ITEC851 – Assignment 2

Nicolas CORMIER – 41350790

Macquarie University – Semester 2 2008

Kernel Based Malicious Code: Kernel Hacking

The main agenda of this document is to expose and compare the techniques and mechanisms used by kernel operating malicious software. In term of malicious code, there are two main benefits in operating within the kernel rather than as a common user application. Firstly, it enables attackers to access fairly more functionalities than they could access in user land. Secondly, it is much more difficult for counter measures applications to detect malicious activity within the kernel: the main option for user-land malicious code to alter their environment is by exploiting vulnerabilities in the APIs. Consequently, anti malware applications monitor first and foremost for APIs misusage. In fact, by operating within the kernel, the attackers limit significantly their detection surface.

It is fairly important to highlight that this document does only cover kernel targeted attacks. Malicious code applications may use the kernel as a vector for other kinds of attacks - like user-land or hardware attacks. However, we deliberately limit our study to malicious code that aims to attack the kernel by patching its runtime image.

Note: we define "kernel based malicious code" as a third-party malicious code that operates as any kernel routine. Which means full access to the whole kernel virtual address space image and running in kernel mode.

We can distinguish two well spread techniques used by kernel based malicious code: *Hooking* and *DKOM* (Direct Kernel Object Manipulation). Both patch the kernel memory, but for two different purposes: the *Hooking* technique aims to modify the control flow in order to extend or decrease the functionality of a routine while trying to preserve as much as possible the original behavior. It is basically used to monitor and control inputs and outputs of targeted kernel routines. The *DKOM* technique is used to directly modify private kernel data in order to modify its behavior. It is basically used for adding or removing dynamic information like processes or drivers in order to fake the user or the system.

The first part of the document exposes the *Hooking* and *DKOM* techniques on Windows and Unix. The second part examines how malicious codes are injected into the kernel image on the same operating systems. Finally, the last part extrapolates these techniques to the VMS operating system.

I. Hooking

As stated before, the *Hooking* technique aims to alter a control flow of a specific routine in order to filter or control its behavior. To illustrate the concept, we tried to implement two kernel based hooks on each operating system: a system call hook and an exception hook. Both aim to acquire a control flow from the following user land actions: system calls or exception raises.

I.1. *Hooking* - Windows

The Windows system call hooking example (*src.tgz:windows_kernel/ssdt_hooking/*) is based on the SSDT (System Service Descriptor Table) patching. This table contains the routines exported by the kernel. When a user wants to call a system routine, it basically uses an identifier, which refers to an entry in this table. By patching this table, we can replace an existing entry by our malicious code.

I.2. *Hooking* - Unix

The Unix system call hooking example (*src.tgz:freebsd_kernel/syscall_hooking*) uses basically the same notions as the previous example. FreeBSD - and all the Unixes in general - uses a syscall table to associate an identifier with a routine address. When a user process interrupts to call a system routine, it queries it by using an identifier - just like Windows. By patching the syscall table, we can replace an existing entry by our own code.

I.4. *Hooking* - Bonus

Hooking technique is not limited to kernel patching and can be used in a broad range of other domains. To illustrate this we also provide an IDT hooking example (*src.tgz:{windows,freebsd}_kernel/idt_hooking*) for Windows and Unix as well as a user process hooking example on Windows using IAT (*src.tgz:windows_kernel/iat_hooking)*. The IDT can be considered more as a "hardware" hooking technique while the IAT is a user land process hooking technique. The IAT is used on Windows as a "dynamic" routines binding: when an application - that uses shared libraries - is compiled, there is no clean way to guess the addresses of the needed shared routines. The IAT is the answer to this problem: instead of calling directly a hardcoded address, the call is made on an address read at run-time from the IAT. At the application launch, the loader is responsible for filling the IAT in function of where the shared libraries have been mapped. It is possible to hook one of these entries in order to get control of the user application. A same technique can be used on Unix by hooking the PLT.

II. *DKOM*

As stated before, *DKOM* technique aims to modify the kernel's private data in order to alter its behavior. To illustrate the concept, the two following examples propose to modify the kernel's private data in order to hide a process.

II.1. *DKOM* - Windows

The example (*src.tgz:windows_kernel/dkom*) modifies the double linked list used by Windows to store all the active processes.

II.2. *DKOM* - Unix

The example (*src.tgz:freebsd_kernel/dkom*) modifies the double linked list used by FreeBSD to store all the processes.

III. Main Door

All the techniques described before require an access to the kernel private memory. In this section we will compare the different mechanisms available that enable third-party developers to modify the operating system kernel image by adding their own piece of code directly into it.

III.1. Main Door - Windows

By using the Windows DDK, it is possible to inject executable code within the kernel image. However, Windows tries to prevent drivers as much as possible to play with its data by obfuscating the accesses to them. This last chance control is not sufficient and there is a real security issue behind this design. *Note: Theoretically, only administrators group can insert new drivers into the system.*

III.2. Main Door - Unix

Like Windows, Unixes (BSD, Linux, Solaris) propose the same type of mechanism to extend the kernel functionalities. But unlike Windows, most of them don't even try to obfuscate the access to kernel's private data. As we saw in the previous examples, patching the syscall table on FreeBSD is really trivial. Here again, there is a real security issue with this design choice. *Note: Theoretically, only root user can insert a new kernel module into the FreeBSD system.*

III.3. Main Doors

Probably all the "serious" commercial operating systems have set up and built a broad range of controls in order to protect their kernel from the outside attacks. The paradox is that most of them also propose - by default - a way to by pass all theses controls by letting third-party code operate inside the kernel rather than outside. In term of security, there is definitely an issue with this design choice.

IV. Back Door

A significant benefit can be gained from exploiting an unexpected entrance to the kernel. In fact, usually all the eyes are turned to the main door, which makes it tough for the attacker to use it successfully. In this section, we expose a practical technique to illustrate this concept. The following example enables a common user to execute privilege code (ring 0) without using the operating system's driver/kernel module API. This example is mainly based on IA32 features: it consists in modifying the GDT in order to install a call-gate for executing ring 0 code from a ring 3 application. However, it requires from the operating system to expose a mechanism to read and write the physical memory from user land.

The following process explains how the technique operates:

1. Acquire an access to the computer's physical memory

2. Get the GDT virtual address (in the current user process context just by calling the *sgdt* instruction)
3. By analyzing the current process address translation context, guess the physical address of the GDT
4. Map the GDT into our process (by this way we by pass the memory protection of the GDT)
5. Modify the GDT by inserting a call gate
6. Call the call gate

IV.1. Back Door - Windows

Windows proposes a mechanism to access and modify the physical memory through the virtual device *\Device\PhysicalMemory*. The example (src.tgz:windows_kernel/ring0_without_driver) is the implementation of this technique on Windows. However, the user has to be administrator of the computer to use it.

IV.2. Back Door - Unix

The same technique can be used on IA32/Unixes like Linux (/dev/kcore), Solaris and FreeBSD.

It is not unusual to find custom drivers installed on Unixes systems that enable a full access to the physical memory.

V. The VMS Case

The previous sections cover two aspects regarding kernel based malicious code: how the code is injected within the kernel and how the code operates once injected. This section discusses these two aspects applied to the OpenVMS operating system.

Note to the lecturer: I didn't have the time to realize practical tests to validate what I have read/understood of the VMS device driver operating model. Most of the gathered information comes from "Writing OpenVMS Alpha Device Drivers in C", the OpenVMS C documentation and other materials that can be found in the Reference section.

V.1. Main & Back Doors - VMS

VMS, like Windows and Unixes, proposes a set of APIs and tools that enable third-party developers to inject code - in form of device drivers - within the VMS kernel virtual address space. In addition, again like Windows and Unixes, device drivers run in kernel mode.

It is important to note that VMS provides a set of kernel APIs, which propose to create Kernel Processes that can be really useful to create malicious code.

The tool to install a new device driver on OpenVMS is called *SYSMAN* but system service calls can also be used to register and load a driver (SYS$LOAD_DRIVER). Regarding back doors, it seems that no exploit has permitted yet to install device drivers as a default user on the default VMS configuration. However, as Windows and Unixes, it is possible to create and install a device driver that accesses and modifies - for example - the physical memory from the user land as seen before. But, unlike Windows and Unixes, this kind of access is not provided by default with the system.

V.2. Hooking and DKOM - VMS

Both techniques - Hooking and DKOM - are based on the kernel run-time image patching. As seen before, it is possible to operate within the OpenVMS kernel and as a result to access its private data. However, it seems that - like Windows - OpenVMS does not expose most of its data to device drivers. In addition, the OpenVMS source code is not available to anyone - like Windows. Regarding other hooking techniques like IDT, it is theoretically possible to hook interrupt handlers because device drivers run in kernel mode and consequently can use privilege CPU instructions.

V.3. VMS as Secure as Windows and Unixes?

Once the malicious code is injected within the kernel, VMS is as vulnerable as Windows and Unixes. However, it is fairly important to highlight that the access to the VMS kernel is far more difficult to reach - for an attacker - than on the other operating systems: on windows and Unixes, it is possible to exploit a vulnerability - like a buffer overflow - in a root running service or daemon and consequently obtain a root access to the computer. Which basically enables the attacker to do everything they want on the system and for example install a malicious device driver. Whereas, VMS requires from the server administrator to deliberately install a malicious driver because the design of the operating system does not allow privilege escalation.

VI. Conclusion

Kernel based malicious code is a real threat and needs to be managed as any operating system risk. In addition, on server environment, dynamic kernel module loading is usually unnecessary and a simple counter measure for this risk consists in just disabling it. In fact, most of the operating systems listed before propose an option to disable dynamic module loading after the operating system boot and it is also possible on most Unixes to build-in all the needed device drivers within the kernel and just completely disable dynamic kernel module loading.

Once injected, a kernel based malicious code may access and modify the whole operating system behavior and it is fairly more difficult to detect and heal an infected kernel than a user land service or application.

Bibliography

Goldenberg, R. E. *OpenVMS Alpha Internals and Data Structures* .

HP. (n.d.). *VMS Faq*. Retrieved from http://h71000.www7.hp.com/faq/vmsfaq_003.html

International, V. (n.d.). *Getting Serious About VMS Hacking*. Retrieved from http://phrack.org/issues.html?issue=23&id=8

kad. (n.d.). *Handling Interrupt Descriptor Table for fun and profit.* Retrieved from http://www.theparticle.com/files/txt/hacking/phrack/p59-0x04.txt

Malaysia, K. K.-S. (n.d.). *Kernel Malware:TheAttackfromWithin*. Retrieved from http://www.f-secure.com/weblog/archives/kasslin_AVAR2006_KernelMalware_paper.pdf

mxtone. (n.d.). *Analyzing local privilege escalations in win32k .* Retrieved from http://www.uninformed.org/?v=10&a=2&t=pdf

Rieck, K. (n.d.). *Papillon 0.5.4 – Solaris Security Module Documentation and Manual*. Retrieved from http://www.roqe.org/papillon/papillon.pdf

Shah, A. (n.d.). *Analysis of Rootkits: Attack Approaches and Detection Mechanisms .* Retrieved from http://www.cc.gatech.edu/~salkesh/files/RootkitsReport.pdf

sqrkkyu, t. (n.d.). *Attacking the Core : Kernel Exploiting Notes*. Retrieved from http://www.phrack.com/issues.html?issue=64&id=6

Sun. (n.d.). *Malware Detection at OpenSolaris.org* . Retrieved from http://opensolaris.org/os/project/forensics/Tools/Detection/

Various. (n.d.). Retrieved from DeathRow: http://deathrow.vistech.net/

Various. (n.d.). *OpenVMS driver in C*. Retrieved from http://forums13.itrc.hp.com/service/forums/questionanswer.do?admit=109447627+1226298784441+28353475&threadId=1201842

Various. (n.d.). *OpenVMS Hack Faq*. Retrieved from https://gein.vistech.net/openvms-hack-faq.txt

Various. (n.d.). *The OpenVMS® Operating System*. Retrieved from http://www.pottsoft.com/home/vms/vms.html

Various. (n.d.). *Undocumented OpenVMS features*. Retrieved from http://zinser.no-ip.info/www/eng/vms/qaa/undoc.htmlx

Various. (n.d.). *VMS books*. Retrieved from http://wwwvms.mppmu.mpg.de/vms/ava/vms_book.html