适合读者：溢出爱好者
前置知识：汇编基础

亲密接触 MS06-055

文/图 failwest

MS06-055 曝光简史

2006 年 9 月 19 日，CVE 首先公布了这个 0day 漏洞，并编号为 CVE-2006-4868。网站还跟踪报道了其他一些世界著名的安全组织和安全网站对这个 0day 的技术讨论。

2006 年 9 月 19 日 11:14:35， 美国计算机应急响应组（US-CERT）同时公布了这个漏洞，并给出了一些简要的技术细节。

2006 年 9 月 19 日，NVD 报道了该漏洞。

2006 年 9 月 20，该 0day 的溢出攻击测试代码被“xsec”的“nop”发布。该攻击代码迅速在网络上传播开来。

2006 年 9 月 20 日，中国的安全公司中联绿盟（NSFOCUS）在国内首先报道了该漏洞。

2006 年 9 月 22 日，中国计算机应急响应组（CN-CERT）报道了该漏洞。

2006 年 9 月 26 日，微软发布了针对该漏洞的安全补丁 MS06-055（KB925486）。

2006 年 9 月 29 日，中国的安全公司启明星辰（VENUS）报道了该漏洞。

这份时间表显示出了一次网络安全应急响应的全过程。当漏洞刚开始在网上被披露的时候，听说微软正在对补丁进行测试，本来计划 10 月 10 日正式发布，后来迫于溢出代码已经在网上传播开来的压力，为了不至于造成大规模损失，被迫提前两周发布了安全补丁。

至此，这个安全事件基本过去。我们也可以从中发现国内的安全组织和应急响应机构对 0day 的敏感程度和反映速度与国外权威的机构 CERT 之间还是有一定差距的（绿盟除外）。中国教育网计算机应急响应组 CCERT（http://www.ccert.edu.cn/）甚至没有对这个 0day 做出响应。

漏洞分析及利用

我们可以在微软的安全技术网页或者 US-CERT 上得到一些关于这个漏洞的简要描述：微软的 IE5.0 以上的版本会支持一种向量标记语言（VML）来绘制图形。IE 在解析畸形的 VML 的时候会产生堆栈溢出的错误。如果利用者精心构造一个含有这样畸形的 VML 语句的网页，并骗取用户点击这样的网页，就可以利用 IE 在用户的机器上执行任意代码。

参考这些简要的描述和“nop”公布的攻击代码，我们可以实际动手调试一下这个漏洞。

引起栈溢出的是 IE 的核心组件 vgx.dll。这个文件在 **C:\Program Files\Common Files\Microsoft Shared\VGX** 和 **C:\WINDOWS\system32\dllcache** 两个目录下。如果你的机器上已经打过 MS060-055 的补丁的话，你可以到 **C:\Windows\$NtUninstallKB925486$**找到原来的有漏洞的文件。

我们首先用 IDA 把它反汇编，获得一个比较全局的把握。发生栈溢出的函数是

_IE5_SHADETYPE_TEXT::Text(unsigned short const *, int)，在这个函数初始化的地方有一条指令：

sub esp ,214h

它开辟了一个 0x214 的栈空间。在 Text()函数执行的过程中，会两次调用另一个函数 Ptok()。Ptok()会把 HTML 页面中 VML 描述里一个域中的字符串按照 Unicode 的形式复制到这个栈空间中。

下面我们用一个实际的例子来说明这个溢出问题。我的调试机器是 Windows XP SP2。首先创建一个 HTML 文件，其内容如下：

```
<html xmlns:v="urn:schemas-microsoft-com:vml">
<head>
<title>failwest</title>
<style>
<!--v\:* { behavior: url(#default#VML); }-->
</style>
</head>
<body>
<v:rect style="width:444pt;height:444pt" fillcolor="black">
<v:fill method="QQQQ"/>
</v:rect>
</body>
</html>
```

这是一个含有 VML 描述的页面。保存后用 IE 打开，应该可以看到一个很大的黑色的正方形。这就是 VML 语言相对于图片格式的好处——只要几个字节描述一下形状、颜色、坐标信息等属性，就能绘制出精确的图形——想想这么一个正方形用 jpeg 或者 bmp 表示要多少字节吧！讨论 VML 技术不是本文的重点，我们继续讨论这里的溢出问题。用 OllyDBG 连接到现在的 IE 进程上，在 Text()函数开始的地方下断点。

我们可以用 IDA 辅助查找到 Text()函数的入口地址，我这里是 0x6FF3ED46。直接用组合键 Ctr+G 跳到这个位置，按 F2 下断点（注意，如果你是用 OllyDBG 直接打开 IE，而不是采用 attach 的话，vgx.dll 可能还没有被 load，那么 IDA 中找到的 RVA 可能是无效区域）。

刷新页面，这时候 OllyDBG 会在函数入口处断下进程，用 F8 单步执行，一直到**“6FF3ED92 call vgx.6FF3ECE6”。**这时候注意一下栈中的状态，就在离栈顶几个字节的地方，有 4 个 Unicode 形式的字符“Q”被复制到栈空间中了！原来这个 Call 就是在调用 Ptok()，VML 描述中<v:fill method="QQQQ"/>里双引号之间的 ASCII 将被转换为 Unicode 复制入 Text()函数的局部变量。

在我的实验环境中，栈中数据是这样分布的：

0012BE70：Unicode 字符串起址
0012C070：存放 Security Cookie
0012C074：EBP
0012C078：return address
0012C084：返回后的 ESP

很明显，实际为 Unicode 字符串分配的可使用空间是 0x200，也就是 10 进制的 512 个字节，换成 Unicode 就是 256 个字符，减去字符串结尾的两个字节的 0，栈中能够接受的最多的字符就是 255 个。

这是一个比较典型的栈溢出。如果我们在页面中 VML 对应的域中包含超过 255 个字母的话，经过 Ptok()函数的复制，栈中的函数返回信息就会被覆盖。如果页面中包含有 Shellcode 的话，那么点击一个含有这样页面的链接时，你的 IE 可能就会悄无声息地执行了攻击者想要的代码，从恶作剧的弹出对话框，到开后门、下木马、格式化硬盘，在技术上都没有什么质的区别。

这里大家要注意一点，如果你要制作自己的 Shellcode 来利用这个漏洞，还需要解决一个问题。Ptok()在把字符从 html 中复制到栈空间的时候会进行 Unicode 转换，这将破坏我们的 Shellcode！然而在解析 html 的时候，是有一些转义字符的。比如“邐”将会按照 10 进制数字解释后面的 37008，换成 16 进制就是 0x9090，所以 html 里这样一个字符串“邐”将以 0x9090 的形式出现在栈空间。

在“nop”的攻击代码中，给出了一个用来处理 Shellcode 的函数，这里一并给出。大家可以把自己的 Shellcode 用这个函数处理后放在页面里去实验。函数代码如下：

```
//////////////////////////////////////////////////////////////////////
void convert2ncr(unsigned char * buf, int size)
{
	int i=0;
	unsigned int ncr = 0;
	for(i=0; i<size; i+=2)//read a word
	{
		ncr = (buf[i+1] << 8) + buf[i];
		fprintf(fp, "&#%d;", ncr);
	}
}
//////////////////////////////////////////////////////////////////////
```

认识 Windows XP SP2 的安全编译选项

经过上面的分析，即使是一个刚学溢出的新手也能在 Windows 2000 上成功实现攻击了；但是在 Windows XP SP2 系统中，vgx.dll 在编译时使用了/GS（Buffer Security Check）选项，因此要成功利用是非常困难的。下面我们就来看看这个编译选项在机器码里做了些什么，到底是怎样保护程序不被溢出攻击的。

用 IDA 反汇编 vgx.dll，其 Text()函数代码如下：

_IE5_SHADETYPE_TEXT::Text(unsigned short const *, int)

```
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
.text:6FF3ED46 ; _IE5_SHADETYPE_TEXT::Text(unsigned short const *, int)
.text:6FF3ED46                    mov     edi, edi
.text:6FF3ED48                    push    ebp
.text:6FF3ED49                    mov     ebp, esp
```

```
.text:6FF3ED4B                  sub     esp, 214h
.text:6FF3ED51                  mov     eax, ___security_cookie
.text:6FF3ED56                  and     dword ptr [ecx], 0
.text:6FF3ED59                  mov     [ebp+4], eax
.text:6FF3ED5C                  mov     eax, [ebp+arg_0]
……..
.text:6FF3ED92                  call    _IE5_SHADETYPE_TEXT::TOKENS::Ptok
.text:6FF3ED97                  test    eax, eax
…….
.text:6FF3EDB1                  lea     ecx, [ebp+var_210]
.text:6FF3EDB7                  call    _IE5_SHADETYPE_TEXT::TOKENS::Ptok
……..
.text:6FF3EDDF                  mov     ecx, [ebp+4]
.text:6FF3EDE2                  call    __security_check_cookie(x)
.text:6FF3EDE7                  leave
.text:6FF3EDE8                  retn    8
$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$
```

大家可能已经注意到了，在函数开始的地方有这样一条指令：

.text:6FF3ED51 mov eax, ___security_cookie

IDA 直接把内存地址 0x6FF44160 翻译成了**“___security_cookie”**。这个“cookie”的值随后被放在了 EBP 头顶上的位置[EBP+4]：

.text:6FF3ED59 mov [ebp+4], eax

在函数返回前的最后一个调用：

.text:6FF3EDE2 call __security_check_cookie(x)

就是用来检查 EBP 前一个 DWORD 的值是否被修改过的。如果这个值和开始时存入的 cookie 不一样（被覆盖），则进入异常处理，正常的函数返回就得不到执行。而要覆盖返回地址，就必须先覆盖这个存放 cookie 的地方！经过动态调试，这个 cookie 的地址在 vgx 的数据段，每次启动 IE 进程的时候都不一样，我们几乎不可能在 Shellcode 中预测到这个 cookie 的值。

通过这样一个保护机制，我们在 Windows XP SP2 上不论怎么精确的覆盖 ESP 这些地方都没有用，因为不执行 ret，我们的 Shellcode 就得不到执行。