Defeating Kernel Native API Hookers by Direct KiServiceTable Restoration

Tan Chew Keong

Vice-President SIG2

www.security.org.sg

Outline

- User-space API calls and Native APIs
- Redirecting the execution path of Native APIs
- Locating and restoring the KiServiceTable
- Defeating Native API hooking rootkits and security tools.

User-space API calls

- User-space Win32 applications requests for system services by calling APIs exported by various DLLs.
- For example, to write to an open file, pipe or device, the WriteFile API, exported by kernel32.dll is used.

User-space API calls

- WriteFile API will in turn call the native API NtWriteFile/ZwWriteFile that is exported by ntdll.dll
- In ntdll.dll, both NtWriteFile and ZwWriteFile points to the same code.
- Actual work is actually performed in kernel-space, hence NtWriteFile/ZwWritefile in ntdll.dll contains only minimal code to transit into kernel code using software interrupt INT 0x2E

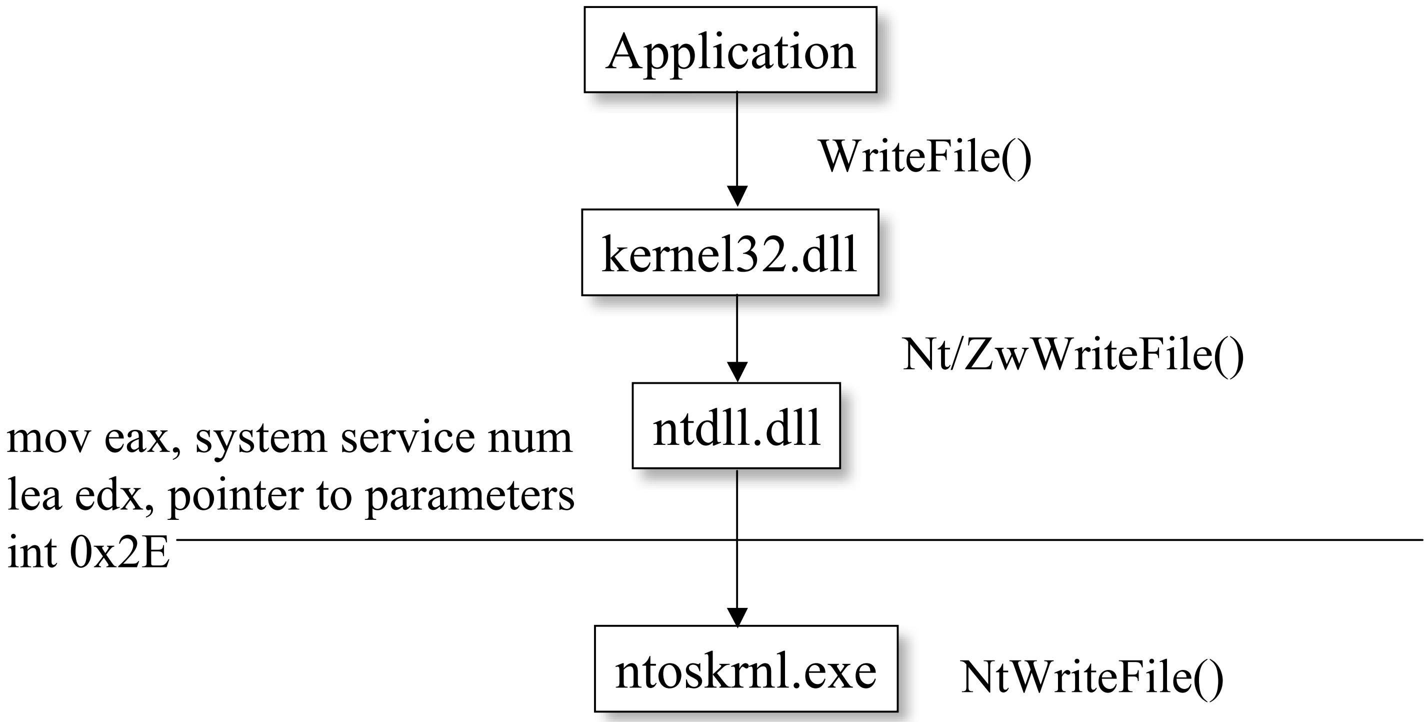
User-space API calls

- In Win2k, NtWriteFile/ZwWriteFile in ntdll.dll disassembles to

```
MOV EAX, 0EDh
LEA EDX, DWORD PTR SS:[ESP+4]
INT 2E
RETN 24
```

- 0xED is the system service number that will be used to index into the KiServiceTable to locate the kernel function that handles the call.

User-space API calls



User-space API calls

- In WinXP, NtWriteFile/ZwWriteFile in ntdll.dll disassembles to

```
MOV EAX, 112h
MOV EDX, 7FFE0300h
CALL DWORD PTR DS:[EDX]
RETN 24
```

- At 7FFE0300h is a pointer to the following function.

```
MOV EDX, ESP
SYSENTER
```

System Service Dispatcher

- The ISR for INT 2E is KiSystemService
- KiSystemService checks the input parameters and calls the correct system service based on entries in the System Service Dispatch Table (KiServiceTable)

KiServiceTable

0	NtAcceptConnectPort
1	NtAccessCheck
2	NtAccessCheckAndAuditAlarm
	…
0xED	**NtWriteFile**
N	

System Service Dispatch Table (SSDT)

KeServiceDescriptorTable

- KiServiceTable is not exported by the kernel.
- However, it's address can be located in KeServiceDescriptorTable.

```
typedef struct ServiceDescriptorTable {
  SDE ServiceDescriptor[4];
} SDT;

typedef struct ServiceDescriptorEntry {
     PDWORD KiServiceTable;
     PDWORD CounterTableBase;
     DWORD ServiceLimit;
     PBYTE ArgumentTable;
} SDE;
```

KeServiceDescriptorTable

- More precisely,

```
KeServiceDescriptorTable.ServiceDescriptor[0].KiServiceTable
```

- ServiceLimit gives the number of entries in the KiServiceTable.
- The rest, ServiceDescriptor[1], ServiceDescriptor[2] and ServiceDescriptor[3] in KeServiceDescriptorTable are not used.

Kernel Native API Hookers

- A device driver that is loaded into kernel-space can modify entries within the KiServiceTable.
- The entries are modified to point to hook functions within the device driver.
- These hook functions can modify the behaviour of the native APIs.

Kernel Native API Hookers

- When modifying KiServiceTable entries, the driver will usually keep a copy of the original entry so that the original native API can be called.

Kernel Native API Hookers

- Pseudo-code

```
NTSYSAPI NTSTATUS NTAPI
  NtXXXHook(.....)
  {
        ManipulateInputParameters(...);
        NtXXXOriginal(....);
        ManipulateReturnBuffers(...);
        return;
  }
```

Kernel Native API Hookers

- Modification of KiServiceTable entries is usually done in DriverEntry.

```
#define SYSTEMSERVICE(_api)
    KeServiceDescriptorTable.ServiceDescriptor[0].ServiceTable[*(DWORD *) ((unsigned
    char *)_api + 1)]

// keep a copy of the original function pointer
 NtWriteFileOrig = (NTWRITEFILE) (SYSTEMSERVICE(ZwWriteFile));

// modifying KiServiceTable entry to point to supplied hook function
// NtWriteFileHook
(NTWRITEFILE) (SYSTEMSERVICE(ZwWriteFile)) = NtWriteFileHook;
```

Kernel Native API Hookers

- In Win2K kernel, ZwWriteFile disassembles to

```
mov     eax, 0EDh
lea     edx, [esp+arg_0]
int     2Eh
retn    24h
```

- Hence, in

```
#define SYSTEMSERVICE(_api)
   KeServiceDescriptorTable.ServiceDescriptor[0].ServiceTable[*(DWORD *)
   ((unsigned char *)_api + 1)]
```

- Code highlighted in red actually retrieves the system service number.
- The actual implementation of the system service is in NtWriteFile.

Process Hiding

- User-space programs use APIs exported by ToolHelp.DLL to obtain list of processes.
- ToolHelp DLL APIs call Nt/ZwQuerySystemInformation exported by ntdll.dll
- User-space programs can also call Nt/ZwQuerySystemInformation directly with SystemProcessesAndThreadsInformation as its first parameter.

Process Hiding

- Kernel rootkit hooks NtQuerySystemInformation and watches for calls with SystemProcessesAndThreadsInformation as its first parameter.
- Modifies return buffer to remove processes to be hidden.
- Return modified buffer to caller.

Process Hiding

- Pseudo-code

```
NTSYSAPI NTSTATUS NTAPI
   NtQuerySystemInformationHook(
               IN ULONG SystemInformationClass,
               IN OUT PVOID SystemInformation,
               IN ULONG SystemInformationLength,
               OUT PULONG ReturnLength OPTIONAL)
   {
     ntS = NtQuerySystemInformationOrig(....);

     if(ntS == STATUS_SUCCESS &&
        SystemInformationClass == SystemProcessesAndThreadsInformation)
     {
               ManipulateReturnBuffers(SystemInformation,...);
     }
     return ntS;
   }
```

Process Hiding

- Return buffer is an array of the following structure.

```
typedef struct _SYSTEM_PROCESS_INFORMATION {
    ULONG           NextEntryDelta;         // offset to the next entry
    ULONG           ThreadCount;            // number of threads
    ULONG           Reserved1[6];           // reserved
    LARGE_INTEGER   CreateTime;             // process creation time
    LARGE_INTEGER   UserTime;               // time spent in user mode
    LARGE_INTEGER   KernelTime;             // time spent in kernel mode
    UNICODE_STRING  ProcessName;            // process name
    KPRIORITY       BasePriority;           // base process priority
    ULONG           ProcessId;              // process identifier
    ULONG           InheritedFromProcessId; // parent process identifier
    .....
    .....
SYSTEM_THREAD_INFORMATION Threads[1];    // threads
} SYSTEM_PROCESS_INFORMATION, * PSYSTEM_PROCESS_INFORMATION;
```

Loaded Driver Hiding

- Kernel rootkits that load into kernel-space as drivers can hide themselves from the list of loaded drivers.
- User-space program can obtain list of loaded drivers by calling Nt/ZwQuerySystemInformation native API, specifying SystemModuleInformation as its first parameter.

Loaded Driver Hiding

```
typedef NTSYSAPI NTSTATUS (*ZWQUERYSYSTEMINFORMATION)(
        IN ULONG SystemInformationClass,
        IN OUT PVOID SystemInformation,
        IN ULONG SystemInformationLength,
        OUT PULONG ReturnLength OPTIONAL);
```

- User-space program provides output buffer in the second parameter.
- When function returns, first DWORD in buffer gives the number of array entries returned.
- The remaining bytes in the buffer is an array of the following structure.

Loaded Driver Hiding

```
typedef struct _SYSTEM_MODULE_INFORMATION
{
        ULONG Reserved[2];
        PVOID ImageBase;
        ULONG ImageSize;
        ULONG Flags;
        USHORT Index;
        USHORT Unknown;
        USHORT LoadCount;
        USHORT ModuleNameOffset;
        CHAR ImageName[256];
} SYSTEM_MODULE_INFORMATION;
```

Loaded Driver Hiding

- Kernel rootkit hooks NtQuerySystemInformation and watches for calls with SystemModuleInformation as its first parameter.
- Modifies return buffer to remove driver to be hidden.
- Return modified buffer to caller.

File Hiding

- Kernel rootkit hooks NtQueryDirectoryFile
- Modifies return buffer to remove file(s) to be hidden.
- Return modified buffer to caller.

Kernel Native API Hookers

- How to detect hooked system services?
  - SSDT will contain entries that point outside the kernel's image.
  - We can determine the list of drivers and their base address using Nt/ZwQuerySystemInformation

```
80400000 001A2340 - \WINNT\System32\ntoskrnl.exe
80062000 00010460 - \WINNT\System32\hal.dll
ED410000 00003000 - \WINNT\System32\BOOTVID.DLL
BFFD8000 00028000 - ACPI.sys
ED5C8000 00001000 - \WINNT\System32\DRIVERS\WMILIB.SYS
```

Kernel Native API Hookers

```
kd> d KeServiceDescriptorTable
8046dfa0  b8 42 47 80 00 00 00 00-f8 00 00 00 9c 46 47 80  .BG.........FG.   ServiceDescriptor[0]
8046dfb0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................   ServiceDescriptor[1]
8046dfc0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................   ServiceDescriptor[2]
8046dfd0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................   ServiceDescriptor[3]

kd> d 804742b8
804742b8  52 dd 49 80 c1 f6 4a 80-3a 04 4b 80 b8 d5 50 80  R.I...J.:.K...P.
804742c8  70 04 4b 80 a2 ce 45 80-be f7 50 80 fe f7 50 80  p.K...E...P...P.
804742d8  38 4a 49 80 f2 a9 50 80-d8 de 4a 80 2d d8 4f 80  8JI...P...J.-.O.
804742e8  49 a6 4a 80 df 4d 49 80-ca b8 44 80 3d 7e 4c 80  I.J..MI...D.=~L.
804742f8  74 ee 5b f7 a9 94 4b 80-e0 db 4f 80 1a 19 40 80  t.[...K...O...@.
80474308  1c 04 4d 80 76 95 41 80-86 6d 4f 80 8e 35 49 80  ..M.v.A..mO..5I.
80474318  a8 f9 44 80 57 07 4b 80-d5 e2 49 80 75 dc 49 80  ..D.W.K...I.u.I.
80474328  a0 92 46 80 84 4b 4f 80-d2 35 49 80 40 92 4c 80  ..F..KO..5I.@.L.
```

Kernel Native API Hookers

- Base on the address in the KiServiceTable, we can determine the driver hooked that System Service.

```
80400000 - \WINNT\System32\ntoskrnl.exe
80062000 - \WINNT\System32\hal.dll
....
BF8B3000 - \SystemRoot\System32\Drivers\Cdfs.SYS
BF6F4000 - \SystemRoot\System32\Drivers\Fastfat.SYS
BF74F000 - \SystemRoot\System32\DRIVERS\ipsec.sys
F75BA000 - \SystemRoot\System32\DRIVERS\KProcCheck.sys
F75BE000 * \SystemRoot\System32\DRIVERS\gotr.sys
```

KProcCheck POC

- KProcCheck v0.1 – POC tool that checks for hooked System Services.

```
C:\>kproccheck -t
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

Checks SDT for Hooked Native APIs

ZwAllocateVirtualMemory    10 \SystemRoot\System32\DRIVERS\gotr.sys [F75BEE74]
ZwCreateFile               20 \SystemRoot\System32\DRIVERS\gotr.sys [F75BEA85]
ZwCreateKey                23 \SystemRoot\System32\DRIVERS\gotr.sys [F75BEC5E]
ZwCreateProcess            29 \SystemRoot\System32\DRIVERS\gotr.sys [F75BEDB7]
ZwDeleteFile               34 \SystemRoot\System32\DRIVERS\gotr.sys [F75BE80C]
ZwGetTickCount             4C \SystemRoot\System32\DRIVERS\gotr.sys [F75BEE27]
ZwLoadDriver               55 \SystemRoot\System32\DRIVERS\gotr.sys [F75BEBF2]
ZwQueryDirectoryFile       7D \SystemRoot\System32\DRIVERS\gotr.sys [F75BE6E8]
ZwQuerySystemInformation   97 \SystemRoot\System32\DRIVERS\gotr.sys [F75BE623]
ZwSetInformationFile       C2 \SystemRoot\System32\DRIVERS\gotr.sys [F75BE8A8]

Number of Service Table entries hooked = 10
```

KProcCheck POC

- KProcCheck consist of two components
  1. KProcCheck.exe, user-space program
  2. KProcCheck.sys, a driver
- The driver KProcCheck.sys compares and checks for hooked System Services in KiServiceTable.
- KProcCheck.exe displays results to user.
- http://www.security.org.sg/code/kproccheck.html

KProcCheck POC

- KProcCheck.sys does not use NtQuerySystemInformation(SystemModuleInformation, ...) to obtain list of loaded drivers.
- It might be hooked....
- Instead, the list is obtained by traversing PsLoadedModuleList directly.

KProcCheck POC

- In Win2k kernel, PsLoadedModuleList is the head of a doubly linked-list of the following.

```
struct {
  LIST_ENTRY link;          // Flink, Blink
  BYTE unknown1[16];
  DWORD imageBase;
  DWORD entryPoint;
  DWORD imageSize;
  UNICODE_STRING drvPath;
  UNICODE_STRING drvName;
  ...
}
```

KProcCheck POC

- But PsLoadModuleList is not exported.
- KProcCheck finds the address of PsLoadedModuleList by scanning the exported function MmGetSystemRoutineAddress for the following instructions.

```
8b35b8e14680      mov      esi,[nt!PsLoadedModuleList (8046e1b8)]
81feb8e14680      cmp      esi,0x8046e1b8
```

KProcCheck POC

- KProcCheck also compares the results of calling Nt/ZwQuerySystemInformation(SystemModuleInformation, ...) with the list obtained by traversing PsLoadedModuleList
- This will reveal drivers that are hidden by hooking NtQuerySystemInformation.

KProcCheck POC

```
C:\>kproccheck -d
KProcCheck Version 0.1 Proof-of-Concept by SIG^2 (www.security.org.sg)

80400000 - \WINNT\System32\ntoskrnl.exe
80062000 - \WINNT\System32\hal.dll
F7410000 - \WINNT\System32\BOOTVID.DLL
F7000000 - pci.sys
F7010000 - isapnp.sys
F7500000 - intelide.sys
F7280000 - \WINNT\System32\DRIVERS\PCIIDEX.SYS
F7288000 - MountMgr.sys
BFFE3000 - ftdisk.sys
...
BF6F4000 - \SystemRoot\System32\Drivers\Fastfat.SYS
BF74F000 - \SystemRoot\System32\DRIVERS\ipsec.sys
F75BA000 - \SystemRoot\System32\DRIVERS\KProcCheck.sys
F75BE000 * \SystemRoot\System32\DRIVERS\gotr.sys --[Hidden]--

Total number of drivers = 73
```

Restoring Hooked Entries

- If we can restore the original values of the KiServiceTable entries, then we can disable kernel rootkits that relies on system service hooking to hide files, processes and drivers.
- But how do we know what're the original values?

Restoring Hooked Entries

- A complete copy of the KiServiceTable can be found in the kernel image file ntoskrnl.exe

```
.data:004742B8 off_0_4742B8    dd offset sub_0_49DD52  ; DATA XREF: sub_0_55A996
.data:004742BC                 dd offset sub_0_4AF6C1
.data:004742C0                 dd offset sub_0_4B043A
.data:004742C4                 dd offset sub_0_50D5B8
.data:004742C8                 dd offset sub_0_4B0470
.data:004742CC                 dd offset sub_0_45CEA2
.data:004742D0                 dd offset sub_0_50F7BE
.data:004742D4                 dd offset sub_0_50F7FE
.data:004742D8                 dd offset NtAddAtom
.data:004742DC                 dd offset sub_0_50A9F2
.data:004742E0                 dd offset NtAdjustPrivilegesToken
.data:004742E4                 dd offset sub_0_4FD82D
.data:004742E8                 dd offset sub_0_4AA649
.data:004742EC                 dd offset NtAllocateLocallyUniqueId
```

Restoring Hooked Entries

- Restoration of hooked entries can be done by
  1. loading a driver, or
  2. directly from user-space by writing to \device\physicalmemory

- Access to \device\physicalmemory allows a user-space program to read/write to physical memory, including kernel memory.

SDTrestore POC

- Our SDTrestore POC code demonstrates the restoration of KiServiceTable entries by writing to \device\physicalmemory
- Using \device\physicalmemory to view physical memory was first used by Mark Russinovich, Sysinternals in his Physmem tool.

\device\physicalmemory

- 90210 – installs a GDT call gate by writing to \device\physicalmemory. Hides process by unlinking it's EPROCESS structure from ActiveProcessLinks.
- crazylord – "Playing with Windows /dev/(k)mem", installs a GDT call gate by writing to \device\physicalmemory and lists processes by traversing ActiveProcessLinks.

\device\physicalmemory

- The follow sequence of steps describe how a user-space program with Administrator privilege can gain read/write access physical memory.

1. Use NtOpenSection native API (exported by ntdll.dll) with SECTION_MAP_READ | SECTION_MAP_WRITE access flags to get a handle to \device\physicalmemory. This will usually fail since the Administrator does not have SECTION_MAP_WRITE access rights to \device\physicalmemory

\device\physicalmemory

2. Use NtOpenSection native API with READ_CONTROL | WRITE_DAC access flag to get a handle to \device\physicalmemory. This allows a new DACL to be added to the \device\physicalmemory object.

3. Add a DACL to \device\physicalmemory, granting SECTION_MAP_WRITE access to the Administrator account.

4. Try to get a handle to \device\physicalmemory again using NtOpenSection native API with SECTION_MAP_READ | SECTION_MAP_WRITE access flags.

\device\physicalmemory

- To write to physical memory, the program must first map the physical memory page into its virtual address space.

```
ntStatus = _NtMapViewOfSection(
     hPhyMem,              // Handle to \device\physicalmemory
     (HANDLE)-1,
     virtualAddr,          // OUT - Virtual memory where the physical memory
                                   is mapped to.
     0,
     *length,
     &viewBase,            // IN/OUT - Physical memory address to map in (page
                                      aligned)
     length,               // IN/OUT - Size of the mapped physical memory
     ViewShare,
     0,
     PAGE_READWRITE        // Map for READ/WRITE access
);
```

\device\physicalmemory

- To restore the KiServiceTable, we need to know the following
  1. Location of KiServiceTable in ntoskrnl.exe kernel file image.
  2. Physical memory address of KiServiceTable

- Cannot hardcode these addresses since they're different between service packs and between Win2K, WinXP.

\device\physicalmemory

- Note NtMapViewOfSection uses physical memory address to map the pages, not virtual address.
- Under Win2k, kernel base is located at 0x80400000 in virtual memory.
- The physical memory address of the Win2k kernel is at 0x400000.

Locating KiServiceTable in Disk Image

- Load ntoskrnl.exe as DLL and get the offset of KeServiceDescriptorTable from it's export table.
- Map in the physical page containing the ServiceDescriptorTable.

PhyAddrServiceDescriptorTable =
    kernelPhysicalBaseAddr + offset_of_KeServiceDescriptorTable

Locating KiServiceTable in Disk Image

- From the mapped page, get the address of KiServiceTable.
- This will be the virtual address of the KiServiceTable in kernel-space, 0x804742b8

```
kd> d KeServiceDescriptorTable
8046dfa0  b8 42 47 80 00 00 00 00-f8 00 00 00 9c 46 47 80  .BG.........FG.
8046dfb0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
8046dfc0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
8046dfd0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
8046dfe0  20 f1 df ff 00 00 00 00-00 00 00 00 00 00 00 00   ...............
8046dff0  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
8046e000  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Locating KiServiceTable in Disk Image

- Need to convert it to offset within the kernel file image.

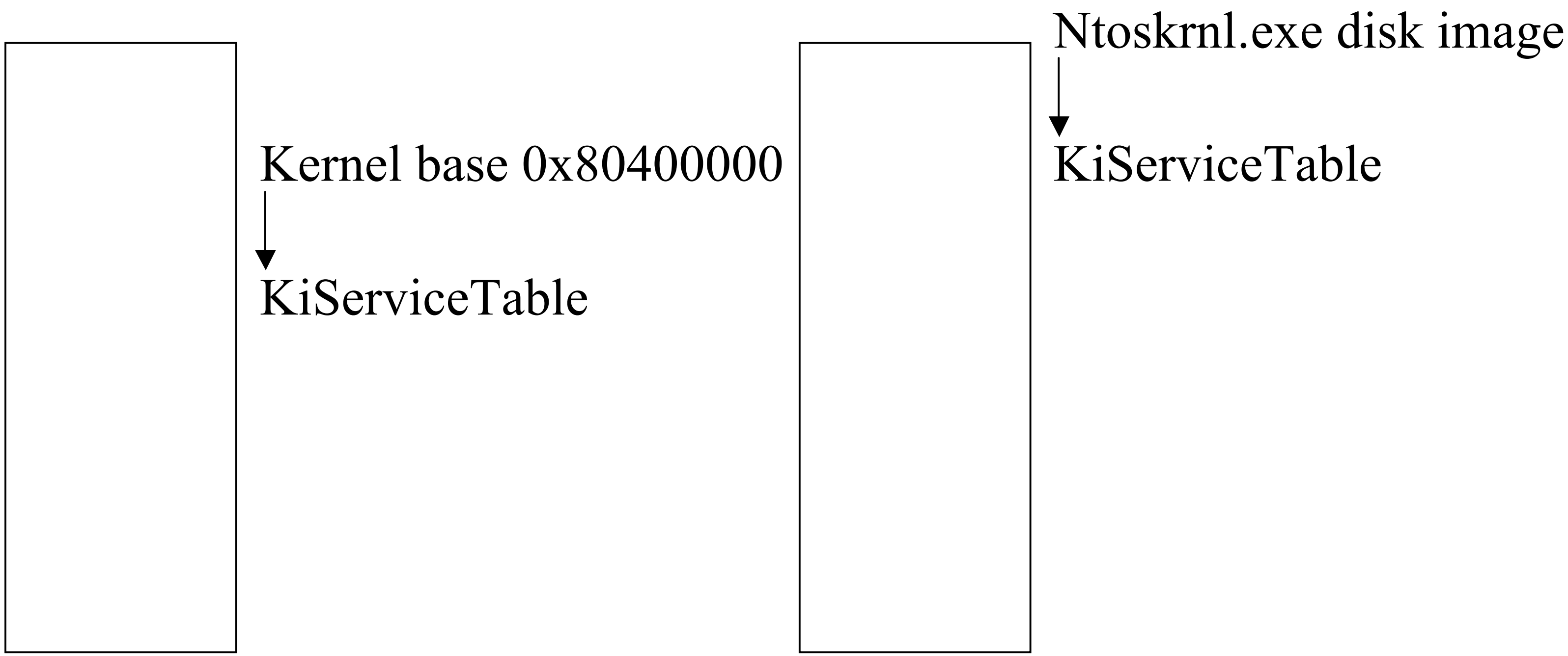
  serviceTableOffset =
  serviceTableVirtualAddr –
  kernelVirtualBaseAddr (0x80400000)

- Using this, we offset into our copy of ntoskrnl.exe to access the original copy of KiServiceTable.

Locating KiServiceTable in Disk Image

- However, this method assumes that the offset of KiServiceTable within the running kernel is the same as the disk image.
- Rootkits or other security tools might have relocated KiServiceTable.
- This will cause the method to fail.

Locating KiServiceTable in Disk Image



Locating KiServiceTable in Disk Image

- 90210 suggested a better way to locate KiServiceTable in the disk image on rootkit.com
- KiServiceTable is referenced in KiInitSystem when the running copy of KeServiceDescriptorTable is being setup.

Locating KiServiceTable in Disk Image

```
INIT:0055AA65          mov     eax, offset dword_0_482258
INIT:0055AA6A          mov     ds:KeServiceDescriptorTable,
                                   offset KiServiceTable
INIT:0055AA74          mov     ds:dword_0_48225C, eax
```

- But KiInitSystem is not exported, so we have to scan ntoskrnl.exe for instructions for the form

```
mov KeServiceDescriptorTable, imm32
```

- It's more efficient and reliable to scan only locations that are referenced by the relocation table.

Locating KiServiceTable in Disk Image

- For each entry in the relocation table, check whether it references KeServiceDescriptorTable.
- If so, check whether it is instruction of the type

```
C7 05
 mov KeServiceDescriptorTable, imm32
```

Patching the KiServiceTable

- After locating the on-disk copy of KiServiceTable, we simply need to map in the physical page containing the KiServiceTable of the running kernel and patch entries that are different.
- Remember to convert the values to kernel virtual address first...

Patching the KiServiceTable

```
for(DWORD i = 0; i < sdtCount; i++)
{
        if((kernelServiceTable[i] - kernelVirtualBase + peXH2.imageBase) !=
                fileServiceTable[i])
        {
                kernelServiceTable[i] = fileServiceTable[i] –
                        peXH2.imageBase + kernelVirtualBase;

                printf("[+] Patched SDT entry %.2X to %.8X\n", i,
                        kernelServiceTable[i]);
        }
}
```

Example Native API Hookers

- Kernel Rootkits
  - NT Rootkit
  - HE4Hook
- Security Tools
  - Sebek Win32
  - DiamondCS Process Guard
  - Kerio Personal Firewall 4

Native API Hooking Rootkit

- HE4Hook hides files by
  1. Hooking the following system services.

```
ZwCreateFile              20 --[hooked by unknown at 81222476]--
ZwOpenFile                64 --[hooked by unknown at 812224A8]--
ZwQueryDirectoryFile      7D --[hooked by unknown at 812224D2]--
```

  2. or, hooking callback tables in file system driver.

- Method 1 can be easily disabled by restoring KiServiceTable entries.

Sebek Win32

- Sebek is a console logger that is commonly used on HoneyPots.
- Captures inputs and outputs of cmd.exe and sends them out as UDP packets to a logging server.
- This allows logging of encrypted cmd.exe sessions.

Sebek Win32

- Sebek hooks the following system services

```
ZwClose                    18 SEBEK.sys [F729A092]
ZwCreateFile               20 SEBEK.sys [F729A98C]
ZwCreateKey                23 SEBEK.sys [F729AD10]
ZwEnumerateKey  3C SEBEK.sys [F729AE02]
ZwEnumerateValueKey        3D SEBEK.sys [F729AA50]
ZwOpenFile                 64 SEBEK.sys [F729A8E6]
ZwOpenKey                  67 SEBEK.sys [F729AD88]
ZwQueryDirectoryFile       7D SEBEK.sys [F729A4CC]
ZwQuerySystemInformation 97 SEBEK.sys [F729A5F0]
ZwReadFile                 A1 SEBEK.sys [F7299CF0]
ZwRequestWaitReplyPort     B0 SEBEK.sys [F7299F14]
ZwSecureConnectPort        B8 SEBEK.sys [F7299FE6]
ZwWriteFile                ED SEBEK.sys [F7299D48]
```

Sebek Win32

- System services are hooked to
  - Hide sebek.sys (antidetection)
  - Hide registry keys that loads sebek.sys (antidetection)
  - ZwReadFile/ZwWriteFile to log cmd.exe
- Restoring KiServiceTable will fully disable sebek's logging and anti-detection capabilities.

DiamondCS Process Guard

- Security tool that protects processes against rogue termination, suspension and prevents loading of malicious kernel drivers.
- Process termination protection is implement by hooking several system services.

DiamondCS Process Guard

```
ZwCreateFile                20 \??\C:\WINNT\System32\drivers\procguard.sys
    [F7392D8A]
ZwCreateKey                 23 \??\C:\WINNT\System32\drivers\procguard.sys
    [F7391F98]
ZwCreateThread              2E \??\C:\WINNT\System32\drivers\procguard.sys
    [F73924FC]
ZwOpenFile                  64 \??\C:\WINNT\System32\drivers\procguard.sys
    [F7392C62]
ZwOpenKey                   67 \??\C:\WINNT\System32\drivers\procguard.sys
    [F7391F64]
ZwOpenProcess               6A \??\C:\WINNT\System32\drivers\procguard.sys
    [F739289E]
ZwOpenThread                6F \??\C:\WINNT\System32\drivers\procguard.sys
    [F73926F8]
ZwRequestWaitReplyPort      B0 \??\C:\WINNT\System32\drivers\procguard.sys
    [F7390AE6]
ZwSetValueKey               D7 \??\C:\WINNT\System32\drivers\procguard.sys
    [F739224E]
ZwWriteVirtualMemory        F0 \??\C:\WINNT\System32\drivers\procguard.sys
    [F7392A40]
```

- Restoring KiServiceTable will disable Process Guard’s process termination protection. BID 10675

Kerio Personal Firewall 4

- KPF prevents malicious code from spawning processes on the user's system by prompting the user for action whenever an unknown/new or modified program is being executed.
- The System Security feature works by hooking the following native APIs.

Kerio Personal Firewall 4

```
ZwCreateFile        20 \SystemRoot\system32\drivers\fwdrv.sys [BFBD3830]
ZwCreateProcess     29 \SystemRoot\system32\drivers\fwdrv.sys [BFBD3380]
ZwCreateThread      2E \SystemRoot\system32\drivers\fwdrv.sys [BFBD35E0]
ZwResumeThread      B5 \SystemRoot\system32\drivers\fwdrv.sys [BFBD3630]
```

- Restoring KiServiceTable will disable Kerio’s process spawn protection.
- BID 11096

Native API Hooking Security Tools

- Security Tools that relies on native API hooking in kernel-space can be disabled by KiServiceTable restoration.
- Need to implement addition protection to prevent this from happening.
  - Prevent driver loading by hooking NtLoadDriver
  - Prevent driver loading hooking NtSetSystemInformation with SystemInformationClass = SystemLoadAndCallImage [13]

Native API Hooking Security Tools

- Prevent write access to \device\physicalmemory by hooking ZwOpenSection.
- Prevent write access to \device\physicalmemory via symbolic link.

Non-Hooking Rootkits

- FU Rootkit [2]
  - Hides driver by unlinking it from PsLoadedModuleList.
  - Hides process by unlinking it from ActiveProcessLinks
- Process Hide – phide [5]
  - Hides process by unlinking it from ActiveProcessLinks. Done via GDT call gate without need to install a driver.

Non-Hooking Rootkits

- Restoring the KiServiceTable will not disable rootkits that do not rely on hooking.

Conclusion

- Kernel rootkits that relies on System Service hooking can be disabled by restoring the KiServiceTable.
- More recent rootkits do not use hooking and are harder to detect/disable.
- Security Tools should not rely solely on System Service hooking to implement their security features.

References

1. Greg Hoglund - original and first public NT ROOTKIT, http://www.rootkit.com
2. fuzen_op - FU Rootkit https://www.rootkit.com/vault/fuzen_op/FU_Rootkit.zip
3. hf - Hacker Defender http://www.rootkit.com/vault/hf/hxdef100.zip
4. joanna - klister
5. 90210, “Process Hide – phide”, 29A#7 magazine, VX Heavens, http://vx.netlux.org/vx.php?id=ep12
6. 90210, “A more stable way to locate real KiServiceTable”, http://www.rootkit.com/newsread.php?newsid=176
7. crazyload, “Playing with Windows /dev/(k)mem”, Phrack 59

References

9. SoBeIt, “Byepass Scheduler List Process Detection”, http://www.rootkit.com/newsread.php?newsid=117
10. firew0rker, “Kernel-mode backdoors for Windows NT”, Phrack 62
11. Opc0de, “How to get some hidden kernel variables without scanning”, http://www.rootkit.com/newsread.php?newsid=101
12. Alex Ionescu, “Getting Kernel Variables from KdVersionBlock, Part 2”, http://www.rootkit.com/newsread.php?newsid=153
13. Greg Hoglund, “Loading Rootkit using SystemLoadAndCallImage”, http://archives.neohapsis.com/archives/ntbugtraq/2000-q3/0114.html

References

14. Tan Chew Keong, "Win2K Kernel Hidden Process/Module Checker 0.1 (Proof-Of-Concept)", http://www.security.org.sg/code/kproccheck.html
15. Tan Chew Keong, "Win2K/XP SDT Restore 0.2 (Proof-Of-Concept)", http://www.security.org.sg/code/sdtrestore.html
16. David A. Solomon, Mark E. Russinovich, "Inside Microsoft Windows 2000 Third Edition"
17. Sven B. Schreiber, "Undocumented Windows 2000 Secrets, A Programmer's Cookbook"
18. Chris Cant, "Writing Windows WDM Device Drivers"