A Brief Survey on Rootkit Techniques in Malicious Codes

Sungkwan Kim, Junyoung Park, Kyungroul Lee
Soonchunhyang University
Shinchang-myun, Asan-si, Republic of Korea
{carpedm, wwkim3, apple}@sch.ac.kr

Ilsun You
Korean Bible University
Seoul, Republic of Korea
isyou@bible.ac.kr

Kangbin Yim*
Soonchunhyang University
Shinchang-myun, Asan-si, Republic of Korea
yim@sch.ac.kr

Abstract

Nowadays, malicious codes are significantly increasing, leading to serious damages to information systems. It is worth to note that these codes generally depend on the rootkit techniques to make it more difficult for themselves to be analyzed and detected. Therefore, it is of paramount importance to research the rootkits to effectively defend against malicious codes. In this paper, we explore and survey the rootkit techniques both in user-level and kernel-level. Several rootkit samples are also utilized for the test and verification purpose.

Keywords: rootkit, malicious codes, keyboard security

1 Introduction

The superlative invention of the Internet in the 20th century has generated innovative developments in the computer industry. While internet speed is increasing, adverse effects are also increasing. A computer virus, which would have taken a long time to propagate in former days, can now spread throughout the world currently in a few seconds or minutes. In the past, the spreading technology was very simple and the path was limited, thus enabling an account technician sufficient time to protect against the virus. However, malicious codes have been intelligently armed with self-deformation and concealment, creating many problems such as Distributed Denial of Services Attacks (DDoS), SPAM, and hijacking of personal information[2]. The number of malicious codes and different types of viruses has continued to increase in the first half of 2011[4]. Security specialists collect, disassemble, and analyze malicious codes to deal with their security threats. Especially, by using reverse engineering, they can trace and analyze each stage of the malicious act so they are able to determine the damage level, spreading speed, eliminating method, and effective countermeasure to prevent malicious codes from spreading. Note that reverse engineering can be used not only by security technicians but also by malicious code developers[5]. These developers make use of reverse engineering to detect the weak points of the malicious codes and make these codes stronger. Furthermore, they create codes that can avoid security specialist's analysis, using anti-debugging and rootkit. Security specialists are therefore researching ways to cope with these evolving malicious codes. However, malicious code generation tools are in circulation through hacking forums and closed news, making malicious codes easier to access, while techniques to evade analysis

*Corresponding Author: LISA Laboratory, Soonchunhyang University, 9418, Engineering Building, 646, Eupnae-ri, Shinchang-myun, Asan-si, Chungcheongnam-do, Republic of Korea, Tel: +82-(0)415301741, Web: http://lisa.sch.ac.kr are progressing. Thus, it is not easy for security specialists to timely and effectively defend against malicious codes. To make matters worse, human resources for analysis of malicious code are deficient and the development speed of analysis technology is not faster than that of malicious codes. Importantly, most malicious codes take full advantage of the rootkit techniques to hide themselves while evading the analysis and detection. Therefore, it is vital for security specialists to study the rootkits in addition to malicious codes. Motivated by this, this paper explores the rootkit techniques used in malicious codes and studies their countermeasures. Additionally, sample rootkits are implemented and executed to analyze the rootkit techniques in more practical and effective.

2 What is the Rootkit?

A rootkit is a software that is developed to enable its associated programs to conceal their presence from analysis and detection while maintaining privileged access to a target system[12]. It is worth to note that contemporary malwares tend to apply the rootkit techniques to install and activate themselves without being detected, thus effectively doing serious damages to victim systems. Once installed, the rootkit can modify the Operating System (OS) or the existing softwares, some of which might otherwise be utilized for malware detection and analysis. Therefore, it is so hard to detect the rootkit and its associated programs because it can subvert security or anti-virus programs whose goal is to find itself. As a result, it becomes an important challenge to discover and handle the rootkit.

Normally, each OS consists of a user-mode, called ‘ring 3’ and a kernel-mode, called ‘ring 0’. In the kernel-mode, actual codes are executed to perform the core tasks given by the OS, whereas in the user-mode those codes are indirectly executed via an Application Programming Interface (API). In other words, in the user-mode, the API is not used to directly process the kernel operations but to assist in accessing the kernel-mode. A rootkit usually conceals itself in the process of accessing the user-mode and kernel-mode. Figure 1 shows a rootkit in detail[3][1].

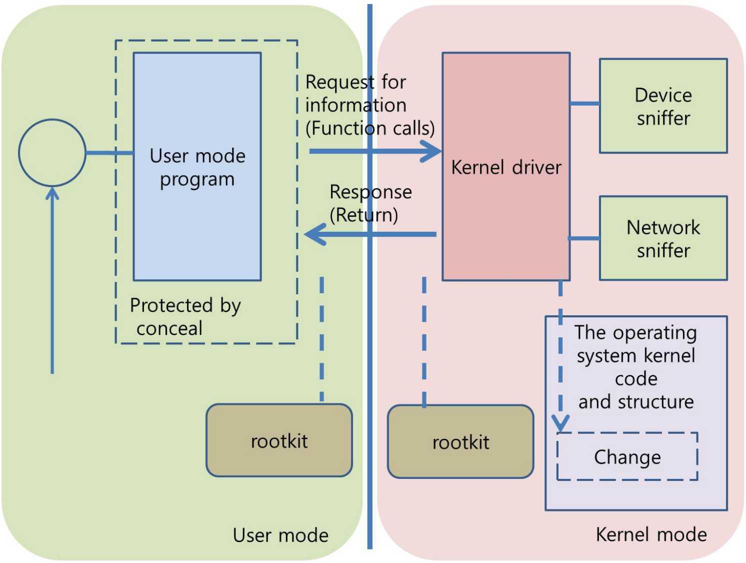


Figure 1: Rootkit’s position and role in a system

As depicted in Figures 2 and 3, rootkits can be divided into three types: the user-level, kernel-level, and other rootkits. In the subsequent sections, we will focus on the user-level and kernel-level rootkits.

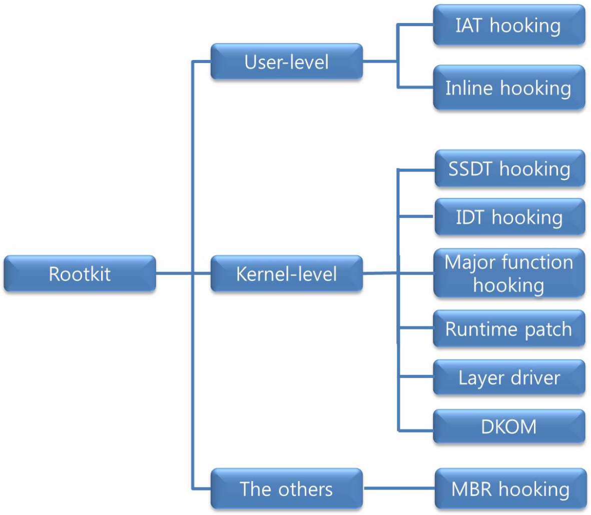
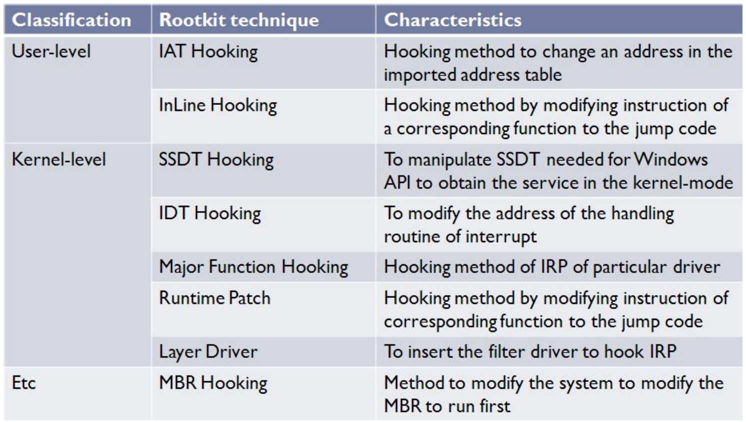


Figure 2: shows this classification



Classification	Rootkit technique	Characteristics
User-level	IAT Hooking	Hooking method to change an address in the imported address table
	InLine Hooking	Hooking method by modifying instruction of a corresponding function to the jump code
Kernel-level	SSDT Hooking	To manipulate SSDT needed for Windows API to obtain the service in the kernel-mode
	IDT Hooking	To modify the address of the handling routine of interrupt
	Major Function Hooking	Hooking method of IRP of particular driver
	Runtime Patch	Hooking method by modifying instruction of corresponding function to the jump code
	Layer Driver	To insert the filter driver to hook IRP
Etc	MBR Hooking	Method to modify the system to modify the MBR to run first

Figure 3: Types of rootkit

3 User-level Rootkit

The user-level rootkits exploit the API hooking to hide themselves since they have no authority to control the kernel structure. In more detail, the API Hooking helps these rootkits to perform their concealment by accessing the program's memory address space and manipulating its information. However, the main drawback of the user-level rootkits is that they are easily detected by the rootkit detector. There are 3 subsystems: Win32, POSIX, OS/2 on Windows. Each sub system is composed of APIs, through which a process asks its OS to execute its required operations. Therefore, a user-level rootkit can manipulate APIs, *i.e.* API hooking, to modify the regular operations of its OS at its will. The API hookings are largely classified into Import Address Table (IAT) hooking and inline hooking. In the IAT hooking, a hacker tries to replaces an origianl function address stored in IAT with the address of the rootkit. In this way, if the original function is called, the rootkit is executed. Unlike the IAT hooking, the inline hooking tries to modify the function code of a victim program at its will, thus being independent on the function address.

In order to hook the API, the rootkit code should be inserted into another process' address space. The representative method for this is the Dynamic-Link Library (DLL) Injection

3.1 DLL Injection

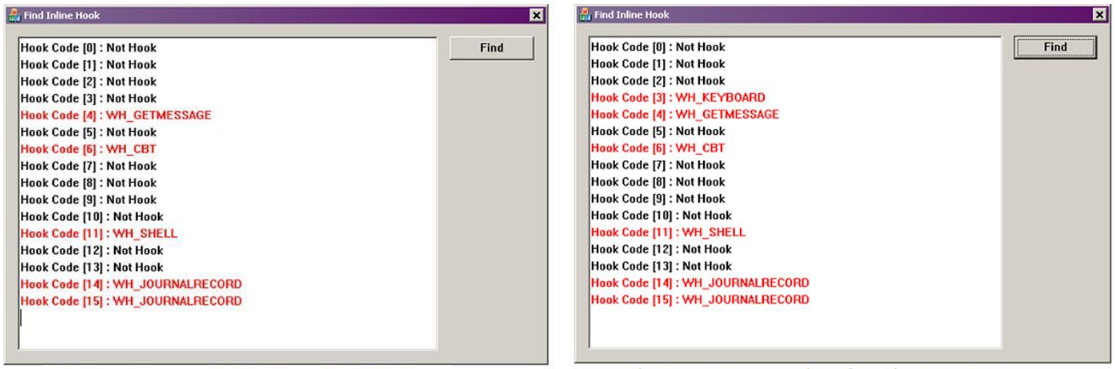
The DLL injection is a method aiming to insert a rootkit code or a malicious code into another process' address space. Once a malicious DLL is injected into a program, a hacker can amend the execution flow of the program's API calls. This method can be categorized into three techniques: using registry, using hooking function, and using remote thread.

- DLL Injection using registry

A hacker can conduct the DLL injection by modifying the registry value of `HKEY_LOCAL_MACHINE\Software\Microsoft\WindowsNt\CurrentVersion\Windows\AppInit_DLLs` and adding a malicious DLL to `windows NT/2000/XP/2003.` In order to use this injection, rebooting is necessary because the changed registry value should be applied in the system. Once the registry change is reflected, the system needs no more booting for the newly started processes. Clearly it is easy to run this method, but at least one reboot is necessary. Thus, the security specialist can just check the registry in every booting to prevent this DLL injection.

- DLL injection using window hooking function

In a windows OS, application programs are executed based on events. The SetWindowsHookEx[9] function is defined to process an event in a special way by allowing an application to install an user-defined hook procedure into a hook chain. As shown in the function prototype, this function is called with an event, which will be hooked, and a hook procedure. Then, the hook procedure is called whenever the event happens once being loaded into the virtual memory address by the function. It is necessary to monitor the DLL injections using windows hooking or remote thread in real-time because they are immediately operated without rebooting. For this goal, in the case of using the window hooking function, we can use the PDESKTOPINFO pDeskInfo structure which is in the THREADINFO structure and has the desktop information. The array of a global hooking list is registered inside the DESKTOPINFO structure PHOOK[7] aphkstart. Since the HOOK structure has information on the hooking event type, simply checking this field can determine whether global hooking is set. There are 15 global hooking types. Results of global hooking before and after the injection are shown in Figure 4.



```
Hook Code [0] : Not Hook
Hook Code [1] : Not Hook
Hook Code [2] : Not Hook
Hook Code [3] : Not Hook
Hook Code [4] : WH_GETMESSAGE
Hook Code [5] : Not Hook
Hook Code [6] : WH_CBT
Hook Code [7] : Not Hook
Hook Code [8] : Not Hook
Hook Code [9] : Not Hook
Hook Code [10] : Not Hook
Hook Code [11] : WH_SHELL
Hook Code [12] : Not Hook
Hook Code [13] : Not Hook
Hook Code [14] : WH_JOURNALRECORD
Hook Code [15] : WH_JOURNALRECORD
```

a) Detection results before the injection

```
Hook Code [0] : Not Hook
Hook Code [1] : Not Hook
Hook Code [2] : Not Hook
Hook Code [3] : WH_KEYBOARD
Hook Code [4] : WH_GETMESSAGE
Hook Code [5] : Not Hook
Hook Code [6] : WH_CBT
Hook Code [7] : Not Hook
Hook Code [8] : Not Hook
Hook Code [9] : Not Hook
Hook Code [10] : Not Hook
Hook Code [11] : WH_SHELL
Hook Code [12] : Not Hook
Hook Code [13] : Not Hook
Hook Code [14] : WH_JOURNALRECORD
Hook Code [15] : WH_JOURNALRECORD
```

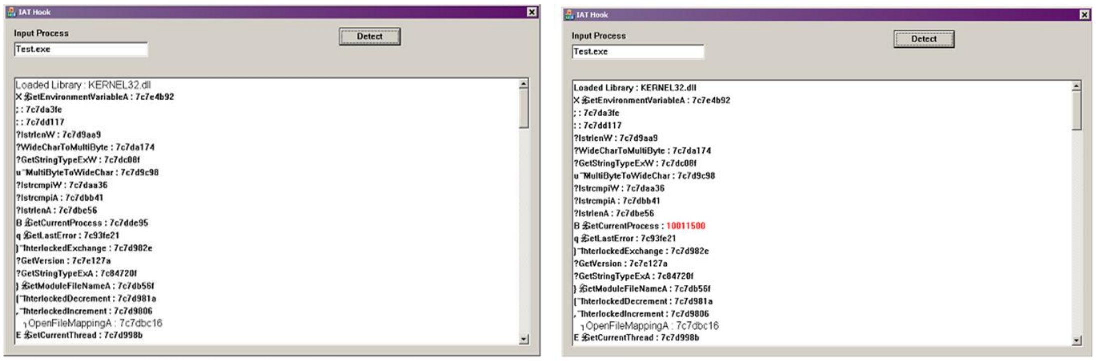
b) Detection results after the injection

Figure 4: Method for DLL injection using window hooking function

An embodied example attack code proceeds to global hooking at the keyboard entry. We can confirm that hooking is not established at the 4th distribution before performing hooking. After hooking, global hooking is established on the keyboard event at the 4th distribution. Therefore, a bulwark can detect global hooking by periodically chasing an established global hooking event.

- DLL Injection using remote thread

This method creates a remote thread for a victim process to inject the rootkit DLL in the address area of that process. For this purpose, the CreateRemoteThread function[6] is used. In order to start this injection, we first use the VirtualAllocEx function[10] to assign a memory to the address space of a victim process, and use the WriteProcessMemory function[11] to put the DLL name. Then, the LoadLibrary function is called for the DLL load, and the CreateRomoteThread function is executed to create the thread in the destination process. This DLL injection is a more intellectual attack because it can attack not only the global process, but every process. Thus, countering this method is very time consuming and causes an overload of the system because all processes should be examined. This attack loads the DLL in which the malicious code is inserted, creates the thread and executes that code, so it is detectable by checking the numbers of the generated threads. However, in the case of the network server program, the above method is not effective because the number of threads changes dynamically according to the client's request. This problem is solved by testing if the DLL loaded in the process is a certified one. Because the malicious DLL should be loaded for attack, it can be more efficient to detect a DLL without a permit or a DLL that is not included in the original file. This is shown in Figure 5.



Test.exe

```
Loaded Library : KERNEL32.dll
X GetEnvironmentVariableA : 7c7e4b92
;: 7c7da3fe
:: 7c7dd117
?lstrlenW : 7c7d9aa9
?WideCharToMultiByte : 7c7da174
?GetStringTypeExW : 7c7dc08f
u¯MultiByteToWideChar : 7c7d9c98
?lstrcmpiW : 7c7daa36
?lstrcmpiA : 7c7dbb41
?lstrlenA : 7c7dbe56
B GetCurrentProcess : 7c7dde95
q GetLastError : 7c93fe21
]¯InterlockedExchange : 7c7d982e
?GetVersion : 7c7e127a
?GetStringTypeExA : 7c84720f
} GetModuleFileNameA : 7c7db56f
[¯InterlockedDecrement : 7c7d981a
,¯InterlockedIncrement : 7c7d9806
 ┐OpenFileMappingA : 7c7dbc16
E GetCurrentThread : 7c7d998b
```

a) Detection results before the injection

Test.exe

```
Loaded Library : KERNEL32.dll
X GetEnvironmentVariableA : 7c7e4b92
;: 7c7da3fe
:: 7c7dd117
?lstrlenW : 7c7d9aa9
?WideCharToMultiByte : 7c7da174
?GetStringTypeExW : 7c7dc08f
u¯MultiByteToWideChar : 7c7d9c98
?lstrcmpiW : 7c7daa36
?lstrcmpiA : 7c7dbb41
?lstrlenA : 7c7dbe56
B GetCurrentProcess : 10011500
q GetLastError : 7c93fe21
]¯InterlockedExchange : 7c7d982e
?GetVersion : 7c7e127a
?GetStringTypeExA : 7c84720f
} GetModuleFileNameA : 7c7db56f
[¯InterlockedDecrement : 7c7d981a
,¯InterlockedIncrement : 7c7d9806
 ┐OpenFileMappingA : 7c7dbc16
E GetCurrentThread : 7c7d998b
```

b) Detection results after the injection

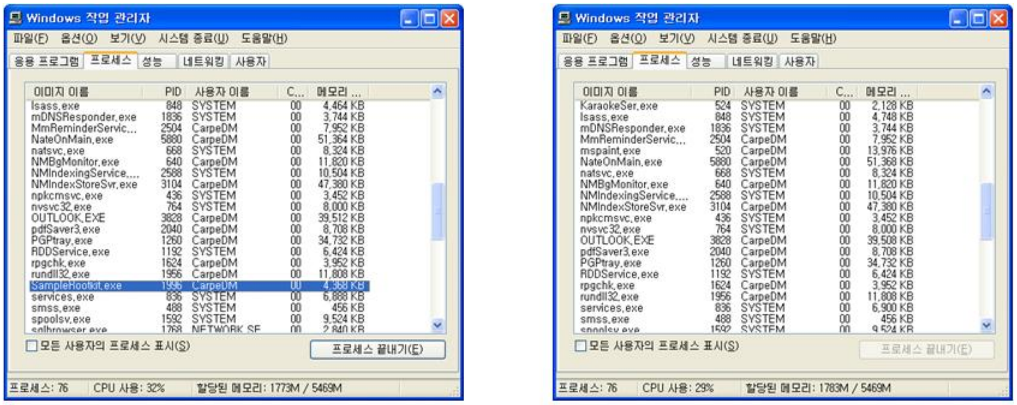
Figure 5: Method for DLL injection using remote thread

4 Kernel-level Rootkit

It is difficult to detect the kernel-level rootkit because it performs hidden behaviors by hooking the native API, and executes in the same level as anti-rootkit softwares. The most commonly used hooking methods in the kernel mode include System Service Descriptor Table (SSDT), Interrupt Descriptor Table (IDT), and a major function of device driver object, and so on, most of which are based on execution codes. Because modified execution codes can be detected by the anti-rootkit software, some attackers adopt the Direct Kernel Object Manipulation (DKOM) technique, which modifies kernel-level data .

4.1 System Service Dispatch Table (SSDT) Hooking

A Microsoft windows supports sub-systems for the Win32, POSIX, and OS/2. Hence, for these subsystems, a specific kernel structure is prepared called the System Service Dispatch Table (SSDT). This table maintains the memory addresses of the related fuctions in the order of the system calls. In addition, the System Parameter Table (SSPT) is used to manage the size of parameters used in each system service factions, and in a Microsoft windows, it is possible to obtain various types of data by the ZwQuerySystemInformation function. This function provides a list of running processes; thus, a rootkit can hide one of them by changing the address of the NwQuerySystemInformation function. Figure 6 shows an example of a hidden process using this technique.


Windows 작업 관리자
파일(F) 옵션(O) 보기(V) 시스템 종료(U) 도움말(H)
응용 프로그램 프로세스 성능 네트워킹 사용자
□ 모든 사용자의 프로세스 표시(S)
프로세스 끝내기(E)

이미지 이름	PID	사용자 이름	C...	메모리 ...
lsass.exe	848	SYSTEM	00	4,464 KB
mDNSResponder.exe	1836	SYSTEM	00	3,744 KB
MmReminderServic...	2504	CarpeDM	00	7,952 KB
NateOnMain.exe	5880	CarpeDM	00	51,364 KB
natsvc.exe	668	SYSTEM	00	8,324 KB
NMBgMonitor.exe	640	CarpeDM	00	11,820 KB
NMIndexingService....	2588	SYSTEM	00	10,504 KB
NMIndexStoreSvr.exe	3104	CarpeDM	00	47,380 KB
npkcmsvc.exe	436	SYSTEM	00	3,452 KB
nvsvc32.exe	764	SYSTEM	00	8,000 KB
OUTLOOK.EXE	3828	CarpeDM	00	39,512 KB
pdfSaver3.exe	2040	CarpeDM	00	8,708 KB
PGPtray.exe	1260	CarpeDM	00	34,732 KB
RDDService.exe	1192	SYSTEM	00	6,424 KB
rpgchk.exe	1624	CarpeDM	00	3,952 KB
rundll32.exe	1956	CarpeDM	00	11,808 KB
SampleRootkit.exe	1996	CarpeDM	00	4,368 KB
services.exe	836	SYSTEM	00	6,888 KB
smss.exe	488	SYSTEM	00	456 KB
spoolsv.exe	1592	SYSTEM	00	9,524 KB
sqlbrowser.exe	1768	NETWORK SE	00	2,840 KB

프로세스: 76 CPU 사용: 32% 할당된 메모리: 1773M / 5469M

a) process list before hooking

이미지 이름	PID	사용자 이름	C...	메모리 ...
KaraokeSer.exe	524	SYSTEM	00	2,128 KB
lsass.exe	848	SYSTEM	00	4,748 KB
mDNSResponder.exe	1836	SYSTEM	00	3,744 KB
MmReminderServic...	2504	CarpeDM	00	7,952 KB
mspaint.exe	520	CarpeDM	00	13,976 KB
NateOnMain.exe	5880	CarpeDM	00	51,368 KB
natsvc.exe	668	SYSTEM	00	8,324 KB
NMBgMonitor.exe	640	CarpeDM	00	11,820 KB
NMIndexingService....	2588	SYSTEM	00	10,504 KB
NMIndexStoreSvr.exe	3104	CarpeDM	00	47,380 KB
npkcmsvc.exe	436	SYSTEM	00	3,452 KB
nvsvc32.exe	764	SYSTEM	00	8,000 KB
OUTLOOK.EXE	3828	CarpeDM	00	39,508 KB
pdfSaver3.exe	2040	CarpeDM	00	8,708 KB
PGPtray.exe	1260	CarpeDM	00	34,732 KB
RDDService.exe	1192	SYSTEM	00	6,424 KB
rpgchk.exe	1624	CarpeDM	00	3,952 KB
rundll32.exe	1956	CarpeDM	00	11,808 KB
services.exe	836	SYSTEM	00	6,900 KB
smss.exe	488	SYSTEM	00	456 KB
spoolsv.exe	1592	SYSTEM	00	9,524 KB

프로세스: 76 CPU 사용: 29% 할당된 메모리: 1783M / 5469M

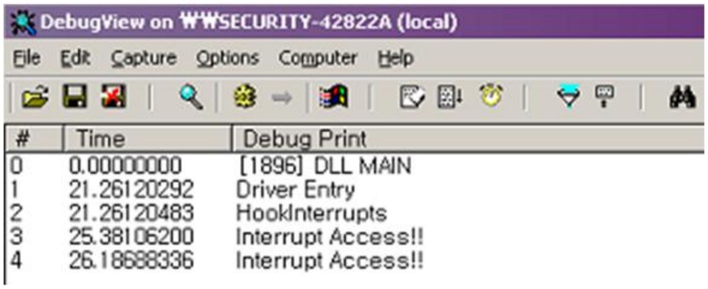
b) process list after hooking

Figure 6: SSDT hooking result

Note that the above example is the rootkit program that deodorizes keyboard data. With the help of the SSDT hooking, the rootkit hides itself to evade the detection. To address this attack, we can maintain the white list including all SSDT tables to detect forge/modulation.

4.2 Interrupt Descriptor Table (IDT) Hooking

The Interrupt Descriptor Table (IDT) is a system table used to handle the interrupt. The interrupt is caused by software or hardware, and a keyboard interrupt is one of the typical interrupts. An OS executes the keyboard interrupt by calling the keyboard interrupt handler when a user types a key. The sidt instruction can be used to get the information and location of an IDT. Therefore, an attacker is able to hook by collecting the IDT information using this instruction, and Figure 7 depicts an example.


DebugView on ₩₩SECURITY-42822A (local)
File Edit Capture Options Computer Help

```
#  Time          Debug Print
0  0.00000000    [1896] DLL MAIN
1  21.26120292   Driver Entry
2  21.26120483   HookInterrupts
3  25.38106200   Interrupt Access!!
4  26.18688336   Interrupt Access!!
```

Figure 7: IDT hooking result

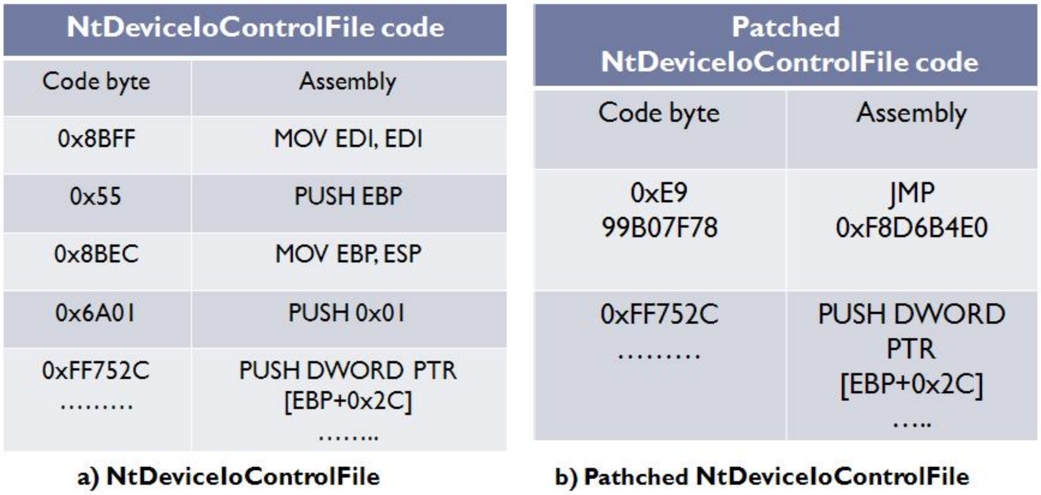
When the keyboard interrupt happens, the above example prints a particular message by the hooked interrupt. Note that the actual keyboard data can be leaked by this technique. We are able to detect forge/modulation by constructing the white list including all IDT tables.

4.3 Hooking to Inline Function

In the inline function hooking, the rootkit inserts an unconditional jump within the body of a victim function to an area of the memory which it can control with some malicious codes. For example, the

NtDeviceIoControlFile function code[2] can be patched into a new one [4] so that it can jump to the rootkit code. For the actual test, we use the NtDeviceIoControlFile, which sends the device control code and related information to the device driver. Using this, we can hide or change the information.

Figure 8 shows the NoDeviceIoControlFile and patched NtDeviceIoControlFile codes while Figures 9 and 10 show the execution result of the sample rootkit.



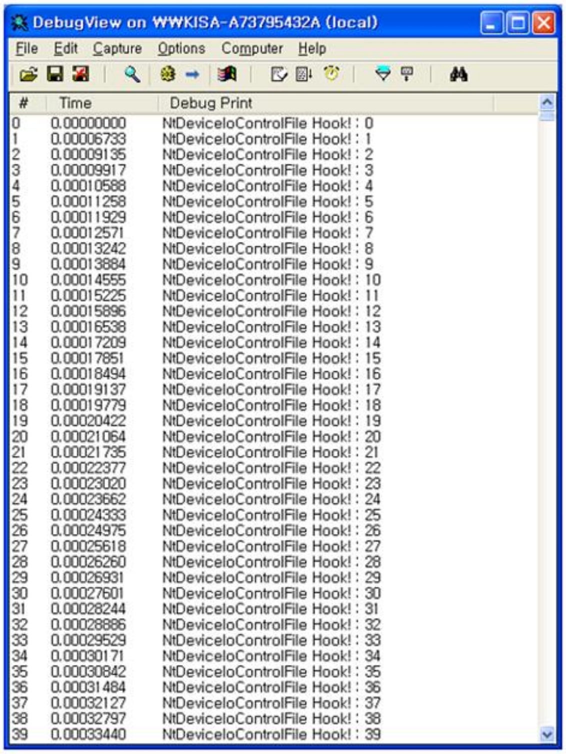
NtDeviceIoControlFile code	
Code byte	Assembly
0x8BFF	MOV EDI, EDI
0x55	PUSH EBP
0x8BEC	MOV EBP, ESP
0x6A01	PUSH 0x01
0xFF752C ………	PUSH DWORD PTR [EBP+0x2C] ……..

a) NtDeviceIoControlFile

Patched NtDeviceIoControlFile code	
Code byte	Assembly
0xE9 99B07F78	JMP 0xF8D6B4E0
0xFF752C ………	PUSH DWORD PTR [EBP+0x2C] …..

b) Pathched NtDeviceIoControlFile

Figure 8: The result of NoDeviceIoControlFile code patch


DebugView on WWKISA-A73795432A (local)
File Edit Capture Options Computer Help

```
#   Time        Debug Print
0   0.00000000  NtDeviceIoControlFile Hook! : 0
1   0.00006733  NtDeviceIoControlFile Hook! : 1
2   0.00009135  NtDeviceIoControlFile Hook! : 2
3   0.00009917  NtDeviceIoControlFile Hook! : 3
4   0.00010588  NtDeviceIoControlFile Hook! : 4
5   0.00011258  NtDeviceIoControlFile Hook! : 5
6   0.00011929  NtDeviceIoControlFile Hook! : 6
7   0.00012571  NtDeviceIoControlFile Hook! : 7
8   0.00013242  NtDeviceIoControlFile Hook! : 8
9   0.00013884  NtDeviceIoControlFile Hook! : 9
10  0.00014555  NtDeviceIoControlFile Hook! : 10
11  0.00015225  NtDeviceIoControlFile Hook! : 11
12  0.00015896  NtDeviceIoControlFile Hook! : 12
13  0.00016538  NtDeviceIoControlFile Hook! : 13
14  0.00017209  NtDeviceIoControlFile Hook! : 14
15  0.00017851  NtDeviceIoControlFile Hook! : 15
16  0.00018494  NtDeviceIoControlFile Hook! : 16
17  0.00019137  NtDeviceIoControlFile Hook! : 17
18  0.00019779  NtDeviceIoControlFile Hook! : 18
19  0.00020422  NtDeviceIoControlFile Hook! : 19
20  0.00021064  NtDeviceIoControlFile Hook! : 20
21  0.00021735  NtDeviceIoControlFile Hook! : 21
22  0.00022377  NtDeviceIoControlFile Hook! : 22
23  0.00023020  NtDeviceIoControlFile Hook! : 23
24  0.00023662  NtDeviceIoControlFile Hook! : 24
25  0.00024333  NtDeviceIoControlFile Hook! : 25
26  0.00024975  NtDeviceIoControlFile Hook! : 26
27  0.00025618  NtDeviceIoControlFile Hook! : 27
28  0.00026260  NtDeviceIoControlFile Hook! : 28
29  0.00026931  NtDeviceIoControlFile Hook! : 29
30  0.00027601  NtDeviceIoControlFile Hook! : 30
31  0.00028244  NtDeviceIoControlFile Hook! : 31
32  0.00028886  NtDeviceIoControlFile Hook! : 32
33  0.00029529  NtDeviceIoControlFile Hook! : 33
34  0.00030171  NtDeviceIoControlFile Hook! : 34
35  0.00030842  NtDeviceIoControlFile Hook! : 35
36  0.00031484  NtDeviceIoControlFile Hook! : 36
37  0.00032127  NtDeviceIoControlFile Hook! : 37
38  0.00032797  NtDeviceIoControlFile Hook! : 38
39  0.00033440  NtDeviceIoControlFile Hook! : 39
```

Figure 9: the execution result of the sample rootkit using inline function hooking

This method alters the flow of the internal practice function without changing the contents of the table that has the address of the relevant function because the address of the function itself is not changed. Similar to other hooking, the factor value which is delivered when the system call or function is called can be altered by the inserting rootkit code.

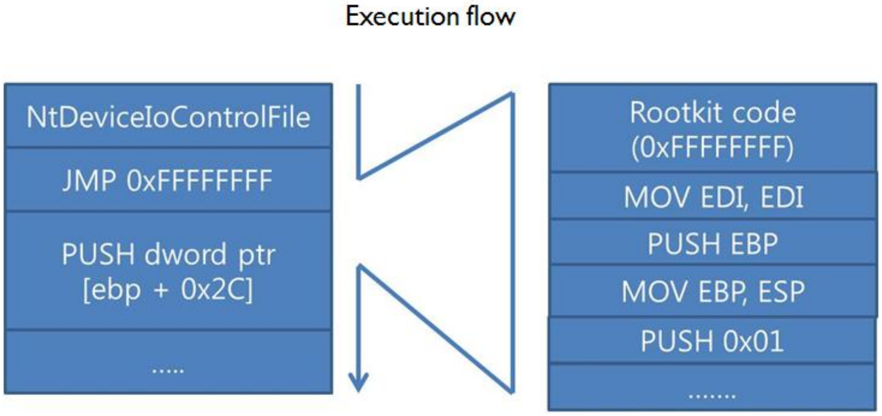
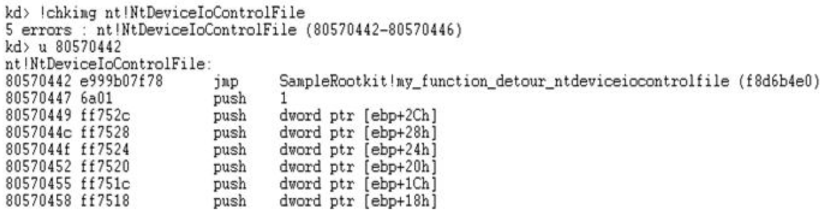


Figure 10: Alteration of operation flow by detour patch

This technique can be detected by checking the beginning part of the original function because its basic action changes the code of the destination function for hooking. To determine if it is hooked, we can also calculate the CRC of the code. Figure 11 shows the result that determines the image coincidence using WinDbug.



```
kd> !chkimg nt!NtDeviceIoControlFile
5 errors : nt!NtDeviceIoControlFile (80570442-80570446)
kd> u 80570442
nt!NtDeviceIoControlFile:
80570442 e999b07f78      jmp     SampleRootkit!my_function_detour_ntdeviceiocontrolfile (f8d6b4e0)
80570447 6a01            push    1
80570449 ff752c          push    dword ptr [ebp+2Ch]
8057044c ff7528          push    dword ptr [ebp+28h]
8057044f ff7524          push    dword ptr [ebp+24h]
80570452 ff7520          push    dword ptr [ebp+20h]
80570455 ff751c          push    dword ptr [ebp+1Ch]
80570458 ff7518          push    dword ptr [ebp+18h]
```

Figure 11: Alteration of operation flow by detour patch

Five errors appear and the memory area with the error is 0x8057442-0x80570446. As a result of disassembling the memory 0x8057442, we can confirm that it diverges to `my_function_detour_ntdeviceiocontrolfile`, which is the Hooking function of the sample rootkit. To retreat the detected function back to the original function, the driver file or non-changed function code can be made.

4.4 Jump Template Technique

In the case of the above hooking, when an attacker wants to hook more functions, they have to patch many functions because a rootkit function should return an original function after it is executed. This procedure is not easy and therefore a new rootkit technique called the jump template technique has been developed. The technique first executes the rootkit function, and then calls the original function while returning to the normal service. Because an attacker needs to return a result of the called original function for a stealth attack, the implementation overhead is reduced due to the automatic call of the original function without the need to patch each function. Figure 12 shows the flow chart of jump template technique and Figure 13 illustrates the implement result of sample rootkit. This technique should let each interrupt routine indicate its corresponding rootkit code address.

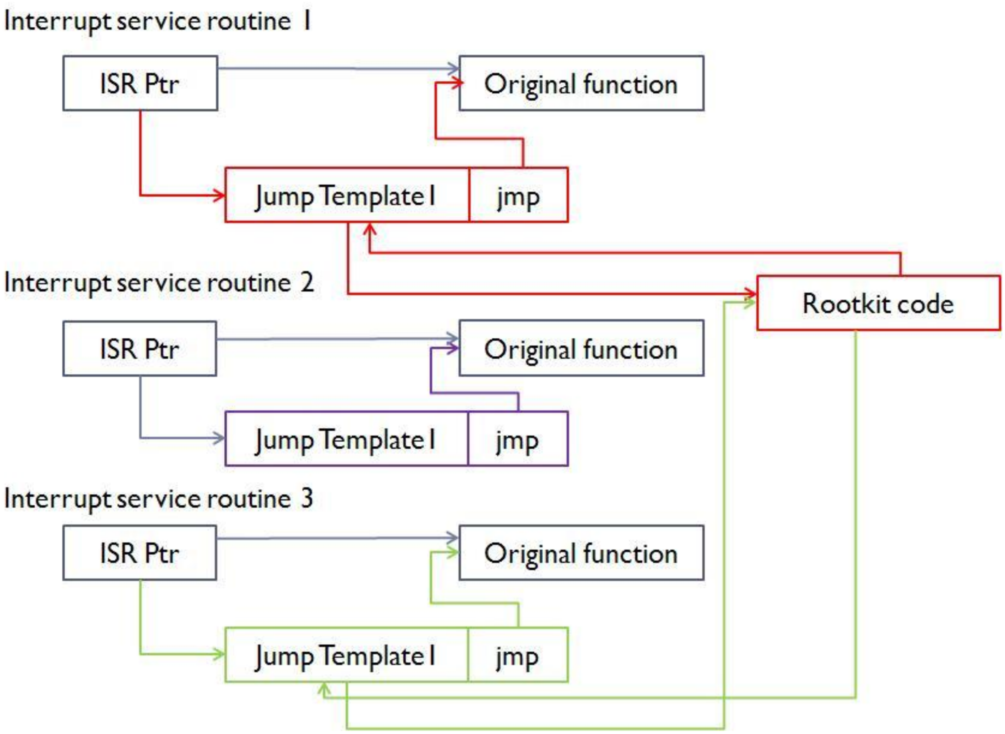
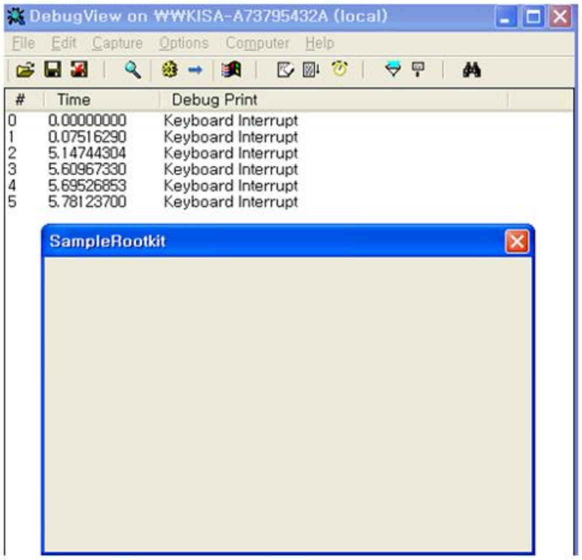


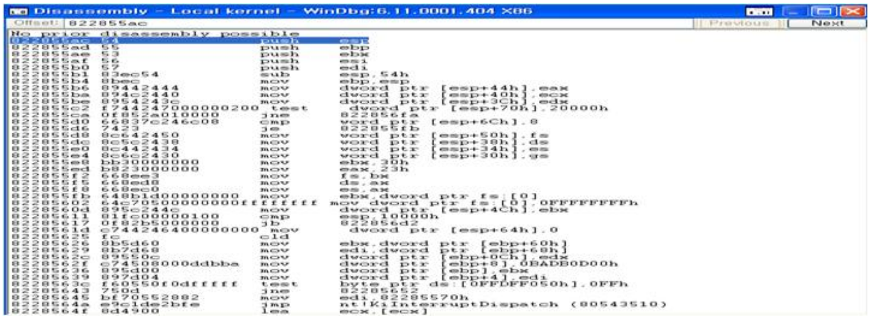
Figure 12: Flow chart of the jump template technique


DebugView on ₩₩KISA-A73795432A (local)
File Edit Capture Options Computer Help
SampleRootkit

```
#  Time        Debug Print
0  0.00000000  Keyboard Interrupt
1  0.07516290  Keyboard Interrupt
2  5.14744304  Keyboard Interrupt
3  5.60967330  Keyboard Interrupt
4  5.69526853  Keyboard Interrupt
5  5.78123700  Keyboard Interrupt
```

Figure 13: Execution result of the sample rookit using jump template

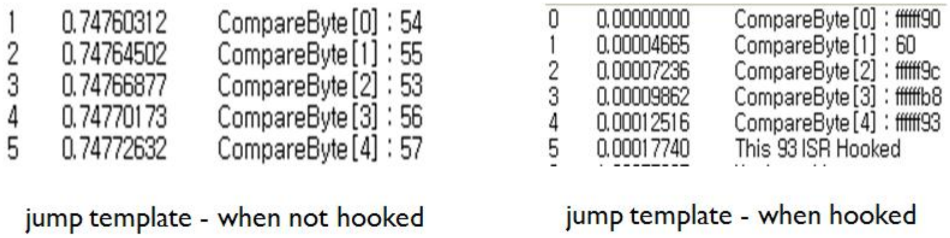
The basic concept of the jump template is that an original code changes from an interrupt service routine to the jump template code. Therefore, the detecting method is limited when verifying an address of the interrupt service routine because a defender does not decide whether or not the interrupt service routine address is correct. Hence, the defender can detect the jump template technique by checking the first five bytes. This means that the first five bytes of the original code are not equal to the jump template code. To prove this, we show disassemble codes of the original interrupt service routine in Figure 14.



```
Disassembly - Local kernel - WinDbg:6.11.0001.404 X86
Offset: 822855ac
No prior disassembly possible
822855ac 54               push    esp
822855ad 55               push    ebp
822855ae 53               push    ebx
822855af 56               push    esi
822855b0 57               push    edi
822855b1 83ec54           sub     esp,54h
822855b4 8bec             mov     ebp,esp
822855b6 89442444         mov     dword ptr [esp+44h],eax
822855ba 894c2440         mov     dword ptr [esp+40h],ecx
822855be 8954243c         mov     dword ptr [esp+3Ch],edx
822855c2 f744247000000200 test     dword ptr [esp+70h],20000h
822855ca 0f852a010000     jne     822856fa
822855d0 66837c246c08     cmp     word ptr [esp+6Ch],8
822855d6 7423             je      822855fb
822855d8 8c642450         mov     word ptr [esp+50h],fs
822855dc 8c5c2438         mov     word ptr [esp+38h],ds
822855e0 8c442434         mov     word ptr [esp+34h],es
822855e4 8c6c2430         mov     word ptr [esp+30h],gs
822855e8 bb30000000       mov     ebx,30h
822855ed b823000000       mov     eax,23h
822855f2 668ee3           mov     fs,bx
822855f5 668ed8           mov     ds,ax
822855f8 668ec0           mov     es,ax
822855fb 648b1d00000000   mov     ebx,dword ptr fs:[0]
82285602 64c70500000000ffffffff mov dword ptr fs:[0],0FFFFFFFFh
8228560d 895c244c         mov     dword ptr [esp+4Ch],ebx
82285611 81fc00000100     cmp     esp,10000h
82285617 0f82b5000000     jb      822856d2
8228561d c744246400000000 mov      dword ptr [esp+64h],0
82285625 fc               cld
82285626 8b5d60           mov     ebx,dword ptr [ebp+60h]
82285629 8b7d68           mov     edi,dword ptr [ebp+68h]
8228562c 89550c           mov     dword ptr [ebp+0Ch],edx
8228562f c74508000ddbba   mov     dword ptr [ebp+8],0BADB0D00h
82285636 895d00           mov     dword ptr [ebp],ebx
82285639 897d04           mov     dword ptr [ebp+4],edi
8228563c f60550f0dfffff   test    byte ptr ds:[0FFDFF050h],0FFh
82285643 750d             jne     82285652
82285645 bf70552882       mov     edi,82285570h
8228564a e9c1de2bfe       jmp     nt!KiInterruptDispatch (80543510)
8228564f 8d4900           lea     ecx,[ecx]
```

Figure 14: Result of the original keyboard interrupt ISR disassemble

In Figure 14, the first five bytes of an original keyboard interrupt service routine consist of PUSH ESP and PUSH EBP, while those of the jump template code different from these instructions. Hence, we can judge that the interrupt service routine is not hooked when the first five bytes are 54, 55, 53, 56, and 57, otherwise it is hooked. Figure 14 shows an implement result of detecting the jump template technique.



```
1   0.74760312   CompareByte[0] : 54
2   0.74764502   CompareByte[1] : 55
3   0.74766877   CompareByte[2] : 53
4   0.74770173   CompareByte[3] : 56
5   0.74772632   CompareByte[4] : 57
```

jump template - when not hooked

```
0   0.00000000   CompareByte[0] : ffffff90
1   0.00004665   CompareByte[1] : 60
2   0.00007236   CompareByte[2] : ffffff9c
3   0.00009862   CompareByte[3] : ffffffb8
4   0.00012516   CompareByte[4] : ffffff93
5   0.00017740   This 93 ISR Hooked
```

jump template - when hooked

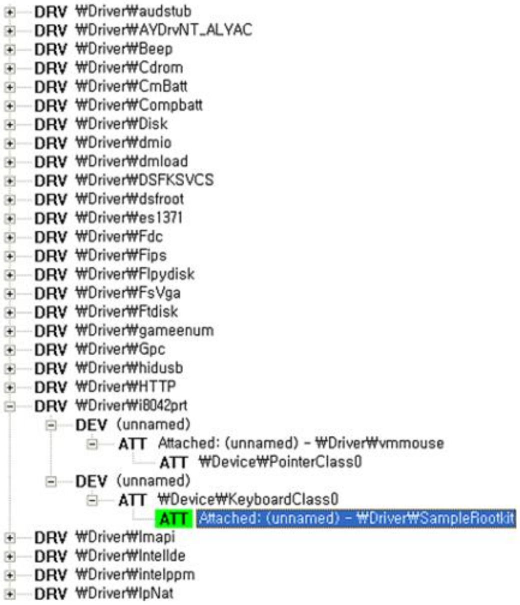
Figure 15: Jump template when hooked

In Figure 15, we can see that 0x93 (keyboard) interrupt is hooked. To treat a hooked function, a defender generates an original function code and then modifies the hooked interrupt service routine to the generated original function code.

4.5 Layered drivers

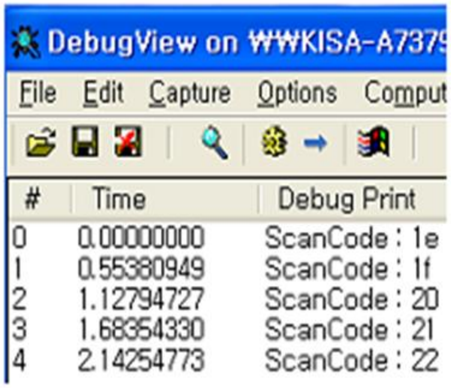
The layered drivers are formed into a hierarchical chain or stack to structuralize the hardware-related drivers physically and functionally. This structure has an advantage for implementation because a developer only implements the part of an additional function when a driver is developed. This means that the developer needs not to implement the entire driver codes and thus the cost for implementation is reduced. Almost all the current hardware devices have driver chains to support layered drivers. In detail, the lowest driver deals with hardware-related functions while the highest driver handles error codes and queries, and structuralizes data to deliver a lower layer driver. Hence, an attacker can steal or hide data which is transferred to the lower or higher layer driver and it calls the rootkit by the layered drivers. The layered drivers use the I/O Request Packet (IRP) for communicating to each other. If a rootkit driver is successfully inserted between the driver chains, it is possible to steal the IRP, which has a lot of important information such as USB data, keyboard data and so on. In the case of a keyboard driver, a request to read the inputted keyboard scan code is generated when a user inputs the key while the IRP is generated.

The generated IRP is transferred to the i8042prt driver, which is the lowest driver in the keyboard driver stacks. In this process, if an attacker inserts the rootkit driver at the higher driver of the i8042prt driver, he or she can modify or steal the IRP. This is because the I8042prt driver reads the keyboard data from a keyboard output buffer and the driver then transfers the IRP to the higher driver after storing the keyboard scan code in the IRP. It means that the IRP transferred to the rootkit driver is stored in the keyboard scan code and the attacker is thus able to steal or modify the keyboard scan code. We build a sample rootkit that deodorizes keyboard data through this technique. Figure 16 is shown sample rootkit stack inserted in PS/2 keyboard device stack. Figure 16 is shown the result of program execution.



```
DRV ₩Driver₩audstub
DRV ₩Driver₩AYDrvNT_ALYAC
DRV ₩Driver₩Beep
DRV ₩Driver₩Cdrom
DRV ₩Driver₩CmBatt
DRV ₩Driver₩Compbatt
DRV ₩Driver₩Disk
DRV ₩Driver₩dmio
DRV ₩Driver₩dmload
DRV ₩Driver₩DSFKSVCS
DRV ₩Driver₩dsfroot
DRV ₩Driver₩es1371
DRV ₩Driver₩Fdc
DRV ₩Driver₩Fips
DRV ₩Driver₩Flpydisk
DRV ₩Driver₩FsVga
DRV ₩Driver₩Ftdisk
DRV ₩Driver₩gameenum
DRV ₩Driver₩Gpc
DRV ₩Driver₩hidusb
DRV ₩Driver₩HTTP
DRV ₩Driver₩i8042prt
    DEV (unnamed)
        ATT Attached: (unnamed) - ₩Driver₩vmmouse
            ATT ₩Device₩PointerClass0
    DEV (unnamed)
        ATT ₩Device₩KeyboardClass0
            ATT Attached: (unnamed) - ₩Driver₩SampleRootkit
DRV ₩Driver₩Imapi
DRV ₩Driver₩Intellde
DRV ₩Driver₩intelppm
DRV ₩Driver₩IpNat
```

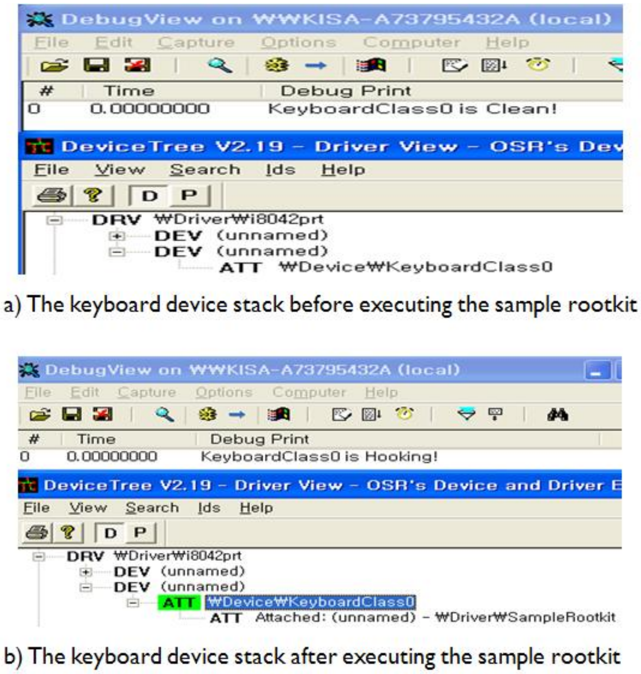
Figure 16: sample rootkit driver inserted in ps/2 keyboard device stack



```
#   Time          Debug Print
0   0.00000000    ScanCode : 1e
1   0.55380949    ScanCode : 1f
2   1.12794727    ScanCode : 20
3   1.68354330    ScanCode : 21
4   2.14254773    ScanCode : 22
```

Figure 17: sample rootkit results using layer driver

In the case of the approach of the layered drivers, the rootkit driver needs to link the device chains. By using this, we can detect this rootkit. That is, if an unknown driver connects to the device chains, the defender can decide that the driver is the rootkit driver. For this, the defender must know all the device chains of all the drivers. In order to prove this detection method, we implement a sample rootkit driver that is attached to the i8042prt driver. Figure 19 shows the detection result of the rootkit by the layered drivers.


DebugView on ₩₩KISA-A73795432A (local)
File Edit Capture Options Computer Help
DeviceTree V2.19 - Driver View - OSR's Dev
File View Search Ids Help
D P

```
#   Time          Debug Print
0   0.00000000    KeyboardClass0 is Clean!
```

```
DRV ₩Driver₩i8042prt
    DEV (unnamed)
    DEV (unnamed)
        ATT ₩Device₩KeyboardClass0
```

a) The keyboard device stack before executing the sample rootkit

DebugView on ₩₩KISA-A73795432A (local)
File Edit Capture Options Computer Help
DeviceTree V2.19 - Driver View - OSR's Device and Driver E
File View Search Ids Help
D P

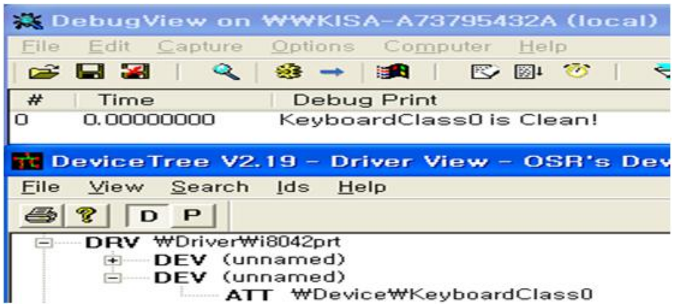
```
#   Time          Debug Print
0   0.00000000    KeyboardClass0 is Hooking!
```

```
DRV ₩Driver₩i8042prt
    DEV (unnamed)
    DEV (unnamed)
        ATT ₩Device₩KeyboardClass0
            ATT Attached: (unnamed) - ₩Driver₩SampleRootkit
```

b) The keyboard device stack after executing the sample rootkit

Figure 18: The detection result of the keyboard rootkit driver by layered drivers

In the case of an original device stack, there is no driver higher than the \Device\Keyboardclass0, although when inserting the rootkit driver, we know that the \Device\Samplerootkit driver exists as shown in Figure 19. Therefore, we can decide that the \Device\Samplerootkit driver is the rootkit driver. In order to treat the rootkit driver, a defender inserts an anti-rootkit driver at the lower stack of the inserted rootkit driver. Using this countermeasure, if the anti-rootkit driver can erase the keyboard scan code in the IRP, the rootkit driver cannot steal and modify the keyboard scan code. Nevertheless, there is a competition problem with this countermeasure between the attacker and the defender because the rootkit driver is able to insert at the lower stack of the anti-rootkit driver. Hence, this countermeasure can treat the rootkit driver in a short period of time. The other countermeasure involves detaching the detected driver from the device stacks by the IoDetachDevice function [8]. Figure 19 shows the treatment result of the sample anti-rootkit driver. Figure 19, a keyboard scan code is exposed before loading the


DebugView on ₩₩KISA-A73795432A (local)
File Edit Capture Options Computer Help
DeviceTree V2.19 - Driver View - OSR's Dev
File View Search Ids Help
D P

```
#   Time          Debug Print
0   0.00000000    KeyboardClass0 is Clean!
```

```
DRV ₩Driver₩i8042prt
    DEV (unnamed)
    DEV (unnamed)
        ATT ₩Device₩KeyboardClass0
```

Figure 19: The treatment result of the sample anti-rookit driver

anti-rootkit driver. Otherwise, after loading the anti-rootkit driver, the keyboard scan code is not exposed to anyone.

5 Conclusion

This paper has briefly surveyed the rootkit techniques while discussing the countermeasures. Such a survey is powered by the sample rootkits, which are specially implemented to show how they operate. As a future work, it is necessary to study the latest rookit technologies in newly emerging computing platforms such as smart phones, cloud computing, and so forth.

References

[1] H. G. and B. J. *Rootkits: Subverting the Windows Kernel*. Pearson Education, Inc., 2005.

[2] J. Kim, J. Lee, E. Park, E. Jang, and H. Kim. Research on Modeling and Simulation Technique for Cost Analysis on DDoS Attack Damage Scale and Dealing Method. In *Proc. of the Korea Simulation Association (KSS'09), Korea*, pages 39–47, December 2009.

[3] Y. Kwon. *Rootkit, Reveal Your Identity: The Story of spears and shields*. Microsoftware, 2008.

[4] A. Lab). *Monthly Security Report*. Ahn lab, September 2011.

[5] D. Lee, S. Lee, and G. Lee. Research on early detection technique on cyber threats based on honey net. In *Proc. of the Korea Information Assurance Society (KIAS'05), Korea*, pages 67–72, December 2005.

[6] msdn. CreateRemoteThread function. MSDN(Microsoft), September 2012. `http://msdn.microsoft.com/en-us/library/ms682437(VS.85).aspx`.

[7] msdn. Hooks Overview. MSDN(Microsoft), September 2012. `http://msdn.microsoft.com/en-us/library/ms644959(VS.85).aspx`.

[8] msdn. IoDetachDevice routine. MSDN(Microsoft), September 2012. `http://msdn.microsoft.com/en-us/library/ff549087(VS.85).aspx`.

[9] msdn. SetWindowsHookEx function. MSDN(Microsoft), September 2012. `http://msdn.microsoft.com/en-us/library/ms644990(VS.85).aspx`.

[10] msdn. VirtualAllocEx function. MSDN(Microsoft), September 2012. `http://msdn.microsoft.com/en-us/site/aa366890`.

[11] msdn. WriteProcessMemory function. MSDN(Microsoft), September 2012. `http://msdn.microsoft.com/en-us/library/ms681674(VS.85).aspx`.

[12] Wikipedia. Rootkit. `http://en.wikipedia.org/wiki/Rootkit`.

Sungkwan Kim is currently working towards his B.S. in Information Security Engineering at Soonchunhyang University, Republic of Korea. Also, he is a member of the Lab. of Information Systems Security Assurance (LISA) leaded by Prof. Kangbin Yim. His research interests include vulnerability analysis, obfuscation, systems security, access control and insider threats.

Junyoung Park is currently working towards his B.S. in Information Security Engineering at Soonchunhyang University, Republic of Korea. Also, he is a member of the Lab. of Information Systems Security Assurance (LISA) leaded by Prof. Kangbin Yim. His research interests include vulnerability analysis, kernel mode root kit, systems security, access control and insider threats.

Kyungroul Lee received his B.S and M.S. degrees from Soonchunhyang University, Republic of Korea in 2008 and 2010 respectively. He is currently working towards his Ph.D. degree in the same university. His research interests include vulnerability analysis, kernel mode root kit, obfuscation, systems security, access control and insider threats.

Ilsun You received his M.S. and Ph.D. degrees in Computer Science from Dankook University, Seoul, Korea in 1997 and 2002, respectively. Since March 2005, he has been an Assistant Professor in the School of InformationScience at the Korean Bible University, South Korea. Dr. You served or is currently serving on the organizing or program committees of international conferences and workshops including IMIS'07-13, MIST'09-12, MobiWorld'08-12, and so forth. Also, he has served as a guest editor for more than 10 international journals. He is on the editorial boards of International Journal of Ad Hoc and Ubiquitous Computing (IJAHUC), Computing and Informatics (CAI), and Journal of Korean Society for Internet Information (KSII). His main research interests include mobile Internet security and formal security veri?cation.

Kangbin Yim received his B.S., M.S., and Ph.D. from Ajou University, Suwon, Korea in 1992, 1994 and 2001, respectively. He is currently an associate professor in the Department of Information Security Engineering, Soonchunhyang University. He has served as an executive board member of Korea Institute of Information Security and Cryptology, Korean Society for Internet Information and The Institute of Electronics Engineers of Korea. He also has served as a committee chair of the international conferences and workshops and the guest editor of the journals such as JIT, MIS and JoWUA. His research interests include vulnerability assessment, code obfuscation, malware analysis, leakage prevention, secure platform architecture and mobile security. Related to these topics, he has worked on more than forty research projects and published more than ninety research papers.