1. 栈溢出漏洞

此次产生漏洞的系统为Windows 2000 sp4英文版，可以在msdn I tell you中下载镜像文件进行验证。由于Windows2000并未开启GS守护，因此在进行漏洞利用的时候可以通过覆盖函数返回地址跳转shellcode执行。

此漏洞位于neipai32.dll的NetpwPathCanonicalize中。此函数有6个参数：

{

Uint16 path[]；

Uint8 can\_path[]；

Uint32 maxbuf;

Uint16 prefix[];

Uint32\* pathtype;

Uint32 pathflags;

}

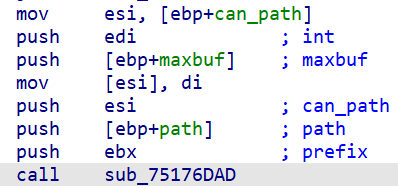
此函数进行Unicode字符串处理，将prefix串加上\与path串相连并复制输出到can\_path中，输出串的大小为maxbuf。

表达式为：prefix[]+’\’+path[] -> can\_path[];（这只是大体功能，对此函数我的问题是pathtype的用处、pathflags的用处、函数中的栈区最大和maxbuf是否相关，接下来的静态分析可能能解决也可能解决不了）。

2． 漏洞函数静态分析

在进行静态分析之前需要定位到NetpwPathCanonicalize函数，通过使用PE Load

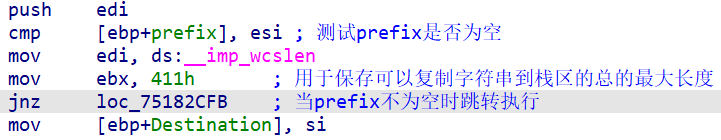
进行漏出点的定位，也可以通过ida pro自带的导出函数定位到NetpwPathcanonicalize（此函数前边有好多验证的函数，由于嵌套函数过多，并未分析出不满足的条件）直接定位到漏出点如图1所示:



**图1**

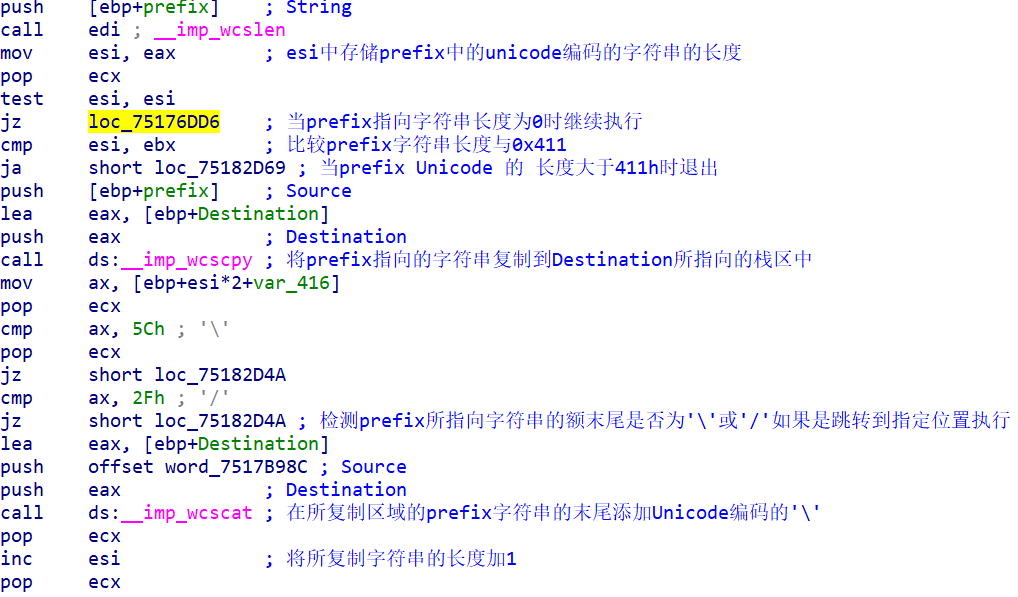
传入的参数与NetpwPathCanonicalize的参数基本相同这里着重关注prefix、path、can\_path三个参数，溢出点就出现在三个参数指向字符串的复制的过程中。

进入此函数后：



**图2**

图2进行prefix指向区域的字符串的验证，如果所指向区域字符串长度为0，或者并未指向字符串的继续向下执行不执行prefix到Destination的复制，这里的Destination是在栈区开辟的长度为414h字节长度的用来保存字符串的中间体。



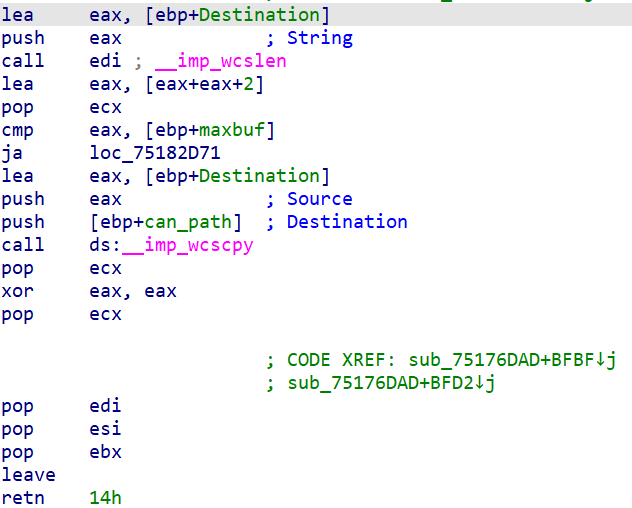
**图3**

图3是从prefix指向的字符串到Destination区域的复制，问题就发生在此处，由于开辟的414h长度的栈区的单位是字节，但用来验证字符串长度的函数wcslen是计算Unicode编码的字符串的长度，在之后长度检验是用0x411进行的如果长度达到0x411时实际长度是0x822字节这样一来就会超出栈区空间的大小，进而覆盖掉返回地址，进而产生溢出。



**图4**

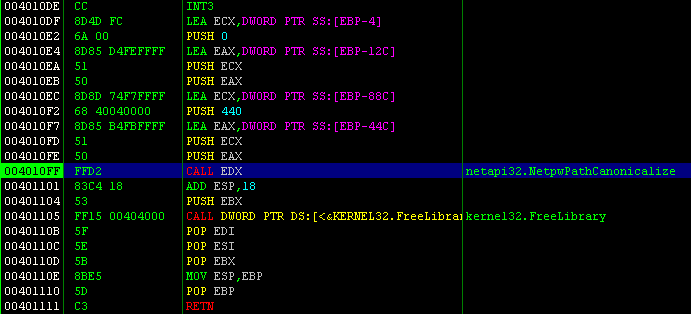
溢出产生在图4中，esi中保存的是prefix的Unicode长度+1,之后又加上了path指向字符串Unicode的大小，在和0x411比较的时候由于按照Unicode计算长度则合理的配置0x420字节的恶意数据就会通过数据长度检验，进而溢出到返回地址执行到图5的ret 14h的时候就会执行覆盖返回地址指向区域的代码，至此静态分析结束



**图5**

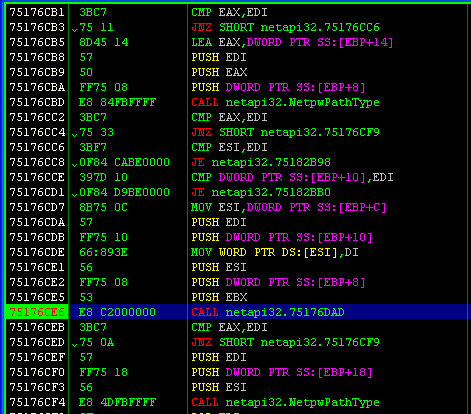
3． 用od进行在本地进行动态调试

操作系统：windows xp sp3，编译器：VC6.0 DLL文件：Windows2000 sp4 英文版 system32/netapi32.dll，测试代码windows/local/exploit.c

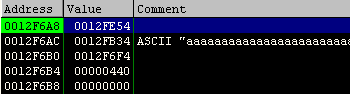


**图3.1 NetpwPathCanonicalize进入点**

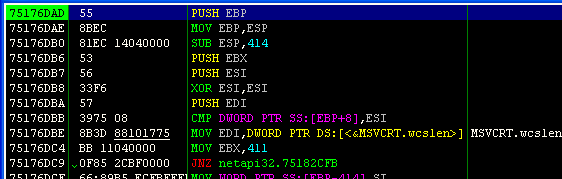
进入待分析函数中在如图3.2所示的高亮处下断点，按F9执行至此，查看栈区中内容进行验证正是传入的参数。进入此函数后开辟了0x414字节的栈区，之后将prefix指向的字符串复制到栈区，这一次复制并未影响到返回地址，返回地址位于ebp+4处。继续执行path复制到栈区后再次观察返回地址的内容此时返回地址出现奇怪的值，观察寄存器ecx指向所复制恶意数据在栈区中起始位置。由此返回地址指向的代码只要是jmp/call ecx便可以执行恶意代码。Jmp/call ecx可以在插件中的ollyfindaddr中确认如图3.10.如果没有此插件可以从网上下载。



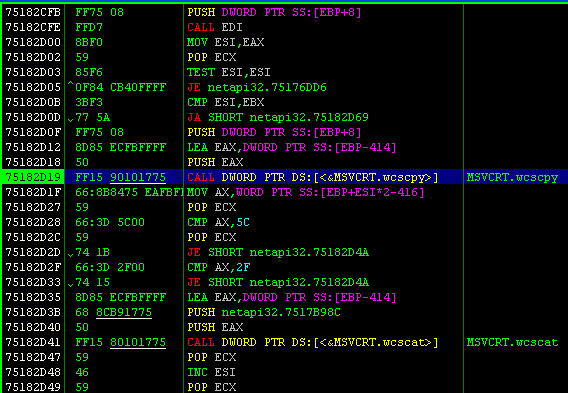
**图3.2 漏洞产生函数**



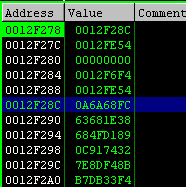
**图3.3 进入漏洞产生函数前栈区的内容**



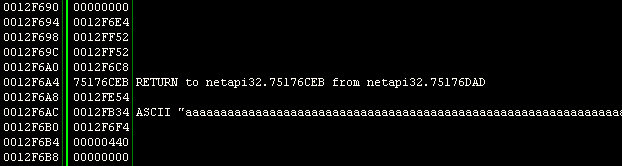
**图3.4 开辟0x414字节的栈区空间**



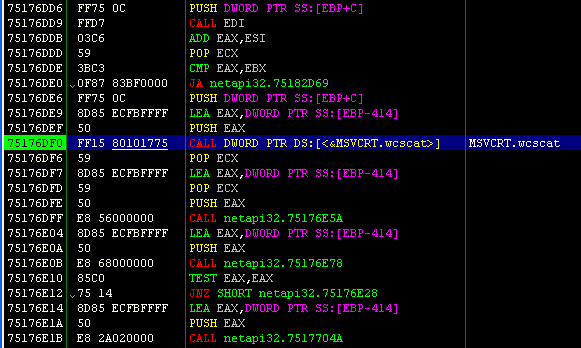
**图3.5 复制prefix指向的内容到栈区中**



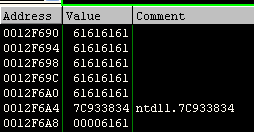
**图3.6 栈区中内容**



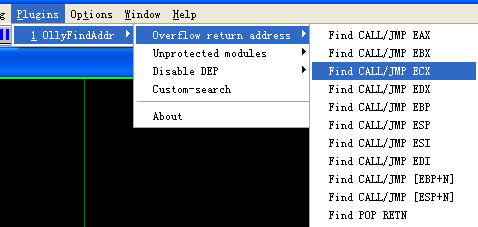
**图3.7 第一次复制后的返回地址的数据**



**图3.8 将path指向的字符串复制到栈区**



**图3.9 path中的内容复制到栈区后的返回地址**



**图3.10 插件jmp/call ecx**

4. Windows上远程过程执行的分析与实现

如何远程利用此漏洞呢？在本地机上可以通过加载dll文件然后执行其中导出的函数，但如果换到远程主机上又如何调用产生漏洞的函数呢？这里需要借用RPC：远程过程调用，一种进程间通信机制。

这里通过一个hello world实例演示rpc

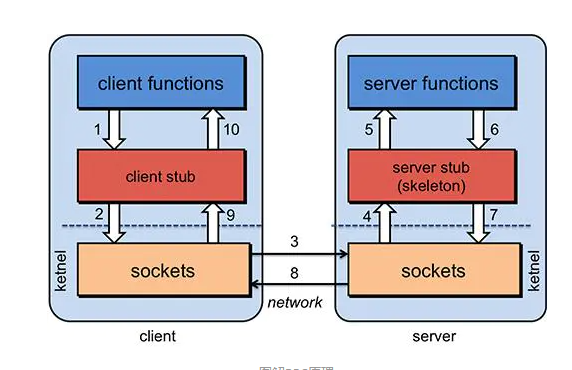
操作系统：Windows xp （服务机和客户机均为Windows xp）编译器 vs2008

首先简单的介绍一下rpc框架

Rpc框架包含四个内容：

1. 客户端（client）：服务的调用方。
2. 客户端存根（client stub）：存放服务端的地址信息，将客户端的数据打包成网络数据通过网络发送给服务方。
3. 服务端存根（server stub）：接受客户端发送过来的消息，将消息解包，并调用本地的方法。
4. 服务端（server）：真正的服务提供者。

RPC调用过程如图4.1所示

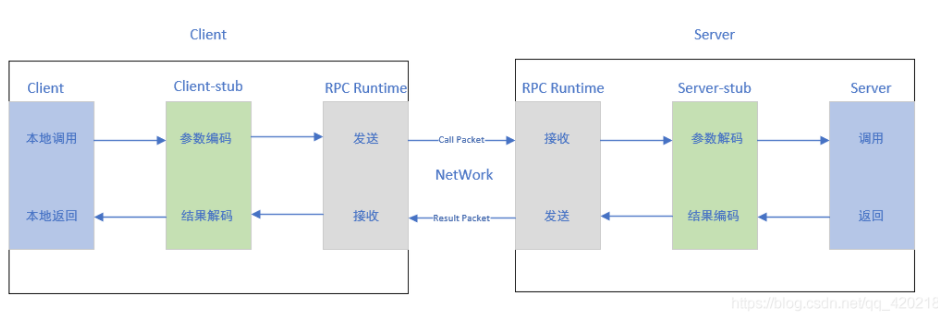


**图4.1 RPC调用原理**

调用过程为：

1. Client以本地调用方式（接口）调用服务。
2. Client stub 接受调用后，负责将方法，参数封装成能够进行网络传输的消息
3. 客户端通过socket传输网络消息
4. Server stub将收到的消息解码，根据解码结果调用本地服务。
5. Server在本地执行过程并将结果发送给server stub
6. Server将结果封装为网络消息，发送给客户端
7. Client stub接受网络消息并进行解码，解码后的结果发送给客户端。
8. Client接受返回的结果。

图4.2为调用过程

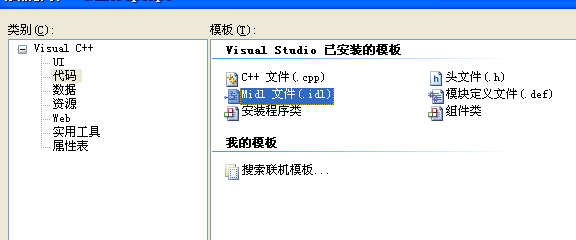


**图4.2 RPC调用过程**

接下来通过一个实例来介绍Windows上RPC调用过程

RPC的接口标准使用了IDL语言标准描述，通过IDL文件定义客户端与服务器之间的通信接口（这一方面的内容我也不是很了解，可以自行在网络上搜索）Windows上的远程调用接口文件可以参照[www.samba.org/ftp/unpacked/samba\_current/librpc/idl](http://www.samba.org/ftp/unpacked/samba_current/librpc/idl)下的idl文件进行编写。

在vs中调加idl文件如图4.3所示：



**图4.3 调加idl文件**

Hello.Idl内容为

[

uuid("62560D0C-C276-45cd-B60C-2922BE90031B"),

version(1.0)

]

interface HelloWorld

{

void Hello([in,string]const char \*psz);

void ShutDown(void);

}

（这里的接口文件我不太明白，客户端可能使用uuid来标识客户端与服务端之间的绑定，这里只是我的猜测，有错误希望指正，但是在客户端的绑定的时候用到了一个叫做endpoint的东西，我在测试的时候如果绑定与服务端不一致将导致连接不成功）

接口中的函数将在服务端实现然后在客户端进行调用。（我在测试的时候，即便客户端与服务端的函数名不一致但只要处于同一位置就会按此位置的函数执行，就比如假如客户端将UselessProc与Shutdown对调，那客户端调用Shutdown的时候在服务端实际执行的是UselessProc，所以我猜测这里用的向量标记函数而不是函数名标记）接口后的helloworld要与acf中接口后的接口名一致否则编译不通过。

调加acf文件，VS中并没有现成acf文件结构随意添加一个.c文件后重命名即可

Hello.acf中的内容为

[

implicit\_handle(handle\_t HelloWorldBinding)

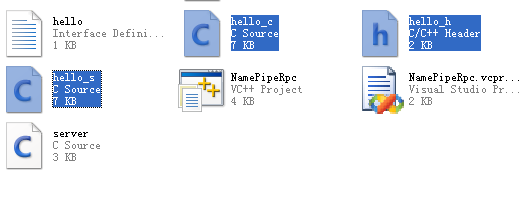
]

interface HelloWorld

{

}

然后进行编译将生成如图4.4所示的三个文件



**图4.4 生成的文件（标蓝）**

其中的hello\_s.c供服务端使用，hello\_c.c供客户端使用，hello\_h.h供客户端与服务端共同使用。

服务端实现：将生成的hello\_s.c和hello\_h.h添加到新建的项目中然后添加server.c,此文件的实现细节在windows/rpc/server.c中进行了描述，具体看源码。

客户端实现：同服务端类似，将将生成的hello\_c.c和hello\_h.h添加到新建的项目中然后添加client.c,此文件的实现细节保存在Windows\rpc\client.c中。

服务端编译运行后，客户端编译运行，这样在客户端调用的Hello就会在服务端的控制台打印信息（）。

Windows上的rpc是通过smb协议通信的后续会抓通信的包进行smb协议分析，进而摆脱Windows通用rpc标准直接借用socket实现远程通信。

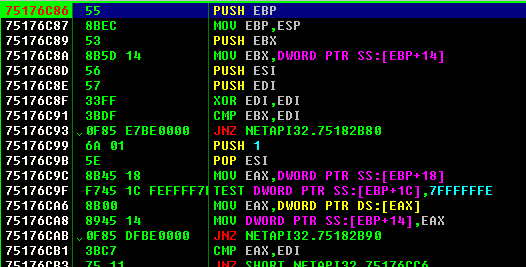
5. 运用攻击机对靶机进行远程代码注入

攻击机系统 Windows xp sp3，编译器：vs2008，靶机系统：Windows 2000 sp4英文版，攻击代码保存在Windows/rpcexploit/bing文件夹中（包含接口文件exploit.idl，配置文件:exploit.acf，攻击代码exploit.c）

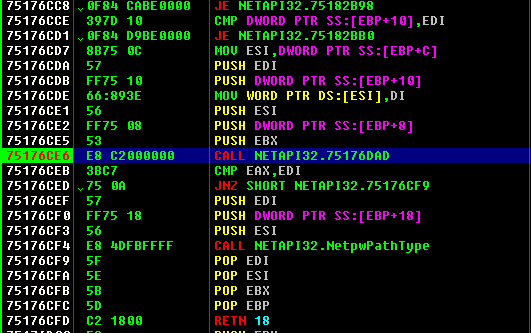
攻击端执行代码后在客户端会ding的一声但并没有弹出弹窗，由于入侵的服务并没有开启gui，所以不会弹出窗口。

动态分析过程。在Windows 2000 sp4中打开ollydbg并attach到services上，并用loadpe确定的NetpwPathcanonicalize函数起始点下断点，在攻击端执行代码后就会停在断点处。

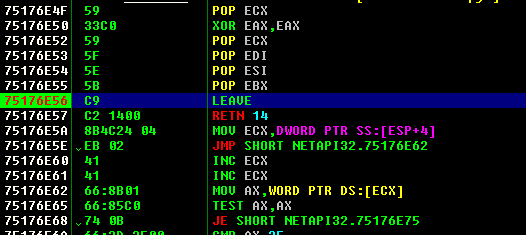
此次动态分析与上边本地静态分析基本一致，这里就截下主要的图片，具体内容不做赘述。



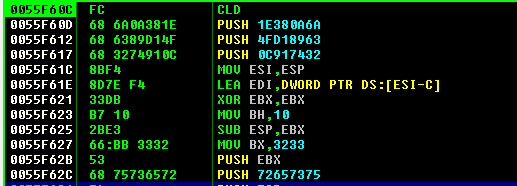
**图5.1 入侵函数进入点**



**图5.2 产生漏洞的函数**



**图5.3 跳转shellcode执行**



**图5.4 shellcode**

这次的入侵并没有造成控制靶机的效果为了能供控制靶机，这里采用bingshell的shellcode通过telnet 靶机IP 6666进行正向连接（之所以不使用反向连接是因为我不会写。。。。。。。）具体实现代码在windows/rpc/exploit/bindshell文件夹中。（但是这个正向连接依旧存在问题，可能是shellcode执行后破坏了栈区空间，或者是栈区不平衡函数无法正常退出，解决这个问题放在后续的shellcode优化中）。

（后续更新）

6. 分析smb协议

7. 优化shellcode

8. 不借用接口直接用smb协议进行攻击