARNING: int80 handler hijacked!\n");
        else
                printk("Interrupt 80 OK\n");

        if(!verify_sysenter())
                printk("WARNING: Sysenter instruction hijacked!\n");
        else
                printk("Sysenter instruction OK\n");

        if(!find_kernel_code_limits()) {
                printk("kernel code limits not found!\n");
                return -1;
        }

        sct_addr = locate_sys_call_table();
    if (!sct_addr) {
        printk("Cannot find sys_call_table. Aborting\n");
        return 0;
        }
        sys_call_table = (void *)sct_addr;

        printk("SCT Defender: sys_call_table at %p\n", sys_call_table);

        printk("SCT Defender: verifying sys_call_table integrity...\n");
        if(!sys_call_table_scan())
                printk("Failed!!\n");
        else
                printk("Done.\n");

        clone_table();

        correct_sys_call_table = kmalloc(SYS_CALL_TABLE_SIZE, GFP_KERNEL);
        sys_call_hook(correct_sys_call_table, sys_call_table);
```

```
        if(!verify_syscall_jmp())
                printk("WARNING: system_call_jump modified!\n");
        else
                printk("System_call_jump_OK\n");

        for(i = 0; i<255; i++){
                if(!verify_inline(i))
                        printk("Inline hook found in system call %d!\n", i);
        }

        printk("SCT Defender: Active.\n");

        return 0;
}

void cleanup_module(void){

  /* unhooks the syscalls */
  sys_call_unhook();
  printk("SCT Defender exiting...\n");
}

MODULE_LICENSE("GPL");
MODULE_AUTHOR(AUTHOR);
MODULE_DESCRIPTION(VERSION);
```

— End of “sctdefender.c” —

— Begin of “sctdefender.h” —

```
#ifndef __MAIN__
#define __MAIN__

#define PROCKALLSYMS "/proc/kallsyms"
#define MODNAME      "sctdefender"
#define SCT_SIZE          256

#define HEXSIZE       8
#define SYS_CALL_TABLE_SIZE SCT_SIZE * 4

void sys_call_hook(void **, void**);
unsigned long process_call(int call_number);

#ifdef  CONFIG_FRAME_POINTER
```

```
#define STUB_CODE(sys_call_number) \
        unsigned long jmp_syscall = process_call(sys_call_number); \
        asm("movl %0, %%edx" : : "r" (jmp_syscall)); \
        asm("mov %ebp, %esp;" \
                "pop %ebp"); \
        asm("jmp *%edx");
#else
#define STUB_CODE(sys_call_number) \
        unsigned long jmp_syscall = process_call(sys_call_number); \
        asm("movl %0, %%edx" : : "r" (jmp_syscall)); \
        asm("jmp *%edx");
#endif

#endif // __MAIN__
```

— End of “sctdefender.h” —

— Begin of “syscalls.c” —

```
#include <asm/uaccess.h>
#include "main.h"

static asmlinkage void syscall_stub0(void){ STUB_CODE(0); }
static asmlinkage void syscall_stub1(void){ STUB_CODE(1); }
static asmlinkage void syscall_stub2(void){ STUB_CODE(2); }
... (repeated until 255) ...
static asmlinkage void syscall_stub255(void){ STUB_CODE(255); }

void sys_call_hook(void **sys_call_table, void **correct_sys_call_table){
        int i;

        correct_sys_call_table[0] = syscall_stub0;
        correct_sys_call_table[1] = syscall_stub1;
        correct_sys_call_table[2] = syscall_stub2;
... (repeated until 255) ...
        correct_sys_call_table[255] = syscall_stub255;
        //sys_call_table[24] = correct_sys_call_table[24];
        for(i = 0; i < SCT_SIZE; i++) {
                sys_call_table[i] = correct_sys_call_table[i];
        }
}
```

— End of “syscalls.c” —

Glossary

API Application Programming Interface

CPU Central Processing Unit

DKOM Direct Kernel Object Manipulation

DLL Dynamic Link Library

DMA Direct Memory Access

DR0-7 Debug Registers (0 to 7)

DSO Dynamic Shared Objects

EIP Extended Instruction Pointer

ELF Executable and Linkable Format

GOT Global Offset Table

HIDS Host-Based Intrusion Detection System

HIPS Host-Based Intrusion Prevention System

IAT Import Address Table

IDS Intrusion Detection System

IDT Interrupt Descriptor Table

IDTR Interrupt Descriptor Table Register

IRP IO Request Packet

ISR Interrupt Service Routine

LKM Linux Kernel Module

OS Operating System

PCI Peripheral Component Interconnect

PEB Process Environment Block

PID Process ID

PLT Procedure Linkage Table

SEH Structured Exception Handler

SSDT System Service Dispatch Table

SVV System Virginity Checker

VFS Virtual File System

VMA Virtual Memory Areas