前置知识：无
关键词：破解、ESP定律、SHE链

Anti ESP定律

文/woosheep

自从某年某月某位大侠发现了ESP定律，脱壳者笑了，写壳者哭了。从兼容的角度看，执行流从壳进入OEP后，没有什么追求可以比保持寄存器的原始环境更高了，而最简捷的实现方法，就是使用pushad和popad。是的，一切本该如此，先是在入口处放置pushad，然后进行壳的处理，最后popad完毕就跳转到OEP。然而，ESP定律打破了这套搭配的默契，高手是可以另辟蹊径的，可写壳的门槛就大大提高了。

得有个办法，不能让ESP定律再这么肆虐下去了。首先分析一下ESP定律的特点，经典的ESP定律是在入口的pushad下一条指令，对ESP指向的内存地址下硬件断点，运行之后，程序就会断在即将跳转入OEP的popad之后。为了模仿这个过程，我写了一段实验代码：

```
_declspec(naked)main()
{
_asm
{
pushad
call doSomething
popad
jmp oep
}
}
```

这是一个main函数的函数体，其中oep和doSomething均是空函数的函数名，反汇编后其函数体只有一条retn指令。相信熟悉C的朋友都会知道_asm是内联汇编的标记，但这个_declspec(naked)则有点面生。MSDN对其有详尽的解释，简单来说，就是要求编译器不要添加额外的代码。以上面代码为例，如果不加_declspec(naked)标示，编译器则会自动加一段诸如"push ebp"和"mov ebp,esp"这样的指令，起到保证堆栈平衡等作用，但由于我们是模拟壳的保护寄存器现场动作，以jmp oep为结束，并未真正执行到main函数本身的retn，编译器的自作主张反而会影响堆栈平衡，而最直接的解决办法就是设置_declspec(naked)标示。除此之外，如果是纯内联汇编编写的函数，我个人习惯也加上_declspec(naked)标示，这样生成的代码就不会发生变化，彻底杜绝编译器的节外生枝。

现在，我们假定doSomething函数为模拟壳的一些操作，而进入oep函数内则意为壳完成操作开始执行原程序，上面代码反汇编结果如下：

```
00401060   60              pushad
00401061   E8 CAFFFFFF    call    00401000
00401066   61              popad
00401067   E9 94FFFFFF    jmp     00401000
```

我认为这段代码，从结构上来说应该能大致模拟壳的保护寄存器现场动作了。按ESP定律操作，我们会顺利断在jmp 00401000这条指令

上。好，这是正常情况。接下来我们改写一下doSomething函数

```
_declspec(naked)doSomething()
{
_asm
{
lea  eax, clearDR
push eax
push fs:[0]
mov  fs:[0],esp
_emit  0xCC
pop dword ptr fs:[0]
lea esp,[esp+4]
retn
   }
}
```

具体作用先不解释，其中clearDR是一个函数名，代码为：

```
_declspec(naked)clearDR()
{
_asm
{
mov                eax, [esp+0xC]
mov      [eax+0x4], 0
mov      [eax+0x8], 0
mov      [eax+0xC], 0
mov      [eax+0x10], 0
inc        [eax+0xB8]
xor        eax,eax
retn
}
}
```

下面是见证奇迹的时刻。依旧是按ESP定律操作，F9运行，程序没有任何迟疑就直接结束了。由于oep函数只有一个retn，因此执行到oep的标志就是程序结束。当然，这不重要，重要的是，ESP定律没断下来，失效了。

大名鼎鼎的ESP定律就这样被这段不知所云的代码无声无息地干掉了，悬疑的气氛里是不是又平添了几分诡异。如果这是魔术，想必大家都会觉得破解这魔术的关键就在于doSomething函数新添的内容。

明眼的朋友肯定已经发现，这是一段设置SEH链的经典代码。SEH链说白了就是一系列异常处理函数，按处理异常的先后次序由一个双DW的结构链接起来，组成一个链表，这个链表就是SEH链。为SEH链添加新节点说起来复杂，但其实编码很简单：首先获取要添加入SEH链的异常处理函数的内存地址，然后填写一个由两个DW组成的结构。第一个DW就是刚才获取的异常处理函数的地址，另一个DW则是原来SEH链的最后一个双DW结构的地址，最后一个双DW结构的地址也好找，因为系统规定fs:[0]必须像链尾指针一样指向SEH链的最后一个双DW结构，因此直接取fs:[0]的值即可。我这里添加节点采用比较时髦的堆栈方式实现，当然，只要保证新增加的结构正确，也可以用任何想得到的方式，就不展开说了。不过要记得，添加完新节点，要让fs:[0]指向新添加的双DW结构，整个过程跟链表操作大同小异。至于_emit 0xCC，其实就是int3。C的内联汇编没有提供DB之类的数据类型，只能用_emit直接写机器码，为的就是引发一个中断，调用SEH链进行处理。程序处理完中断，程序接下来执行的就是恢复原来SEH链的操作了，也是标准的链表操作：首先调整起指针功能的fs:[0]，然后"释放内存"，把先前压了两个DW，现在弹出两个DW，堆栈平衡，打完收工。

看来重头戏在clearDR，而clearDR的代码其实也异常的简单。关键是第一句"mov eax,[esp+0xC]"，这句是用来获取一个名为CONTEXT结构的地址。CONTEXT结构是一个相当复杂的结构，号称几乎是WIN32API里唯一一个与处理器有关的结构，总之很复杂，所以这里只需要记住这个结构的偏移0x4的位置是Dr0，即第一个硬件调试寄存器的值，其余的Dr1等三个硬件调试寄存器的地址依次递增即可。所谓硬件调试器的值其实是一个地址，简单来说，如果对0x400000这个地址下硬件断点，那么硬件调试器的值即为0x400000。现在把这个值赋0，硬件断点就失效了，ESP定律也就over了。

说到这里，前面浓墨重彩铺垫的AntiESP定律，其实就草草收场了。回顾一下，很容易发现，AntiESP定律，获取硬件寄存器的值是关

转75页

如图7所示，三台服务器上的LDAP分支是在同一棵树上通过类似于"引用"的特殊属性连接在一起的，三个城市都有自己的管理员来维护各自的分支。三台服务器连接在一起有很多好处，在需要搜索或查询公司所有员工信息的时候就会十分方便。如果某个城市的员工较多或负载过重还可以在这个城市增加LDAP服务器，增加的服务器就如同本地服务器的镜像，可以起到负载均衡的作用。需要注意的是，在这个设计方案里每个LDAP服务器都是可以读写的。

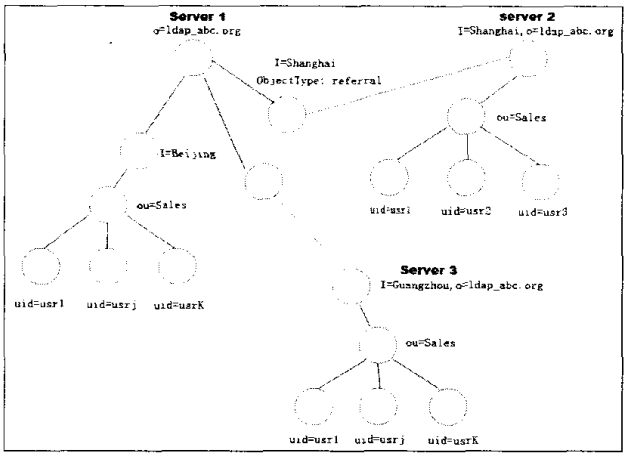

图7 树结构的设计

另一种方案是在北京存放一台主LDAP服务器，同时在北京、广州和上海三个城市分别放置一台辅LDAP服务器。主LDAP服务器只负责写入数据，辅LDAP服务器只读，任何数据库的修改都要先写入主LDAP服务器之后再同步到三个辅LDAP 服务器，这样的拓扑结构也比较适合认证服务器的需要，因为写操作相对于读操作更少，这种设计的优势在于能根据员工的多少或各分公司的负载情况灵活增加或减少服务器，任何一台辅LDAP服务器的瘫痪都不会影响到整个认证系统。在国外的一些案例中，辅LDAP服务器会多达几十甚至上百台。但这种设计也有缺陷， 如果主LDAP出现问题，那么所有的写操作就会受到影响， 目前OpenLDAP还不支持多个主LDAP 服务器的情况，因为设置多个主LDAP服务器有可能会造成整个数据的不一致， 只有少数商用的LDAP 支持多主LDAP的功能。

统一身份认证的未来

在国外，LDAP应用于身份认证已经十分成熟，最近几年国内才开始逐渐流行起来。前面提到的认证方式还存在一些不足，例如用户在使用FTP服务后，如果再使用Samba服务就需要再次输入用户名和密码，目前微软的Active Directory通过管理域用户已经完美的实现了单点登录，Linux可以通过OpenLDAP和Samba实现大部分Active Directory能够实现的功能。相信在不久的将来，用LDAP做身份认证的技术还会渗透到更多领域，包括网络计算机、门禁系统，甚至智能IC卡的应用。

接125页

键，而要获取硬件寄存器的值，可以通过获取CONTEXT结构。而要获取CONTEXT结构，就要引发异常，当程序因异常进入SEH链进行处理时，[esp+0xC]和[ebp+0x10]的值即为CONTEXT结构的起始地址。需要说明的是，我的代码是在Windows XP环境下试验成功的，并不保证[esp+0xC]和[ebp+0x10]能在Vista或Win7下也能得到同样的东西。

AntiESP定律的原理解释完毕了。不过编写SEH链处理函数时需要注意两点，第一，异常处理函数返回时必须保证eax为0，因为eax的值为0系统才会认为异常已经处理，可以返回抛出异常的原代码处执行; 第二，CONTEXT结构中，偏移0xB8处的值为返回执行的地址，初始值为抛出异常的地址，必须正确设置此值，否则会循环触发异常。以本文代码为例，本例触发的是int3，想要处理完int3后继续执行下一条指令，就需要用inc [eax+0xB8]正确设置异常返回地址。因为int3的指令长度为1，所以下一条指令的地址为异常处地址加1。

最后说一下拓展。我这里只是比较厚道地把硬件寄存器的值清0，使其失效就点到为止。实际上对上述代码稍加改动，使其实现判断硬件寄存器的值的值是否为0，就变成一套检测程序是否被非法调试的经典方法。只要检测出程序被调试，剩下的怎么鱼肉调试者就全凭个人想象了。