Tricks of the Hackers: System Function Hooking in MS Windows

Dr. Wolfgang Koch

Friedrich Schiller University Jena

Department of Mathematics and Computer Science

Jena, Germany

wolfgang.koch@uni-jena.de

API Hooking

2

In a previous talk I showed, how to perform API hooking:

- change of addresses in the IAT (Import Address Table) of the executable
- by use of DLL-Injection
- works for one process
  (even if IATs of used DLLs are changed too –
  due to “copy on write” mechanism of shared memory)

API Hooking

3

In a previous talk I showed, how to perform API hooking:

used for testing, monitoring and reverse engineering

as well as for altering the behavior of the operating system or of 3rd party products,

without having their source code available

widely used by hackers and other “bad guys”

System Call Hooking

4

API hooking in a process is widely used by hackers etc.

But for writers of malicious software a **system-wide** change of the behavior of the system is still more important → System Call Hooking

for example:
stealth viruses and root kits can conceal their presence by changing `NtReadFile` or `NtCreateFile`
(if an infected file is to be read the altered system function presents the original file)

Literature Books

5

The Windows Kernel Book:

Mark E. Russinovich
David A. Solomon
Windows Internals, 5th Edition
Covering Windows Server 2008
and Windows Vista

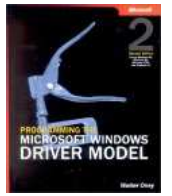
Redmond, Wash : Microsoft Press, 2009
ISBN-13: 978-0735625303

1264 p.

Literature Books

6

The WDM Bible:



Walter Oney
Programming the Microsoft Windows
Driver Model
2nd edition

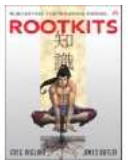
Redmond, Wash : Microsoft Press, 2003
ISBN: 0-7356-1803-8

846 p. + CD-ROM

Literature Books

7

Rootkits:



Greg Hoglund
Jamie Butler
Rootkits:
Subverting the Windows Kernel

Amsterdam : Addison-Wesley, 2005
ISBN-13: 978-0321294319

352 p.

Literature Books

8

"The Windows-API Book":

Jeffrey Richter,
Christophe Nasarre

WINDOWS via C/C++
5th edition

Redmond, Wash : Microsoft Press, 2008
ISBN-13: 978-0-7356-2424-5

820 p. + Companion content Web page

Literature

9

Martin Hinz:

Profiling Windows System Call Activity (in German)
Studienarbeit, Technische Universität Chemnitz, 2006
http://rtg.informatik.tu-chemnitz.de/docs/da-sa-txt/sa-mhin.pdf

Robert Kuster:

Three Ways to Inject Your Code into Another Process, 2003
http://www.codeproject.com/KB/threads/winspy.aspx

Wolfgang Koch:

Tricks of the Hackers: API Hooking and DLL Injection
held at ELTE Budapest, September 2009

Literature

10

Anton Bassov:

Process-wide API spying - an ultimate hack, 2004
http://www.codeproject.com/KB/system/api_spying_hack.aspx

Hooking the native API and controlling process creation
on a system-wide basis, 2005
http://www.codeproject.com/KB/system/soviet_protector.asp

Newsgroups:

comp.os.ms-windows.programmer.nt.kernel-mode
microsoft.public.development.device.drivers

System Call Hooking

11

A lot of API-functions need assistance of the OS kernel –
which runs in **kernel mode** with no restrictions –

for example Input/Output, file access, process management,
memory management …

These functions invoke "**System Services**" –
If you **open and read a file** using `open()` and `read()`
(or using the Windows-API functions
`CreateFile` and `ReadFile`)
the **system functions** `NtCreateFile` and
`NtReadFile` are finally called.

System Call Hooking

12

The system functions `NtCreateFile` and `NtReadFile`
are finally called

→ System Call Hooking

Difficulties:

you have to write a kernel-mode driver

installing a driver without a digital signature on 64-bit Vista
or 64-bit Widows 7 is (almost) impossible

System Call Hooking

13

The `NtXxx` -fuctions – the "Native API" –
are defined in `ntdll.dll` (a dynamic link library).

For `NtReadFile` you just find the following "**stub**":

```
mov  eax,0x102          ; intel syntax:
mov  edx,0x7ffe0300     ; op dest,src
call edx
ret  0x24        ; 9 arguments – 36 bytes
```

System Call Hooking

14

For `NtReadFile` you just find the following "**stub**":

```
mov  eax,0x102        ; system service number
mov  edx,0x7ffe0300   ; always the same address
call edx
ret  0x24             ; specific for the function

0x7ffe0300:
mov  edx,esp          ; arguments on the stack
sysenter              ; switch to kernel mode
ret                   ; Intel: sysenter – AMD: syscall
                      ; older processors: int 0x2E
```

System Service Number

15

```
mov  eax,0x102        ; system service number
```

the **system service number** in register EAX
indicates the function

In the kernel system service dispatcher (*KiSystemService*)
it is used as an index into the SSDT (System Service
Dispatch Table).

The system service numbers are not documented,
they can change between OS versions
(Russinovich, Solomon: even between service packs)

System Service Number

16

The system service numbers are not documented,
they can be obtained from the stubs in `ntdll.dll`:

```
( mov  eax,0x102:  0xB8 0x02 0x01 0x00 0x00 )

byte  *addr;       //typedef unsigned char byte;
uint  ssn, *pi;
char  Function[] = "NtReadFile";
addr = (byte*) GetProcAddress(
        GetModuleHandle("ntdll.dll"), Function);

pi = (uint*)(addr+1); ssn = *pi;
```

System Service Number

17

For `NtCreateSection` I obtained the following system service numbers:

Windows XP (NT 5.1) - 0x32
Windows Vista (NT 6.0) - 0x4B
Windows 7 (NT 6.1) - 0x54

We have to supply our driver with the appropriate number
- or the driver must detect the OS version
- (or the driver can find out the ssn from the corresponding `ZwXxx` kernel function (`ZwCreateSection`))

System Service Number

18

the driver must detect the OS version

```
RTL_OSVERSIONINFOW osvi;

osvi.dwOSVersionInfoSize =
                    sizeof(RTL_OSVERSIONINFOW);

RtlGetVersion(&osvi);
if( osvi.dwMajorVersion == 6 &&
    osvi.dwMinorVersion == 0 )  { Index = 0x4B; }
if( osvi.dwMajorVersion == 6 &&
    osvi.dwMinorVersion == 1 )  { Index = 0x54; }

// Service Pack:  build = osvi.dwBuildNumber;
// or use RTL_OSVERSIONINFOEXW .wServicePackMajor
```

System Service Number

19

Better Solution:
the driver can find out the ssn from the corresponding `ZwXxx` function.

The corresponding `ZwXxx` kernel functions start with the same 5 Bytes as their `NtXxx` counterpart stubs.

```
Index = * (ULONG *)((UCHAR *)ZwCreateSection +1);
```

Hoglund use a macro:

```
#define SYSCALL_INDEX(_Function) \
        *(PULONG)((PUCHAR)_Function+1)

Index = SYSCALL_INDEX(ZwCreateSection);
```

System Call Arguments

20

```
mov  edx,esp        ; arguments on the stack
sysenter            ; switch to kernel mode
```

Kernel code uses a different stack than user mode processes.

KiSystemService receives the address of the top of the callers stack in EDX –
it then copies the arguments to the kernel stack – so the system service functions can use (read) them.

Often arguments are pointers to buffers in user land, use them with care: `ProbeForRead()`, `ProbeForWrite()`

System Call Arguments

21

Often arguments are pointers to buffers in user land, use them with care: `ProbeForRead(), ProbeForWrite()`

```
__try {
     ProbeForRead(UserBuffer, Length, 1);
                     // 1 == TYPE_ALIGNMENT(char)
     RtlMoveMemory(KernelBuffer, UserBuffer, Length);

  } __except( EXCEPTION_EXECUTE_HANDLER ) {
       status = GetExceptionCode();
  }
```

System Call Dispatch

22

The kernel system service dispatcher (*KiSystemService*) uses the ssn (in Reg. EAX) as an **Index** into the **SSDT** (System Service Dispatch Table), which holds the addresses of the system functions.
A second table - **SSPT** - includes the number of bytes of the arguments for each function.
The addresses of both tables can be found in

```
struct SYS_SERVICE_TABLE { … };
```

the addresses of which is exported by the kernel in the variable `*KeServiceDescriptorTable`

System Call Dispatch

23

```
struct SYS_SERVICE_TABLE {
  void          **ServiceTable;     // SSDT
  unsigned long  *CounterTable;
  unsigned long   ServiceLimit;
  unsigned char  *ArgumentsTable;   // SSPT
};
```

(you will find different names of the structure and its members in literature)

```
extern struct SYS_SERVICE_TABLE
                     *KeServiceDescriptorTable;

int Args = KeServiceDescriptorTable->
                          ArgumentsTable[Index];
```

System Call Dispatch

24

To hook a system service function, we can change its address in the SSDT

```
ULONG  *RealCallee;
ULONG **SsdtEntryAddr;

SsdtEntryAddr =
    & KeServiceDescriptorTable->ServiceTable[Index];

RealCallee     = *SsdtEntryAddr;      // OK
*SsdtEntryAddr = (ULONG *) &Proxy;    // No !!!
// We usually have no write access -> blue screen
```

How to work around this, using MDLs (Memory Descriptor Lists), is described by Hoglund; Bassov proposed a shorter way:

System Call Dispatch

25

```
ULONG  Args, *RealCallee;

void StartHook(PWDM_DEVICE_EXTENSION pdx)
{
  void *SsdtEntryAddr, *PhAddr;
  SsdtEntryAddr =
     & KeServiceDescriptorTable->ServiceTable[Index];
  PhAddr = MmMapIoSpace(
            MmGetPhysicalAddress(SsdtEntryAddr),4,0);
  RealCallee = InterlockedExchangePointer(
                                    PhAddr, &Proxy);
  MmUnmapIoSpace(PhAddr,4);
}
```

System Call Dispatch

26

```
void ReleaseHook(PWDM_DEVICE_EXTENSION pdx)
{
  void *SsdtEntryAddr, *PhAddr;
  SsdtEntryAddr =
     & KeServiceDescriptorTable->ServiceTable[Index];
  PhAddr = MmMapIoSpace(
            MmGetPhysicalAddress(SsdtEntryAddr),4,0);
  InterlockedExchangePointer(PhAddr, RealCallee);
  MmUnmapIoSpace(PhAddr,4);
}
```

System Call Hooking

27

Stealth viruses and rootkits can conceal their presence by changing `NtReadFile` (if an infected file is to be read the altered system function presents the original file)

In the Rootkit-book by Hoglund and Butler is shown, how to hide processes by hooking `ZwQuerySystemInformation`

Bassov shows, how to control process creation on a system-wide basis, hooking `NtCreateSection`

System Call Hooking

28

To hook a system service function, we change its address in the SSDT. The SSDT entry then points to our "**proxy**" function.

Often the proxy must call the "real", original system function (pointer `RealCallee` in my code), to utilize its service.

If proxy uses the real function with different arguments, some **preprocessing** is needed to create the new arguments.
If proxy modifies the results of the real function, we need **postprocessing**.

System Call Hooking

29

To hide processes by hooking `ZwQuerySystemInformation` the results of the original `ZwQuerySystemInformation` function (called with the original arguments) are manipulated. No **preprocessing** is needed.

The original function returns a pointer to a linked list, containing all processes. Proxy then removes certain processes off this list in its **postprocessing** part.

Hoglund uses the original **function prototype** for his proxy `NewZwQuerySystemInformation` and for the function pointer `OldZwQuerySystemInformation` :

System Call Hooking

30

```
NTSTATUS  NewZwQuerySystemInformation(
                ULONG SystemInformationClass,   PVOID SystemInformation,
                ULONG SystemInformationLength, PULONG ReturnLength)
{
  NTSTATUS ntStatus;        // no  preprocessing
  ntStatus = OldZwQuerySystemInformation ( SystemInformationClass,
                SystemInformation,  SystemInformationLength,  ReturnLength );
  if( NT_SUCCESS(ntStatus)) {
    . . .      // postprocessing, using pointer SystemInformation
  }
  return ntStatus;
}
```

System Call Hooking

31

Bassov uses a more general method using an assembler **frame routine**:

```
_declspec(naked) Proxy()
{
  _asm{
    pushfd          // save flags
    pushad          // save 8 registers
    mov  ebx,esp
    add  ebx,40     // pointer to arguments of NtXxx
    push ebx        // 1 argument for check
    call check      // preprocessing C-routine
    cmp  eax,1      // result: allow or block ?
    jne  block
```

System Call Hooking

32

```
_declspec(naked) Proxy()
{
  _asm{
    . . .
    jne block

    popad           // proceed to the actual callee
    popfd
    jmp RealCallee  // no postprocessing

 block: popad       // return STATUS_ACCESS_DENIED
    mov ebx, dword ptr[esp+8]   // 1_st argument
    mov dword ptr[ebx],0        // Handle = NULL
    mov eax,0xC0000022L
    popfd
    ret 28  }       // 28 bytes of arguments
}
```

System Call Hooking

33

Proxy

- saves flags and 8 registers
- loads ebx with the address of the arguments on the stack
- calls a C-routine with this address as a parameter,
  this routine (in our case) decides whether or not
  to create the process
- yes: restore registers and flags, jump to original function
  this function returns with an NTSTATUS code
  to the caller of the original function (using ret 28)
- no: restore registers,
  set the handle, the orig. function returns, to NULL
  return with `STATUS_ACCESS_DENIED` using ret 28

System Call Hooking

34

```
_declspec(naked) Proxy()
{
  _asm{ . . . }
}
```

`_declspec(naked)` creates pure function code without a stack frame – intended for assembler code

Probably it is not necessary to save registers,
but who knows?

```
pushfd   // save flags
pushad   // save eax, ecx, edx, ebx,
         //  original esp, ebp, esi, edi : 32 bytes
```

`ret 28` is specific to the `NtXxx` - function

System Call Hooking

35

A similar **assembler frame routine** is a good solution when several `NtXxx` – functions are hooked with the same aim (e.g. logging the calls).

Then **one single routine** can be used for all hooked system functions, the routine somehow must know:

- the original callee
- the ssn Index for logging
- the number of bytes of the arguments for logging
  (and for ret n if necessary, i.e. if the block - branch is used)

A solution like in my previous talk (v. also Bassov 1) is possible, using the `call-indirect` instruction, pointing to a data structure that contains the relevant data

Hooking `NtCreateSection`

36

Bassov (2) shows, how to **control process creation** on a system-wide basis, hooking `NtCreateSection`.

He points out that `NtCreateProcess` is not always called when a process is created (e.g. `CreateProcess()` doesn't use it), so it's no use to hook this system function.

But there is absolutely no way to run any executable file without calling either `NtCreateFile` or `NtOpenFile`, and `NtCreateSection`.

In the latter function it is easier to take a decision whether is it a part of a process creation.

Hooking NtCreateSection

37

Control process creation on a system-wide basis, hooking `NtCreateSection`.

The hook can be utilized in any kind of **parental control** software, e.g. for logging or for preventing certain programs from running.

This can be expanded to accomplish a **secure environment** that prevents execution of any program that does not appear on a list of allowed software.

As a result, the PC is protected against add-on spyware, worms, and Trojans.

Hooking NtCreateSection

38

We hook `NtCreateSection`:

```
Index = * (ULONG*)((UCHAR*)ZwCreateSection +1);
```

In `check()` we first take a decision whether is it part of a process creation:

```
ULONG __stdcall check(PULONG arg)    // -> ret 4
{
  // check the flags
  if((arg[4]&0xf0)==0) return 1;
  if((arg[5]&0x01000000)==0) return 1;
  // no executable -> return 1 -> jmp RealCallee
```

Hooking NtCreateSection

39

```
 // check the flags
 if((arg[4]&0xf0)==0) return 1;
 if((arg[5]&0x01000000)==0) return 1;

arg[4] – IN ULONG  PageAttributes
arg[5] - IN ULONG  SectionAttributes

PAGE_EXECUTE              - 0x10
PAGE_EXECUTE_READ         - 0x20
PAGE_EXECUTE_READWRITE    - 0x40
PAGE_EXECUTE_WRITECOPY    - 0x80

SEC_IMAGE                 - 0x01000000
// hFile (arg[6]) is an executable image file
```

Hooking NtCreateSection

40

Next we find out the file name:

```
HANDLE          hand;
PFILE_OBJECT    file;
OBJECT_HANDLE_INFORMATION info;
//char          pathbuff[128];   // global - small stack
ANSI_STRING     path;
ULONG           len;

 hand =(HANDLE)arg[6];
 ObReferenceObjectByHandle(hand, 0, 0,
                           KernelMode, &file, &info);
 if(!file) return 1;
 RtlUnicodeStringToAnsiString(&path, &file->FileName, 1);

 strcpy(pathbuff, path.Buffer); len =path.Length;
 RtlFreeAnsiString(&path); ObDereferenceObject(file);
```

Hooking NtCreateSection

41

We find out the file name – is it an .exe file?

```
strcpy(pathbuff, path.Buffer); len =path.Length;
...

if(len<4 || _stricmp(&pathbuff[len-4],".exe")) return 1;
// _stricmp:  .exe == .EXE

DbgPrint(" Exe FileName: %s \n", pathbuff);
```

We now have the path of an exe-file,
we can decide whether we want it to run.
If not - `check()` can return 0, and then `proxy()` returns `STATUS_ACCESS_DENIED`

Hooking NtCreateSection

42

We now have the path of an exe-file,
if you need the drive letter too (e.g. D:), use:

```
UNICODE_STRING  DosName;
ANSI_STRING     drive;

IoVolumeDeviceToDosName(file->DeviceObject,
                        &DosName);
RtlUnicodeStringToAnsiString(&drive, &DosName, 1);
ExFreePool(DosName.Buffer);
```

Now you can concatenate the strings drive and path.

```
RtlFreeAnsiString(&drive);
ObDereferenceObject(file);
```

Hooking NtCreateSection

43

We have the path name of the exe-file,
we can decide whether we want it to run.

Run the program only if it appears on a “white” list of allowed software.
Or don’t run the program if it appears on a “black” list of prohibited software.

Bassov shows how to ask the user – he runs a user thread that periodically polls the driver for requests.

But Bassovs method, to return `STATUS_ACCESS_DENIED` on rejection is not elegant – run a dummy program instead.

Hooking NtCreateSection

44

I used the hook to prevent the famous chat-program ICQ from running in class-rooms:

```
if (len<8 || _stricmp(&pathbuff[len-8],"\\icq.exe"))
{ return 1; }

DbgPrint(" caught: %s \n", &pathbuff[len-8]);
a=0;          //   -> return a;
```

Bassovs method, to return `STATUS_ACCESS_DENIED` on rejection is not elegant – run a dummy program instead (that does nothing or just shows a simple message).

We close the underlying .exe-file and open dummy.exe

```
hand = (HANDLE)arg[6];
ZwClose(hand);
```

Hooking NtCreateSection

45

We close the underlying .exe-file and open dummy.exe´:

```
if(a == 0) {
  OBJECT_ATTRIBUTES objAttr;
  IO_STATUS_BLOCK   ioStatusBlock;

  RtlInitUnicodeString(&pathname,
        L"\\DosDevices\\D:\\SysHook\\icq2.exe");

  InitializeObjectAttributes(&objAttr, &pathname,
        OBJ_INHERIT |OBJ_CASE_INSENSITIVE, NULL,NULL);

  ZwClose(hand);

  ZwCreateFile(&hand,
            GENERIC_EXECUTE, &objAttr, &ioStatusBlock,
            NULL, FILE_ATTRIBUTE_NORMAL,
            FILE_SHARE_READ | FILE_SHARE_WRITE,
            FILE_OPEN, FILE_RANDOM_ACCESS, NULL, 0);

  memmove(&arg[6], &hand, 4);  a=1;
}
```

System Call Hooking

46

Thank you.

? Questions ?