

Research on Software Protection Technology Based on Driver

Zhu Hao^{1,*}, Kong Qiongying², Xu Zexin¹, Chen Jiwei², Li Xian¹

¹Qujing No. 1 Middle School, Qujing, China

²Faculty of Information Engineering and Automation, Kunming University of Science and Technology, Kunming, China

Email address:

Zhu102@nenu.edu.cn (Zhu Hao), 310345884@qq.com (Kong Qiongying), 2291743295@qq.com (Chen Jiwei)

*Corresponding author

To cite this article:

Zhu Hao, Kong Qiongying, Xu Zexin, Chen Jiwei, Li Xian. Research on Software Protection Technology Based on Driver. *American Journal of Information Science and Technology*. Vol. 4, No. 3, 2020, pp. 46-50. doi: 10.11648/j.ajist.20200403.12

Received: May 13, 2020; **Accepted:** August 5, 2020; **Published:** August 18, 2020

Abstract: Recent years, with the rapid development of the Internet, the technology of software and hardware changes with each passing day. In order to pursue the economic interest, many software systems which contains fatal flaws are always come into use untimely. Although many software developers have involved a tremendous lot of work to make the life cycle of their software systems long enough. However, the law is strong but the outlaws are ten times stronger. In order to be able to illegally use software related charging functions, hackers improve their illegal cracking techniques. in the process of confronting software protection technology As many software developers only focus on the implementation of software system functions, they overlooked the software encryption protection and reverse cracking. Therefore, in the preliminary stage of studying software protection, researchers developed some relatively useful professional software encryption protection program (Shell for short). However, with the development of cracking techniques, even the strong shell ASProtect which uses powerful encryption algorithms such as Twofish, TEA, Blowfish, and the combination of CRC (Cyclic Redundancy Check) and anti-debugging techniques can be removed by using the free OllyDbg dynamic tracking shell after the disassembly code. Using the stack balance principle to find the shell before the program execution entrance, then combining the powerful functions of LoadPE tool to import table, import address table and relocation table. Presently, VMProtect and driver protection technology are two most important ways to protect software. However, VMProtect will need large amount of code in order to build virtual machines which will act as decoders of bytecode - code generated to protect software. For the same reason, efficiency of executing software protected by VMProtect is very low. This article will introduce current state of software protection and give suggestions to limitation found in current application.

Keywords: Driver protection, Kernel Reboot, Encryption, SSDTHOOK

1. Introduction

Before software protection technology, we can only protect software at level ring3 in the system by means of encrypting Import Tables, IATs and Relocatables. Or encrypt important DLL file so that hackers may have a hard time reverse engineer encryption [6]. However, protected data will be fully exposed to hackers as long as those hackers use dynamic debuggers like OllyDbg to keep track of encryption process that happened in the CPU. It is proven that good encryption program like themida is still vulnerable before adding in software protection [1]. Since drivers are running at ring 0 level, which is the same level

as operating system. Thus, it is allowed to edit any data from 4GB virtual memories [13]. We can protect and hide important data of a software when we use driver program to encrypt it. In the meantime, we can use GDTHOOK, IDTKOOK, SSDTHOOK to increase clearance of software and relocate key codes (decryption code for example) to level ring0. As a result, debuggers that running at ring3 level like OllyDbg won't work, hackers could only use debuggers at ring0 level to do the hacking job [12]. However, debuggers at ring0 level like windbg are way inferior than that in ring3 level. Moreover ring0 level debuggers require more advanced skill level from hackers, which in turn, increased difficulty of decryption greatly [2].

2. Current State of Driver Protection

Current main stream driver protection technologies are SSDTHOOK, kernel reboot technology and APC protection. Abnormality disposal system use APC function or abnormality disposal function to protect important data [14]. World famous TP (TenProtect) uses SSDTHOOK to hide important kernel functions. It creates Deep Inline Hook from NtOpenProcess, NtOpenThread, NtReadVirtualMemory, NtWriteVirtualMemory, KiAttachProcess etc. it also generates special threads in order to test kernel functions for abnormalities and modifications constantly. However, kernel testing tools like XueTr, Kernel Detective, PCHunter could still identify hooked functions easily [5]. Then hackers can reverse engineer those functions and break TP's encryption. The difference between SSDTHOOK and kernel reboot is that making a new copy of the kernel to the memory then activate the HOOK process will bypass tools like XueTr [11]. For the same reason, kernel reboot technology is more effective in driver protection.

3. Process and Restrictions of Kernel Rebooting

3.1. Cloning the Kernel

Code has to be stored in memories in order to be executed

by the CPU [8]. The first step of kernel rebooting is to copy kernel files (if operating system uses 10-10-12, then page kernel file is ntoskrnl.exe; if uses 2-9-9-23, then page kernel file is ntkrnlpa.exe; if program uses window function, then reloading win32k.sys is needed) to a newly allocated memory, just like how operating system loads PE files [10]. Process above is demonstrated below (take ntoskrnl.exe for example).

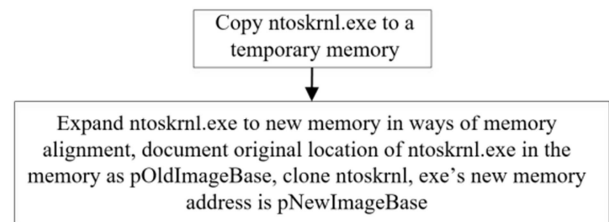


Figure 1. Cloning the Kernel.

3.2. Repair Relocation Table

There are many global variables in ntoskrnl.exe (like KeServiceDescriptorTable) and address of functions (like KiAttachProcess). They are all directly accessible. Those address in clones are the same as before, thus, system will crash if we forget to repair those addresses [9]. Sometimes, it will cause hardware failure. The whole process is shown below.

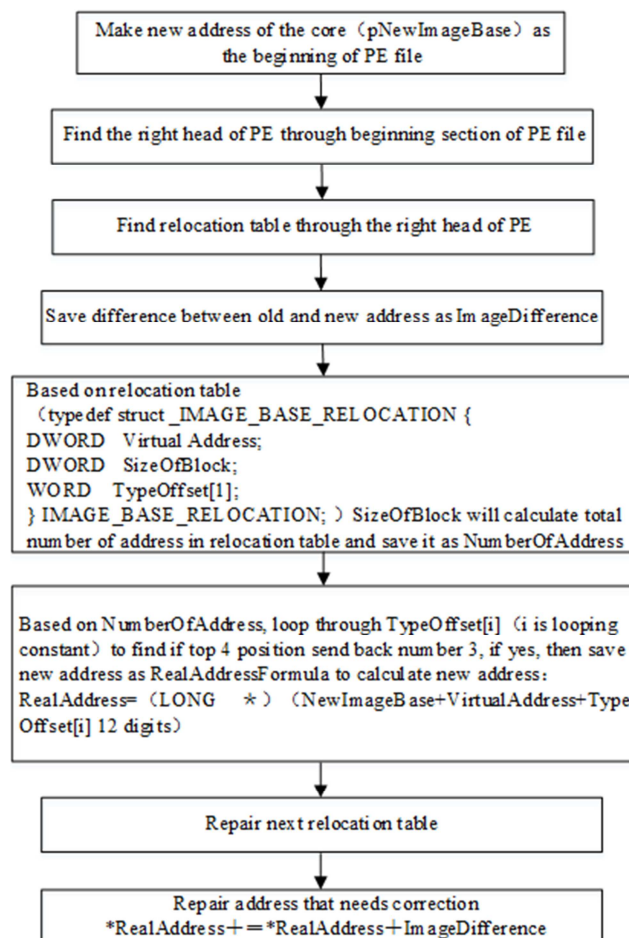


Figure 2. Repair Relocation Table.

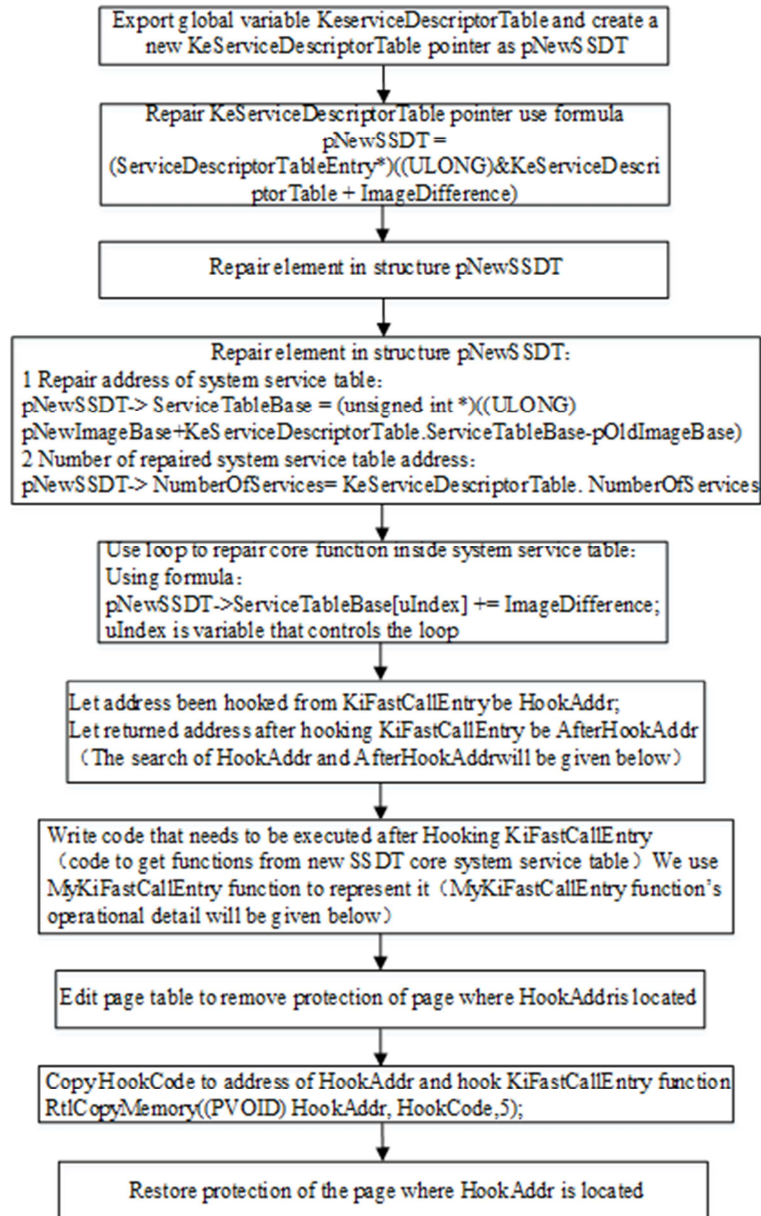


Figure 3. Repair Address of System Service Table.

3.3. Repair KeService Descriptor Table Pointer Address of System Service Table Address of Kernel Functions Inside SSD

After repairing relocation table, when we use API(Application Programming Interface) from ring3 to enter ring0, KeServiceDescriptorTable pointer, address of system service table and address of kernel functions inside SSDT [15]. They all point to the original data as process chart shown below.

3.4. Hook KiFastCallEntry Kernel Function

Now our new kernel has new KeServiceDescriptorTable pointer (pNewSSDT). Data in new system service table are pointed to kernel functions in our new kernel. As we know, when using API from ring3 to enter ring0, KiFastCallEntry function is still calling for functions in the old system service

table located in the old kernel [3]. So we need to Inline Hook KiFastCallEntry function. KiFastCallEntry will allocate pNewSSDT->ServiceTableBase new kernel functions from new system service table when we designated that process. In this way, kernel is reloaded as process chart shown below. We use method of stack backtrace in order to determine addresses that were hooked in KiFastCallEntry [4]. First we use SSDTHOOK to hook a kernel function randomly. Take NtOpenProcess function for example. After hooking NtOpenProcess, we will be entering function MyNtOpenProcess (defined by ourselves) when using OpenProcess at level ring3 to enter ring0. Functions mentioned above is shown below (Key code shown below). NTSTATUS MyNtOpenProcess (_out PHANDLE ProcessHandle, _in ACCESS_MASK DesiredAccess, _in POBJECT_ATTRIBUTES ObjectAttributes,

```

__in_opt PCLIENT_ID ClientId
)
{
    ULONG AddrReturnToKiFastCallEntry;
    UCHAR *HookAddr=NULL;
    int i=0;
    __asm
    {
        pushad
        mov eax,[ebp+0x04]
        mov AddrReturnToKiFastCallEntry,eax
        popad
    }
    HookAddr=(UCHAR *) AddrReturnToKiFastCallEntry;
    for(i=0;i<100;i++)
    {
        if(*p==0x2b&&*(p+1)==0xe1&&*(p+2)==0xc1&&*(p+3)=
        =0xe9&&*(p+4)==0x02)
        {
            addr_hookaddr=(ULONG)p;
            break;
        }
        else
        {
            p--;
        }
    }
    PageProtectOff();
    KeServiceDescriptorTable.ServiceTableBase[122]
    =(unsigned int) OriginalNtOpenProcessAddr;
    PageProtectOn();
}
return ((NTOPENPROCESS)
OriginalNtOpenProcessAddr)(ProcessHandle,DesiredAccess,
ObjectAttributes,ClientId);
}

```

When we just enter MyNtOpenProcess function, [ebp+4] address slot is occupied by AddrReturnToKiFastCallEntry, which is the address returned to KiFastCallEntry after function NtOpenProcess finishes processing [7]. Then, we use AddrReturnToKiFastCallEntry to look back. Though pattern matching, we can find locations to Hook (HookAddr) in KiFastCallEntry. We pick position XX and XX in KiFastCallEntry because storage Ebx, Eax and Edi are written with addresses of kernel functions from original system service table, index and address of original system service table. We Hook KiFastCallEntry function here so that we can find relating addresses of kernel functions in the new kernel based on data stored in Ebx, Eax and Edi. MyKiFastCallEntry code is shown below.

```

void MyKiFastCallEntry()
{
    __asm
    {
        pushad
        pushfd
        push ebx

```

```

        push eax
        push edi
        call GetAddress
        mov [esp+0x14],eax
        popad
        sub esp,ecx
        shr ecx,2
        jmp AfterHookAddr
    }
}
ULONG GetAddress(ULONG ServiceTableBase,ULONG
FuncIndex,ULONG OrigFuncAddress)
{
    if
    (ServiceTableBase==(ULONG)KeServiceDescriptorTable.Se
rviceTableBase)
    {
        if
        (!strcmp((char*)PsGetCurrentProcess()+0x174,"cheatengi
ne-i38"))
        {
            return pNewSSDT->ServiceTableBase[FuncIndex];
        }
    }
    return OrigFuncAddress;
}

```

After hooking KiFastCallEntry, we can find and use functions located in new system service table based on pNewSSDT, Ebx, Eax and Edi.

3.5. Limitations of Kernel Rebooting Protection

After rebooting the kernel, when we are SSDT HOOKING kernel functions from new system service table, hackers won't detect our hooking process using tools like XueTr, Kernel Detective, PCHunter etc. In this way, SSDT HOOK is still effective in protecting software and drivers. Although kernel rebooting could save us from kernel detecting tools, it is still vulnerable against pattern matching. Because files in new kernel contain patterns that could reveal identities, we will be calling such patterns as "kernel fingerprints" in the rest. Patterns like PE symbol, kernel functions will allow hackers to use byte-search to search memories. If PE fingerprints are found in a memory (0X4D 5A), other kernel fingerprints like NtOpenProcess are found or extra processes done by kernel function were found, hackers would acquire strong evidence that the kernel was rebooted. Apparently, hackers can reverse engineer KiFastCallEntry function and figure out the differences after software protection program was executed. Then they could find out where KiFastCallEntry was hooked, all they have to do is to unhook KiFastCallEntry and all protections are decrypted. There are way too many fingerprints in new kernel files; in that case, we need to find a way to hide those fingerprints as much as possible to prevent hackers from finding real identities of those files. Hackers can't break the encryption when they failed to find fingerprints using pattern matching technique. So we need

to encrypt kernel fingerprints before kernel rebooting (for example, xor fingerprints). Using this method, when new kernel is expanded to the memory, files inside the kernel won't show fingerprints. Rather, they look like ordinary data blocks that have been encrypted. When we need to execute functions from new kernel, we can then simply decrypt the new kernel and run software encrypting program then to protect the code.

4. Summary

4.1. Limitations of Kernel Rebooting Protection

Although the drive protection scheme mentioned in the essay can provide an effective protection for the software system, yet the skills require to accomplish by the software developers, and the programming techniques have a relative barriers to entry. So, if the developers want to use the system mentioned by the author to protect the software, it requires to understand the working principles and processes of windows kernel code, which makes the software development harder and extending the period of developing.

4.2. Limitations of Kernel Rebooting Protection

Due to the limitation of the energy and ability, the author hasn't done the experimental demonstration of the software-driven protection scheme. so the author expect to accomplish it as soon as possible, by the way those who interested in the demonstration are welcomed.

4.3. Limitations of Kernel Rebooting Protection

The intellectual property of the software has been severely infringed for a long time. Reverse software crack has damaged the profit of developers seriously. It is hoped that crackers will reduce the persecution on the intellectual property through the research of this paper and this research can inspire those who are interested in the skills of software driver.

References

- [1] Li Ruofeng. Research on Network Intrusion Detection Technology Based on Windows Driver Filtering [D]. China University of Mining and Technology, 2019.
- [2] Meng Chenyu, Shi Yuan, Wang Jiawei, Zhou Jie, Kang Xiaofeng. Windows kernel-level protection system [J]. Software, 2016, 37 (03): 16-20+26.
- [3] Duan Zhixiu. Study Of Software IP Protection [D]. Lanzhou University, 2019.
- [4] Chen Xiaoting. Study of Whether To Forbid Reverse Engineer Software Is Legal——Thoughts Derived From AWS Customer Agreement Regarding Dos and Don'ts [J]. Science Publication, 2019, 27 (03): 59-64.
- [5] Xu Feng. Design and implementation of security monitoring software in Windows x64 system environment [D]. Beijing University of Posts and telecommunications, 2019.
- [6] Dong Jianye. Anti Reverse Engineering In Software Industries [A]. China Institute of Communications Communications Technology Safety Board. 2010 Collections Of Articles Regarding Communication Safety [C]. China Institute of Communications Communications Technology Safety Board: China Institute of Communications, 2010: 5.
- [7] Yi Xiangchen. Security software process protection and reinforcement technology based on Windows system [D]. Tianjin University, 2018.
- [8] Ma HongLi. Study Of Software Protection Based On Windows Kernel [D]. Huazhong University of Science and Technology, 2012.
- [9] Ni Tao. Safety Test On Kernel Drivers Based On Windows Kernel [J]. Informations And Communications, 2017 (01): 183-184.
- [10] Meng ChenYu, Shi Yuan, Wang JiaWei, Zhou Jie, Kang XiaoFeng. Protection System Of Windows Kernel [J]. Software, 2016, 37 (03): 16-20+26.
- [11] Zhao Xiaohua, Zhao Shusheng. User behavior collection solution based on Windows Kernel [J]. Software engineering, 2018, 21 (07): 28-31.
- [12] Wu Jian. Research on 64-bit Windows operating system kernel monitoring [D]. Xiangtan University, 2016.
- [13] C. Basile, D. Canavese, L. Regano, P. Falcarin, B. De Sutter. A Meta-model for Software Protections and Reverse Engineering Attacks [J]. The Journal of Systems & Software, 2018.
- [14] Principle Of Reverse Engineering [M]. Posts & Telecom Press, (Korean) Licheng Yuan, 2014.
- [15] Detailed Analysis Of Windows Kernel [J]. Publishing House of Electronics Industry, Mao DeCao, 2009.