第 3 章 网络病毒的典型入侵机制

这一章阐述最典型的网络入侵机制，即溢出攻击。我们不仅详细分析经典的栈溢出攻击，也详细讨论其复杂的变体，包括堆溢出攻击和单字节溢出攻击，这些在当前的网络病毒中都有广泛应用。一个惊人但非常不幸的事实是，直到第一代溢出攻击技术被揭示出来之后整整二十年的今天，这一攻击仍然是最有效的入侵手段之一，大量商业软件因为具有溢出漏洞而被迫不断地修修补补，其中包括 Microsoft 这样的一流软件开发商，用户则为此蒙受痛苦和损失。

3.1 栈溢出攻击

栈溢出攻击是第一代溢出攻击技术，也称为 *Morris* 代码注入[1]。先看下面一段 C 语言程序，这一函数将来自外部的一串字节复制到其局部数组 buffer，你能看出什么问题吗？

```
#define MAX_BUFFER_SIZE   256

void routine(char* p) /*p[]是来自外部的字符串*/
{
    int x,y,u,v;
    char buffer[MAX_BUFFER_SIZE];
    /*将输入字符串填入 buffer:*/
    strcpy(buffer, p);
    …… ……
    return;
}
```

strcpy(buffer, p)将数组 p[]的内容逐字节复制到数组 buffer[]，但没有检查 p[]的实际长度是否超过 buffer[]的容量，即 MAX_BUFFER_SIZE！设想运行期间 p[]承载的是来自外部的网络消息，而且该消息(有意或无意)确实超过了上界 MAX_BUFFER_SIZE，这将导致数组 buffer 的溢出。这样一来有什么后果呢？

当进程在运行期间调用 routine 之后，相应的栈上的数据布局情况如图 3-1(a)所示。注

[1] Morris 网络蠕虫是网络病毒的开山始祖，爆发于 1988 年，所用的就是本节描述的方法，入侵对象是 VAX 平台上的 finger 服务器。Morris 曾为此被美国政府起诉判刑，当时他是一名二十一岁的大学生，现在是美国麻省理工学院计算机系教授。

意为表达的更接近实际，这里假设运行平台是 Intel CPU，因此这里运用 Intel CPU 的寄存器予以说明，但对其他处理器道理是完全相同的。

图 3-1(a)的栈帧中除 C 函数的局部变量和参数之外，还包括两项：一项是所谓“上一栈帧的地址”，这是一个指针，指向函数 routine 当前的调用者(也是某个函数，不妨称为 routine 的父函数[2])的栈帧在栈空间中的位置(该栈帧在栈空间中必与 routine 的栈帧紧邻，如图 3-1(b)，不妨称其为父栈帧)，当函数 routine 返回之前，该项的值将传递给 EBP 寄存器，使操作系统能够找到 routine 的父栈帧；另一项是所谓“返回地址”，是一个程序指令指针，指向本函数一旦完成后应从什么地址继续执行指令。当函数 routine 完成之后，该项的值将传递给 EIP 寄存器，使操作系统能够找到继续执行的指令位置，显然在正常情况下这一位置实在函数 routine 当前的父函数的函数体内。最后，注意栈帧中这些项的相对位置，这是由 C 编译器生成目标机器代码的方式决定的[3]。

一旦 buffer[]被装入超长的内容，则 buffer[]在栈帧中位于其后面的空间必然会被这些超出的字节改写，而且改写可能一致持续到“返回地址”项。如果这一改写是无意的，将很可能导致 routine 返回到一个根本不是指令位置的地址，使进程异常终止。如果超长的数组来自精心准备的入侵者，这一项的值将被改写为一个特别的值，指向 buffer[]内的某个位置，而在这个位置入侵者已经事先放置了实施恶意行为的机器指令(注意这时 buffer[]的内容已经被外部输入的字节串填充，而这些字节正是来自入侵方！)。于是，当函数 routine 完成后，其所在的进程便开始转入病毒为其准备好的指令序列中执行(图 3-1(c)和(d))。

细心的读者会注意到，入侵者如何精确确定“返回地址”项在栈帧中的相对位置？如果不知道这一位置，如何准确放置那个起诱骗作用的返回地址？实际上攻击者不需要精确知道该项的位置，只要略施小计就可以达到目的，例如从某个大致估计的位置之后重复放置那个诱骗地址即可。

---

[2] 从程序本身看, 一个函数可能被多个函数所调用, 但这只是一种静态图像。这里”父函数”指的是动态情况, 即运行期间直接调用本函数的那个函数。因此, 在任何时刻每个函数只有一个父函数。

[3] 学习过编译原理的读者能记得, 编译器把每个函数的调用语句翻译为一组所谓前导指令(prelog),这些指令的目的是为函数调用做准备,其中最重要的工作就是将参数压入堆栈、在栈空间中分配(及初始化)局部变量、以及分配额外空间存储调用的上下文, EBP 和 EIP 两项就来自这些指令的结果。对应地, 函数在完成之后、返回之前需要清除这些空间, 由编译器生成的清场代码(epilog)完成, 其中包括对 EBP 寄存器和 EIP 寄存器赋值。

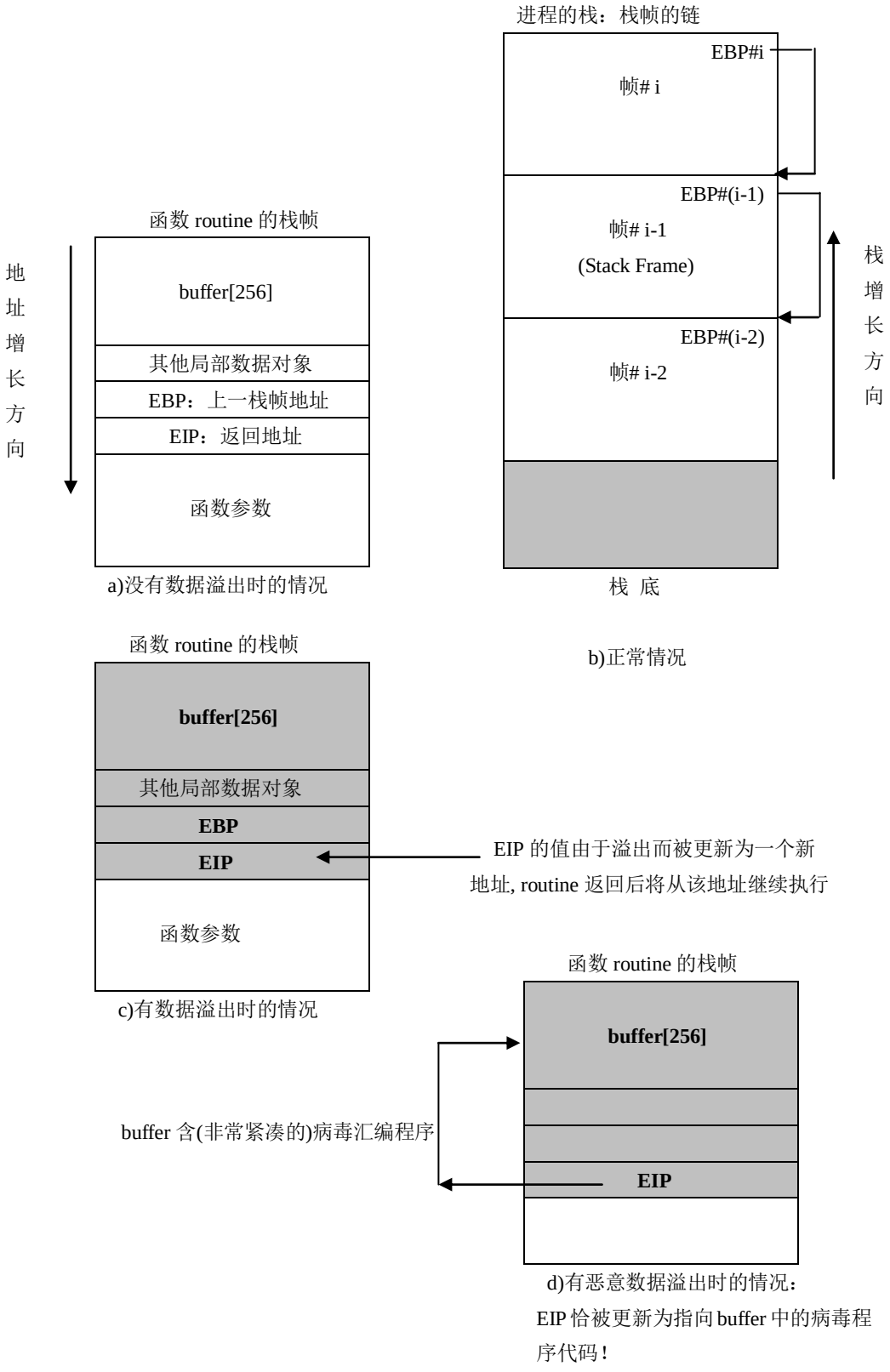

进程的栈：栈帧的链
EBP#i
帧# i
EBP#(i-1)
帧# i-1
(Stack Frame)
EBP#(i-2)
帧# i-2
栈 底
栈增长方向
函数 routine 的栈帧
地址增长方向
buffer[256]
其他局部数据对象
EBP：上一栈帧地址
EIP：返回地址
函数参数
a)没有数据溢出时的情况
b)正常情况
函数 routine 的栈帧
buffer[256]
其他局部数据对象
EBP
EIP
函数参数
EIP 的值由于溢出而被更新为一个新地址, routine 返回后将从该地址继续执行
c)有数据溢出时的情况
函数 routine 的栈帧
buffer[256]
buffer 含(非常紧凑的)病毒汇编程序
EIP
d)有恶意数据溢出时的情况：
EIP 恰被更新为指向 buffer 中的病毒程序代码！

图 3-1　栈溢出攻击

3.2　单字节栈溢出攻击

从上一节的分析清楚地看到，栈溢出攻击所利用的是程序在复制数组时不检查长度是否超界，这无疑是一个编程错误。即使检查是否超界，有时由于粗心的疏忽也导致边界值计算不正确，给入侵者可乘之机。看下面的 C 程序，你能看出什么问题吗？

```
#define MAX_BUFFER_SIZE    256

void routine(char* p) /*p[]是来自外部的子符串*/
{
    char buffer[MAX_BUFFER_SIZE];
    int i, n;
    /*将输入字符串填入 buffer:*/
    n = strlen(p)<=MAX_BUFFER_SIZE ? strlen(p) : MAX_BUFFER_SIZE;
    for( i=0; i<=n; i++)
        buffer[i] = p[i];
    ……
}
```

细心的读者会发现 for 循环在计算数组上界时错误地多计算了一个字节，结果在运行期间可能实际向栈帧写入 257 个字节，即溢出一个字节。如果采用上一节的入侵机制，则攻击者无论如何达不到改写“返回地址”项的目的。但仔细观察图 3-3(a)中“父栈帧”地址项的位置和涵义，不难发现一条间接的攻击途径。为解释这一攻击，这里需要首先解释一个概念，即所谓“Little-Endian”机器和“Big-Endian”机器的区别。

回顾计算机组成原理，我们知道在内存空间每个地址单元物理地存储一个字节。现代计算机指令所能处理的数据越来越长，例如当代 CPU 都是所谓 32 位甚至 64 位机器，指令一次能处理 32 位或 64 位整数。于是自然地出现一个有趣的问题：一个 32 位字 $B_3B_2B_1B_0$ 的四个字节(其中 B_3 是最高字节、B_0 是最低字节)在内存中被如何存储？是如图 3-2(a)那样顺地址增长方向连续存放、还是如图 3-2(b)那样逆地址增长方向连续存放？答案是都可取。事实上不同的 CPU 有不同的约定，分别称为“Little-Endian”机器和“Big-Endian”机器。例如，Intel CPU 就是典型的“Little-Endian”机器，而 IBM 的 PowerPC 是典型的“Big-Endian”机器。

图 3-2 详细描述了单字节溢出攻击。虽然只有一个字节的溢出，但只要“父栈帧地址”项被改变，攻击者就可以在“Little-Endian”机器上(请思考为什么仅在这种机器上以下攻击

才有效？习题 3-2)使函数 routine 返回其父函数之后(注意这时没能改变“返回地址”项，因此 routine 完成后确实返回到其父函数而非任何别的地方)，其父函数在一个伪造的、而非原来的栈帧上继续执行。既然函数的返回地址总是其栈帧内容的一部分，而现在这一栈帧完全由入侵者炮制，因此入侵者完全可以通过伪造一个指向病毒程序的返回地址，使父函数完成后被返回到执行病毒指令！

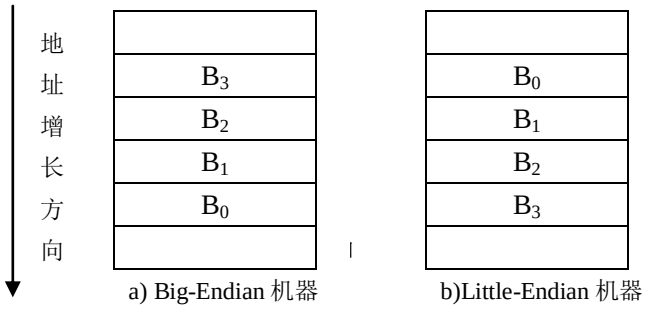


图 3-2　Little-Endian 和 Big-Endian 机器的区别

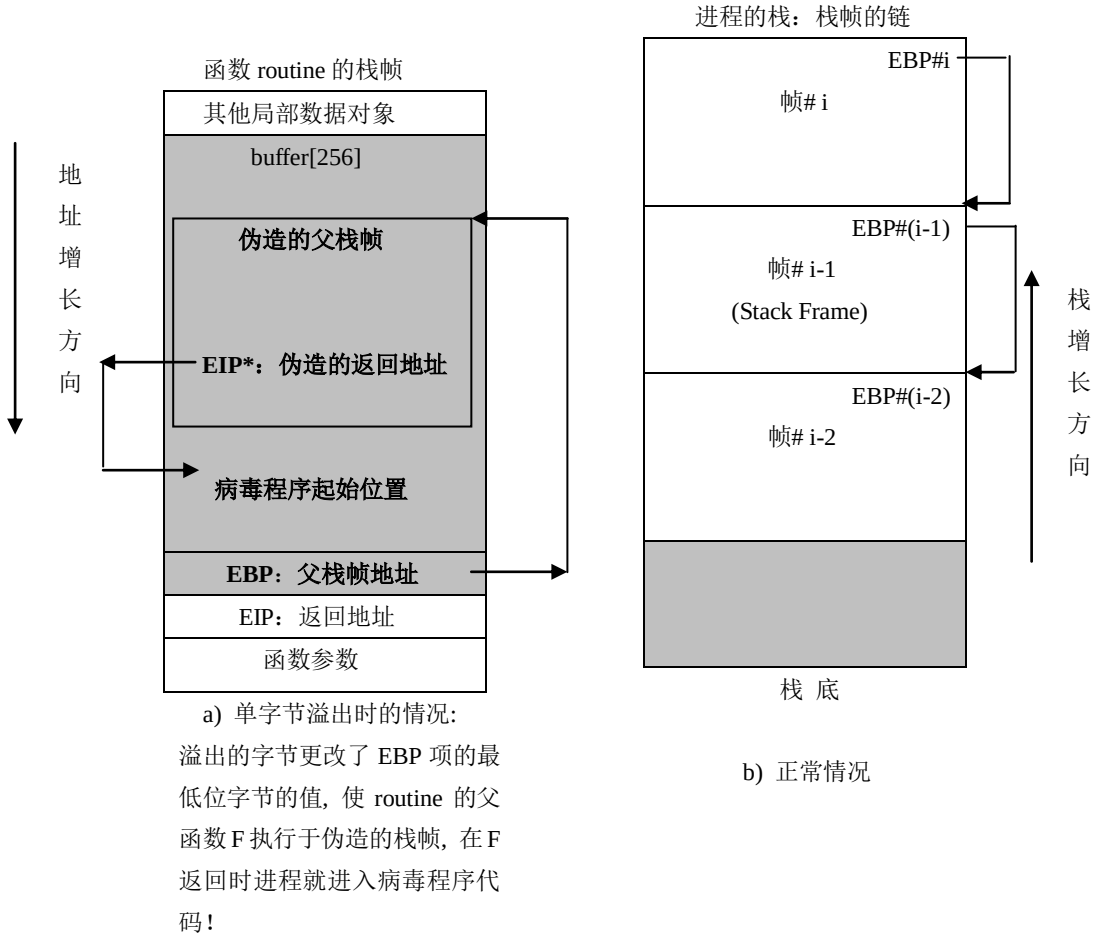


图 3-3 Little-Endian 机器上的单字节溢出攻击

细心的读者会注意到，成功实施这一攻击的关键在于能够成功伪造出父函数的栈帧。父函数的栈帧记录的是父函数在调用函数 routine 之前的临时状态，routine 返回后其父函数需要从这一瞬间状态继续向后执行。如果父栈帧伪造不当导致父函数执行发生错误，不能正常结束而返回，则攻击者将达不到执行病毒代码的目的。此外，和上一节的入侵比较起来，因为伪造的父栈帧需要占用一定空间，buffer[]中留给病毒程序的空间要比栈溢出攻击可利用的空间少许多，病毒程序必须高度紧凑，这些都增加了单字节溢出攻击的难度。但不幸的事实是，所有这些并没有难倒攻击者。

3.3　堆溢出攻击

回顾操作系统，堆(heap)在进程运行期间由操作系统动态分配并映射成为进程空间的一部分。溢出攻击除了可以针对栈(stack)上的数组变量实施之外，也可以针对堆上的变量实施，攻击机理是类似的，这就是所谓堆溢出攻击技术，目的仍然是通过溢出篡改合法数据，再通过这一篡改改变当前进程的执行流。作为第一个例子，看下面的 C 程序：

```
#define MAX_BUFFER_SIZE 256
int main(int argc, char** argv){
{
    int i=0, ch;
    FILE *f;
    static char buffer[MAX_BUFFER_SIZE];
    static char* szFilename = “c:\\procfile.txt”;

    ch = getchar();
    while(ch!=EOF){ /*可能导致 buffer[]溢出  */
        buffer[i] = ch; ch = getchar();   i++;
    }
    f = fopen(szFilename, “w+b”);
    fputs(buffer, f);
    …….
    fclose(f);
}
```

以上程序中的变量 buffer []和 szFilename 在进程空间中的布局如图 3-4(a)所示。根据程序，buffer[]由来自程序输入流的字节顺序填充，如果攻击者炮制的输入流超长而且恰好改写了紧邻 buffer[]的堆变量 szFilename，这意味着该程序将打开一个攻击者指定的文件进行

I/O 操作！如果这是一个特权进程，将意味着攻击者有机会存取操作一个本来没有权限存取的敏感文件。

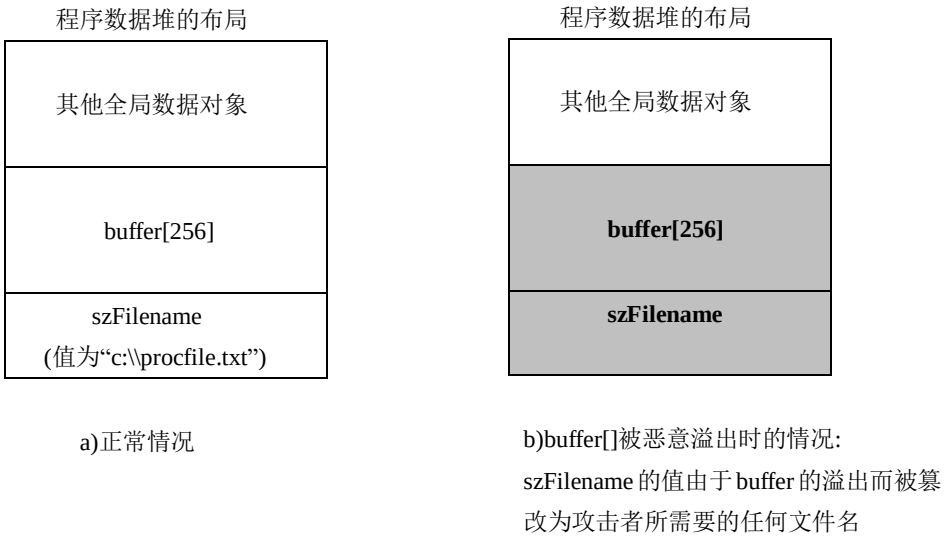


图 3-4　堆溢出攻击的例子：篡改变量值

利用堆溢出攻击还有可能篡改进程空间中的函数地址以改变进程流程。看下面的 C 程序：

```
#define MAX_BUFFER_SIZE 256
int CallBack(const char* szTemp){
    ……
}
int main(int argc, char** argv){
{
    static char buffer[MAX_BUFFER_SIZE];
    static int (*funcptr)(const char*); /*函数指针*/

    funcptr = (int(*)(const char*))CallBack;

    strcpy(buffer, argv[1]);
    /*Size unchecked!*/

    (int)(*funcptr)(argv[2]);
    …….
}
```

以上程序没有检查 argv[1]是否超出的长度，从而使攻击者有可能通过溢出来篡改函数指针 funcptr 的值，结果将以 argv[2](注意该参数也来自攻击者！)为参数执行的不再是预期

的函数 CallBack，而是攻击者指定的任何函数。图 3-5 描述了这一过程。

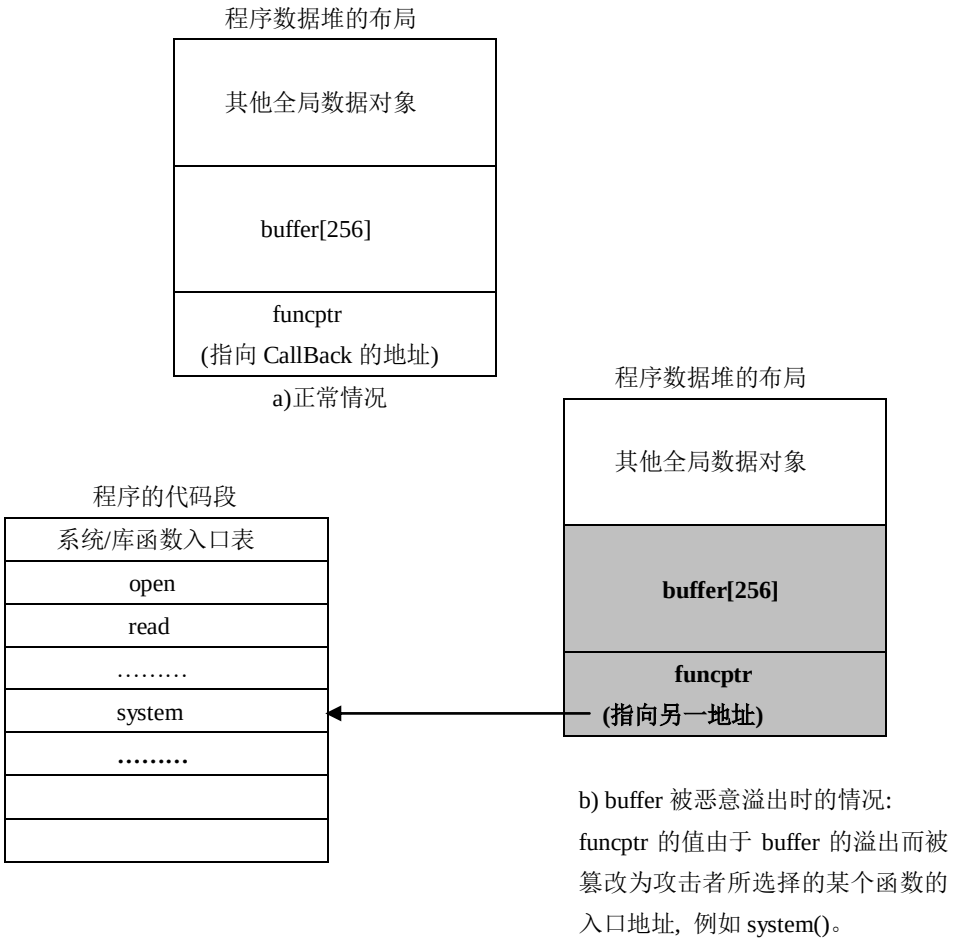


图 3-5　堆溢出攻击的例子：篡改函数指针

一个需要进一步思考的问题是，攻击者如何改写函数指针 funcptr 的值才最有利呢？你可能想到类似于栈溢出攻击那样，使其指向 buffer[]中的病毒代码(根据前面的程序，argv[1]正是装载病毒代码的输入串)，但实际上有更简单的做法，那就是将 funcptr 的值改成某个系统调用的入口地址，例如 system()。这样做之所以可能，是因为在某些应用程序中(实际上是在某些操作系统中)特定系统/库函数在编译后的逻辑位置是可以预测的，参考图 3-5(b)。为了使这样的攻击圆满成功，攻击者还需要伪造(例如)system()的调用参数(对前面的程序就是 argv[2]的值)及当前栈帧。关于这一攻击的一些进一步有趣的问题留做习题 3-3.

3.4　小节及进一步学习的指南

本章分析了一些典型的基于溢出攻击实施入侵的途径，和前一章一样论述的都是“黑客

成就”。当前的网络病毒常常综合运用这些手段，表 3-1 总结了一些以实施溢出攻击为主的著名网络病毒的实例。

表 3-1　现实的病毒实例

病　毒/首次爆发年代	所针对的系统	入侵机制
Morris NetWorm/1988	VAX 上的 Fingerd	栈溢出：利用了 Fingerd 上的 512 字节输入缓冲区，该区未做长度检查；Morris NetWorm 利用含病毒指令的 536 字节消息入侵 Fingerd。
ADM/1998	Linux 上的 DNS 服务器 BIND（TCP#53）	与 Morris NetWorm 相同
CodeRed/2001	Windows 2K 上的 IIS	同上，通过 HTTP GET 消息溢出 IIS 的输入缓冲区，同时利用格式检测漏洞伪造同义/异构的字符串。
Slapper/2002	Linux Apache/OpenSSL	利用两次堆溢出：第一次溢出用以定位目标堆的位置，第二次溢出用以注入恶意代码。
Slammer/2003	Windows2K/XP/2003 上的 SQL Server	溢出 SQL Server 组件 ssnetlib.dll 中的输入栈。注意该入侵发生在微软发布针对此漏洞的安全补丁 MS02-061 六个月之后！而且被入侵的系统都以（错误的！）最高权限配置 SQL Server 的服务。
Blaster/2003	Windows XP/2K	栈溢出 RPC/DCOM
Badtrans/2001	Windows Outlook	MIME 报首解析漏洞
Nimda/2001	Windows 2K 上的 IIS	IIS 文件夹遍历及 MIME 报首解析漏洞

关于入侵机制的更深入的材料见第 2 章所列举的著作，以及这些著作所附的参考文献。

附录 3.1 网络编程概要

网络编程是理解和实现网络反入侵所必须掌握的技术(自然也是实施网络入侵所必须掌

握的技能)。本书不是网络编程的专门教程，但为方便读者理解和查阅，这里概要阐述网络应用程序的编程技术，主要包括网络编程的标准接口函数 socket API 和高性能网络服务器设计技术。这些内容对第 5 章和整个第 III 部分理解网络安全协议的软件实现也都有用。

A3.1.1 Socket API

TCP 或 UDP 实现因特网的数据传输服务，它们位于操作系统内核。从应用程序的观点看，这类服务是操作系统可被调用的功能之一，问题是应用程序如何调用这类功能？

和操作系统提供功能其它调用的方式类似，首先需要将传输服务做一合理的抽象，然后依据这一抽象模型定义相应的接口函数，即 API。针对传输服务，一个普遍适用的、统一的抽象模型是基于 socket[4]的、端-端的、字节流。具体地讲，这一模型有以下涵义[5]：

第一，两个通讯进程各自创建一个 socket 对象，通过适当定义的参数告知 socket 对象通讯对方是谁、以什么协议(例如 TCP 还是 UDP)进行通讯、通讯过程中的某些输入/输出问题如何处理(例如以 TCP 传输数据时，是有任何字节到达时即通知应用程序，还是到达指定的临界字节数后才这样做)，等。

第二，一旦通讯开始，socket 对象负责处理传输过程中一切事务，例如当以 TCP 通讯时，对到达数据的确认、对丢失数据的重传、对 TCP 流量的窗口调节等等均由 socket 对象控制，所有这些对应用程序都是透明的。socket 对象向应用程序所呈现的是一个完全可靠的通讯过程，无须应用程序处理任何传输细节。

第三，socket 对象在网络上传输的仅仅是字节流，即 socket 不对被传输信息的结构做任何假设，所有关于被传输信息的结构和含义的解释都由应用程序完成。

第四，socket-对象为应用程序提供通讯功能的方式与具体协议族无关。事实上，这一模型既适合 TCP/IP，也同样适合其它如 OSI 协议族。

在操作系统内核，socket 对象被实现为操作系统的文件对象。事实上，从应用程序的观点看，socket 对象与一切文件对象所提供的服务都一样，都是输入/输出服务，只不过具体输入/输出的操作性涵义不同，但从面向对象系统设计的角度，这些不同的操作性涵义正是应该向应用程序隐藏而非显露的部分。作为这一实现方式的结果，每个 socket 对象在创建后(和文件对象一样)以一个唯一的文件描述符来标识，在这之后所有的调用都带以该文件描

[4] 中文文献常译为套接字。
[5] 请读者回顾面向对象分析与设计的概念, socket 模型的特点实质上都是面向对象设计所具备的特点。

述符做参数。在这些调用—它们本质上都是 socket 对象上的方法(即 method，面向对象的术语)—中，直接实现输入/输出功能的就是 read 和 write，这两个函数的形式与文件对象上的 read 和 write 函数完全一样。

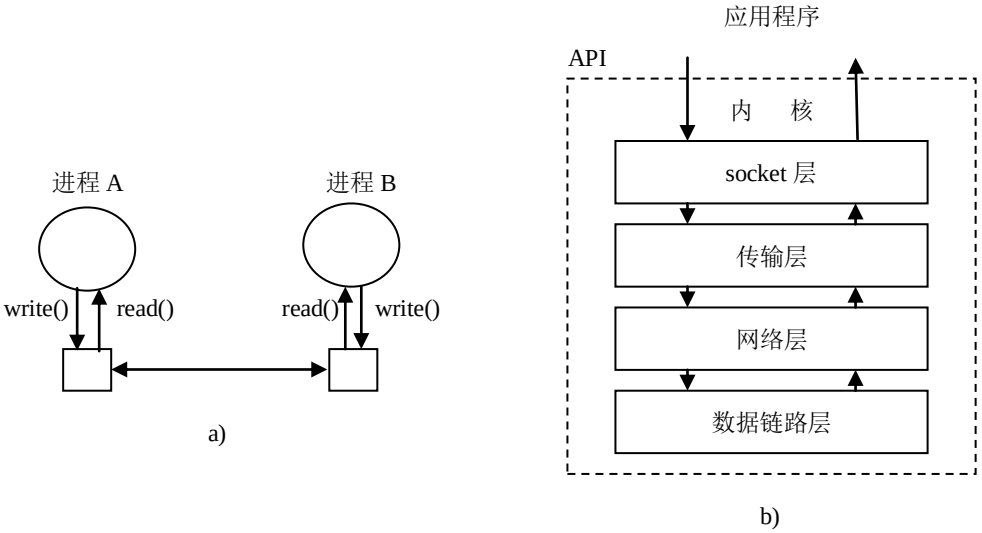


图 A3-1　socket 模型

现在概要解释几个最常用的 socket API-函数的主要功能。在具体应用时，读者需要参考更详细的材料，特别是每个函数的错误返回代码，这对编写实际应用程序是非常有用的。

int socket(int afam, int type, int protocol)

创建一个 socket 对象，返回 socket 对象的文件描述符。参数 afam 指定协议族，例如 AF_INET[6]表示因特网协议；参数 type 表示协议类型，例如 SOCK_STREAM 表示协议族中面向连接的协议，SOCK_DGRAM 表示协议族中的非连接性协议；参数 protocol 一般为 0，表示前两个参数所界定的标准协议。因此，要创建一个 TCP socket 对象，调用 socket(AF_INET, SOCK_STREAM, 0)；创建 UDP 对象，调用 socket(AF_INET, SOCK_DGRAM, 0)。

int bind(int sock, struct sockaddr* localaddr, int addrlen)

对 socket 对象指定地址及传输层端口号，例如对 AF_INET 协议族上的 socket 对象指定就是指定 IP 地址和 TCP 或 UDP 端口号。注意即使程序不调用该函数，操作系统也会自动分配 IP 地址和端口号到新创建的每个 socket 对象，因此这一函数对服务器程序最有用。

int connect(int sock, struct sockaddr* remoteaddr, int addrlen)

从描述符 sock 所标识的本地 socket 对象到 remoteaddr 所指示的远程对象建立连接。如

[6] 在某些操作系统上该常数名字为 PF_INET, 请读者注意具体的编程手册。AF_xxx 表示 Address Family, PF_xxx 表示 Protocol Family。

果 sock 所标识的本地 socket 对象由 socket(AF_INET, SOCK_STREAM, 0)所创建，则这一函数所请求建立的就是 TCP 连接[7]。参数 remoteaddr 中含对方进程的 IP 地址和 TCP 端口号。读者不难想象，函数 connect 主要是被客户进程调用。

int listen(int sock, int queuelen)

listen 使 TCP-socket 对象 sock 准备接受到达的连接请求。参数 queuelen 指示能在该 socket 对象上同时排队等待连接的到达请求的最大数量。

int accept(int sock, struct sockaddr* remoteaddr, int* paddrlen)

这是任何一个基于 TCP 的服务器进程都必须调用的函数，sock 是准备接受连接请求的 socket 对象的描述符(在这一对象上已调用过 listen 函数)。调用 accept 函数的进程或线程进入睡眠，直到有 TCP 连接请求到达并且该连接被正确建立后才被唤醒，这时建立 TCP 连接的三握手过程已经完成，TCP 连接已被建立并且被赋予一个新的(由操作系统自动创建的) socket 对象，accept 的返回值正是该对象的文件描述符。结构*remoteaddr 中含有连接对方的 IP 地址和 TCP 端口号, *paddrlen 是这一结构的实际长度，因此读者不难理解，指针参数 remoteaddr 和 paddrlen 所指向的空间必须在调用 accept 之前已经被分配。

对服务器进程，随后的输入/输出以及关闭连接这些调用都是在 accept 函数返回的那个 socket 对象上进行的。

int read(int sock, char* buff, int size)

int write(int sock, char* buff, int size)

这两个函数并非 socket API 特有，它们就是文件 API，其参数涵义与文件输入/输出时完全相同，只不过现在描述符 sock 所表示的内核对象是 socket 对象。和 accept 一样，这两个函数是阻塞式的，即它们使调用进程或线程睡眠直到其操作完成后返回[8]。

请读者注意，在 UDP 和 TCP-socket 对象上这两个函数都可以被调用，但其操作性涵义有微妙的区别，例如在 UDP-socket 对象上，操作系统在数据发送后便使 write 正常返回，但在 TCP-socket 对象上是在数据被对方确认后才使 write 正常返回，否则使 write 返回错误(write 的返回值为负数)。另外，read/write 函数的行为受 socket 对象配置选项(socket option)控制，例如可以人为设置接收临界水平，使得仅当到达的 TCP 字节总数超过该临界水平时 read 函数才返回，否则一直阻塞；同样，可以设置发送临界水平，使发送缓冲区内积累的字节数超过该临界水平时才真正通过 TCP 发送数据，否则 write 将数据写入发送缓冲区后便返回。

[7] connect()同样可以在 UDP-socket 上被调用, 但意义不同, 这里不详细阐述了。
[8] 读者不难想象, write 引起进程阻塞的机率远小于 read, 除非因为某些原因需要反复重发。

习惯上常对 TCP 通讯使用这两个函数，对 UDP 则使用下面两个函数。

recvfrom(int sock, char* buff, int size, int flags, struct sockaddr* from, int* fromlen)

sendto(int sock, char* buff, int size, int flags, struct sockaddr* to, int tolen)

这两个函数在 UDP-socket 对象上接收和发送 UDP 数据报，也都是是阻塞式函数。

int close(int sock)

在调用这一函数后，socket 对象不再可用，即不能再在该 socket 对象上调用 read/write 这类函数,否则将返回错误。如果是 TCP-socket 对象，从本地 socket 对象到对方 socket 对象的 TCP 连接将被关闭。注意这时并未关闭从对方 socket 对象到本地 socket 对象的 TCP 连接，但本地 socket 对象的内部状态已被置为不可用！这一点和下面的 shutdown 函数截然不同。

如果调用 close 时该 socket 对象上尚有数据等待被读出，这时会发生什么？这时操作系统简单地释放这些数据了事，然后关闭 TCP 连接并置 socket 对象内部状态为不可用。

int shutdown(int sock, int flags)

这是一个较 close 更灵活的函数，它关闭本地 socket 对象到对方 socket 对象的 TCP 连接，这意味着在调用 shutdown 之后不能在本地对象上调用 write，但还可以继续调用 read 正常接受对方发送的数据。

int setsockopt(int sock, int level, int option, char* optval, optlen)

这是一个非常有用的函数，尽管本身非常简单，它的作用在于可以设置大量的 socket 对象配置选项，这些选项用来控制 socket 对象的行为，实际上是 TCP/IP 协议的行为和各种参数。我们不能在此详细列举这些选项的涵义和用法，感兴趣的读者可以查阅本章后面列举的 Stevens 关于网络编程的著作。与这一函数配对的是获取 socket 对象当前配置选项值的函数 getsockopt。

其他辅助函数和宏包括 gethostbyaddr、gethostbyname、gethostname、getpeername、getprotobyname、getprotobynumber、getservbyname、getservbyport、getsockname、htonl、htons、ntohl、ntohs。

A3.1.2　基于 UDP 的客户/服务器系统

图 A3-2 是基于 UDP 的客户/服务器进程相互作用的一般性过程，图中不仅画出应用程序(外侧的两条垂直线)的函数调用过程，而且特别画出了内核中的 UDP 过程(内侧的两条垂直线)，目的是使读者了解 socket API 是如何引发真正的 UDP 通讯过程的。

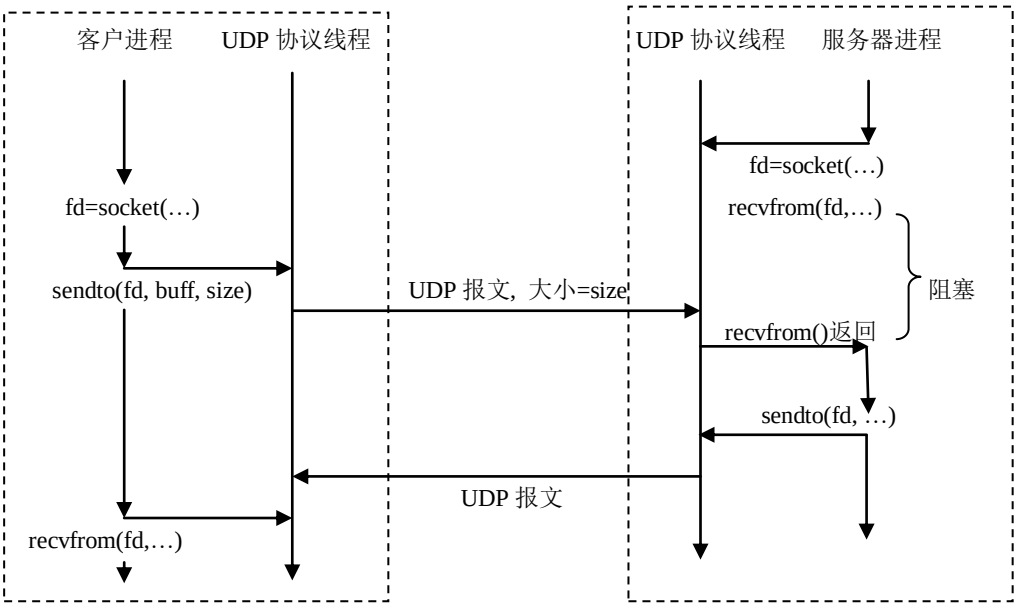


图 A3-2　基于 UDP 的客户/服务器过程

UDP-服务器常见的程序结构如下面这样：

```
int    fd=socket(AF_INET, SOCK_DGRM, 0);

bind(fd, …);
while(1){
      recvfrom(fd, buff, size, ….);
      request_proc(buff, size, …);
      sendto(fd,…);
}
```

注意为了解释主要概念，以上程序没有检查任何 API 的错误返回代码，这在实际编程时是不允许的：必须检查 API 可能返回的错误并做适当处理。实际上，这尤其是服务器编程的重要方面。

正如读者已经理解的，UDP 的最大特点是简单，因此使用 UDP 实现客户/服务器的突出

优点是响应速度快，但 UDP 的另一个突出特点是不可靠，因此某些要求高性能同时也要求一定传输可靠性的服务器常基于 UDP 实现，同时加入一些简单实用的可靠性机制，例如可以仿照 TCP 来实现顺序检查、消息丢失-重传机制。当然，这类补偿机制不应过于复杂，否则不如基于 TCP 来实现服务器。另外一些情形，如视频会议这类要求支持组播的应用，UDP 则是唯一的选择。

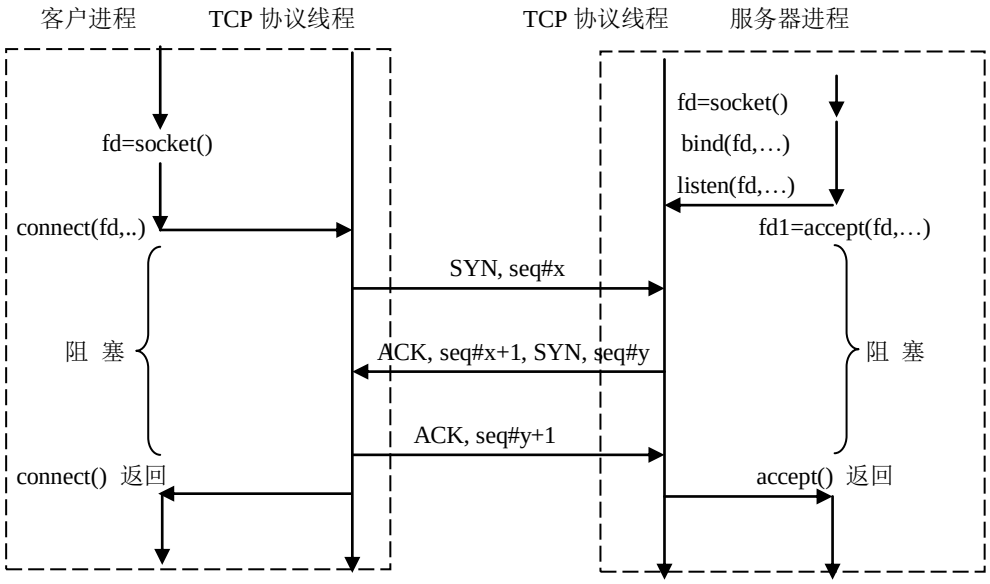
A3.1.3　基于 TCP 的客户/服务器系统

图 A3-3 是基于 TCP 的客户/服务器进程相互作用的一般性过程，与图 A3-2 一样，图中画出了应用程序(外侧的两条垂直线)的函数调用过程和内核中的 TCP 过程(内侧的两条垂直线)之间的关系。TCP-客户进程一般比 TCP 服务器简单，典型的结构如下：

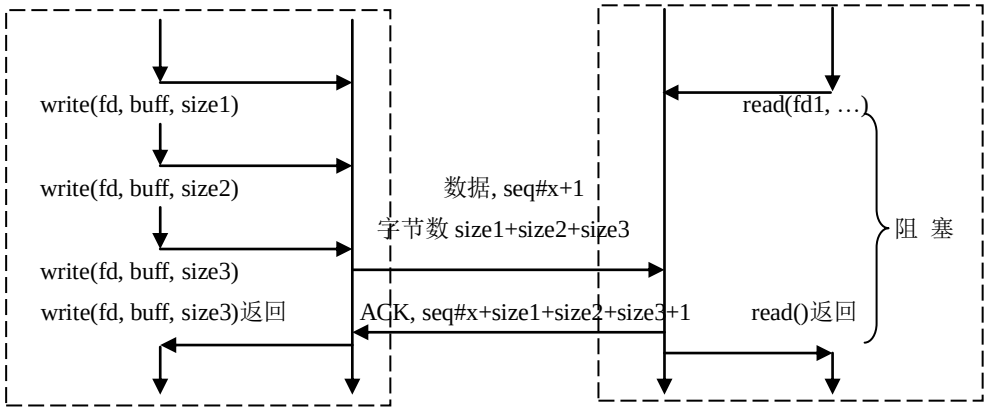
```
int    fd=socket(AF_INET, SOCK_STREAM, 0);

connect(fd, …);
write(fd, …);
read(fd, buff, size);
close(fd);
respond_proc(buff, size, …); /*解析服务器的响应*/
```

最简单的 TCP-服务器进程的结构如下：

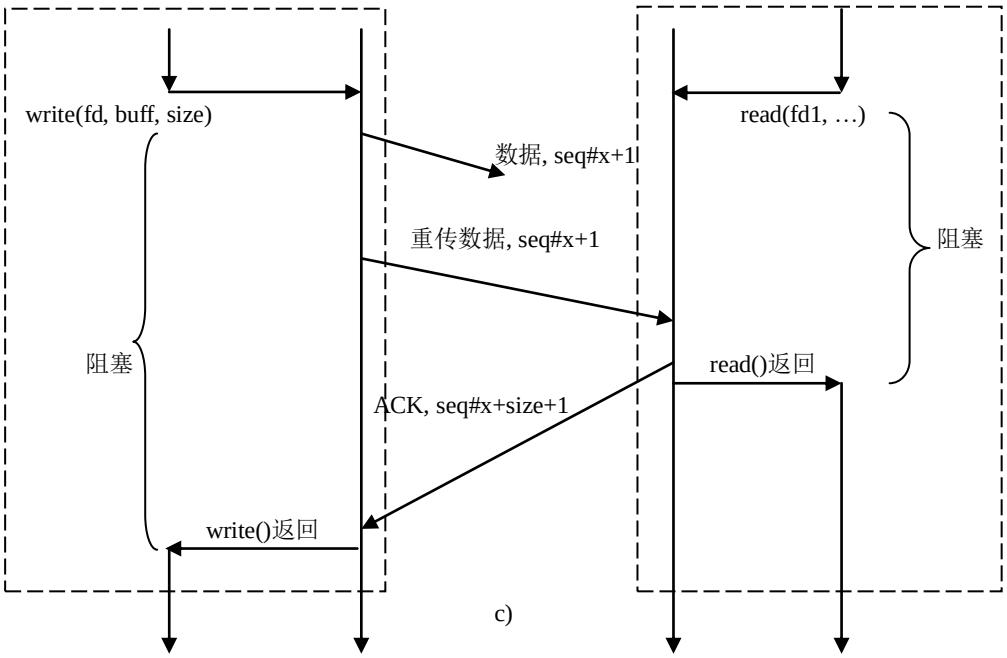
```
 int    fd=socket(AF_INET, SOCK_STREAM, 0);
int fd1;

bind(fd, …);
listen(fd,…);
while(1){
      fd1=accept(fd, ….);
      read(fd1, buff, size);
       /*解析客户请求并响应*/
      ret = request_proc(buff,size,…);
      if(ret>=0){ /*ret<0 表示有错误，例如非法请求*/
           write(fd1,…);
      }
      close(fd1);
 }
```


客户进程
TCP 协议线程
TCP 协议线程
服务器进程
fd=socket()
connect(fd,..)
fd=socket()
bind(fd,…)
listen(fd,…)
fd1=accept(fd,…)
SYN, seq#x
阻 塞
ACK, seq#x+1, SYN, seq#y
阻 塞
ACK, seq#y+1
connect() 返回
accept() 返回

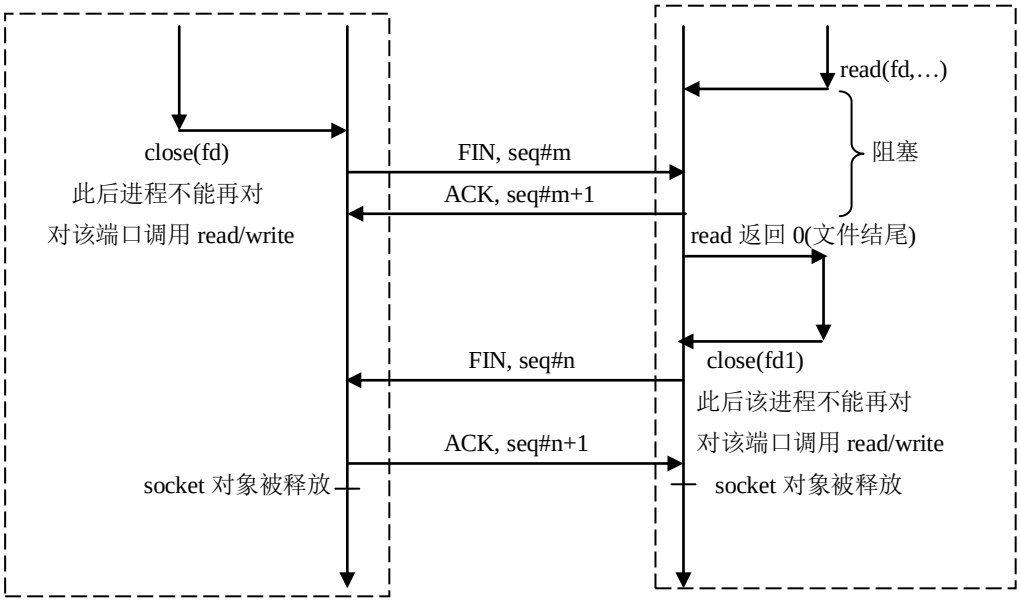
a)


write(fd, buff, size1)
write(fd, buff, size2)
write(fd, buff, size3)
write(fd, buff, size3)返回
read(fd1, …)
数据, seq#x+1
字节数 size1+size2+size3
ACK, seq#x+size1+size2+size3+1
read()返回
阻 塞

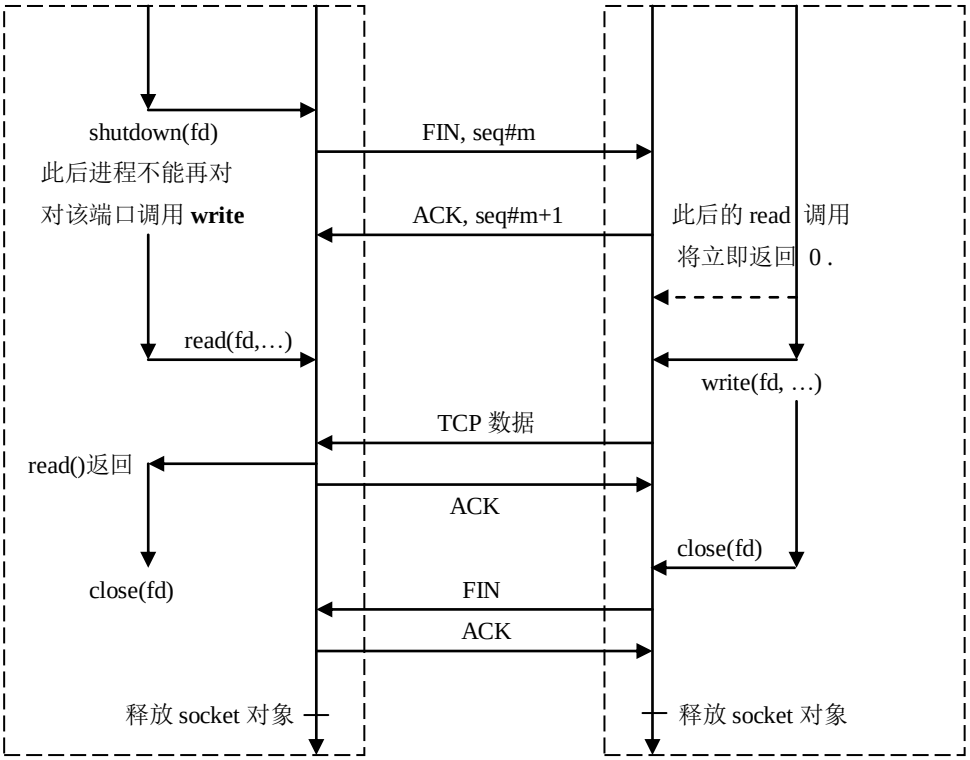
b)


write(fd, buff, size)
数据, seq#x+1
重传数据, seq#x+1
阻塞
ACK, seq#x+size+1
write()返回
read(fd1, …)
阻塞
read()返回

c)



d)



e)

图 A3-3　基于 TCP 的客户/服务器过程

a)建立 TCP 连接　b)缓冲-发送 c)数据成批发送 d)终止连接: 简单终止 e)终止连接: 单向终止

A3.1.3 高性能服务器编程

阅读上一节的 TCP 服务器程序，读者会立刻意识到这是一个典型的串行服务器，它按照到达的顺序依此建立 TCP 连接，一个接一个地处理通过这些连接到来的服务请求。当一个连接上的请求没有处理完时，不会去处理其它连接上的服务请求，哪怕其它连接上的服务要比当前连接上的服务简单得多。如果当前连接上的服务处理需要服务器阻塞一段时间，例如检索很大的磁盘文件或访问大型数据库，服务器也只能经过睡眠-唤醒过程直到当前服务请求被处理完，而不能利用这段时间去处理其它连接上的请求。很明显，这样一种服务器的响应时间和吞吐量都是很不理想的。

从服务器在分布式系统中扮演的角色来看，当然是响应时间越短越好、吞吐量越大越好，特别是吞吐量（对基于 TCP 的服务器而言，可以定义为单位时间内能处理的 TCP 连接数量）常常是度量服务器性能的最重要的指标，吞吐量小也常意味着响应时间长(因为小吞吐量服务器常导致很长的等待服务的排队时间)，因此提高吞吐量常常是服务器程序设计中的重要目标。

提高服务器吞吐量，实质上是要提高服务器支持并发服务的程度，并发性越高，服务器能同时为之服务的请求数量也就越多。既然现代操作系统都支持多进程并发，特别是在 SMP 平台上甚至能实现真正的物理并发，因此对以上串行服务器方案可以自然地改进如下[9]：

```
bind(fd, …);
listen(fd,…);
while(1){
    fd1=accept(fd, ….);
    if(fork()==0){ /*子进程*/
        close(fd);
        read(fd1, buff, size);
        /*解析客户请求并响应*/
        ret = request_proc(buff,size,…);
        if(ret>=0){ /*ret<0 表示有错误, 例如非法请求*/
            write(fd1,…);
        }
        close(fd1);
        exit(0); /*子进程终止*/
    }
}
```

[9] 下面的阐述都是采用 Unix/Linux 的编程技术, 如 fork 函数、父/子进程、pthred_xxx 线程 API 等, 但所有方案都不难针对 Windows 建立起对应的方案。

```
    }
```

这一改进的结果是对每个请求都创建一个单独的进程为之服务，既然这些进程被操作系统并发调度，服务的并发程度也自然随之提高。图 A3-4 是这种方案的运行机制。这种方案也同样适合于实现高吞吐量 UDP 服务器。

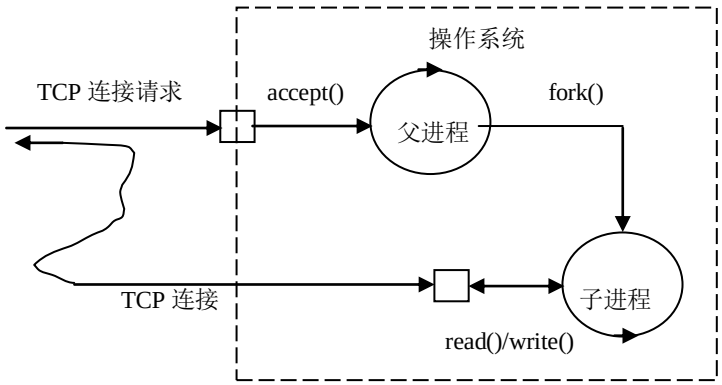


图 A3-4　多进程服务器

另一方面，以上方案也是有代价的：当被同时处理的服务数量很多时，会同时存在同样数量的子进程(且进程数量多于 CPU 数量)，这些进程之间频繁的上下文切换将随着进程数量的增长而成为系统负载的主要因素，因此这一方案提升吞吐量的能力会有一个上限。很明显，操作系统实现上下文切换的代价越小，这一吞吐量的上限也越高。

利用现代操作系统都支持的多线程结构，可以按照这一方案继续提高服务器吞吐量，具体做法是以线程代替子进程来实现并发服务，为每个建立的连接创建一个线程，处理该连接上到来的所有服务请求。

```
/*主线程*/
int   main() {
      int   fd = socket(AF_INET, SOCK_STREAM, 0);
      bind(fd, …);
      listen(fd,…)
      while(1) {
            conn = accept(fd,…);
            /*创建新线程*/
            pthread_create( thread_func, &conn );
      }
      ……
}
/*线程函数*/
int   thread_func( int* conn_fd ) {
      ….. …..
      read(conn_fd, …);
```

```
        解析服务请求并做相应处理;
        write(conn_fd, …); /*回送处理结果*/
        close(conn_fd);
}
```

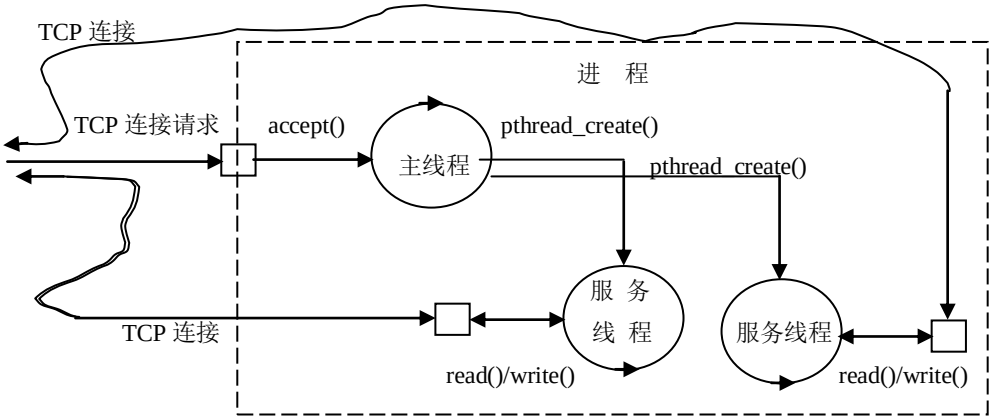


图 A3-5　多线程服务器

当到达的服务请求密度很高时，以上服务器需要在短时间内创建大量线程，尽管创建线程的系统开销比进程要小得多，但短时间内创建很大数量的线程的总开销也会成为限制服务器响应速度的瓶颈，要想提高高密度服务请求下的响应速度，还可以进一步优化以上服务器的多线程调度结构，例如可以将完成服务处理的线程保留一段时间，这样在短期内再次有服务请求到达时，可以直接将该请求分配到空闲的线程，避免了创建新线程的开销，提高响应速度。这就是所谓线程池的思想，其程序结构如下：

```
/*主线程*/
int   main() {
      int   fd = socket(…SOCK_STREAM );
      bind(fd, …);
      listen(fd,…);
      ……
      创建一组线程;
      while(1) {
            conn = accept(fd,…);

            /*分配连接到一个线程.*/
            if(  如果存在一个空闲线程#thrd ) {
                  唤醒线程#thrd;   /*应用题 1*/
            } else {
                  /*没有空闲线程*/
                  pthread_create( thread_func, &conn );
                  唤醒该线程;
            }
```

```
        }
    }

    /*线程函数*/
    int   thread_func( int* conn_fd ) {
        ……..
        while( 1 ) {
            睡眠在预定义的事件上; /*应用题 1*/
            /*被唤醒*/
            read( conn_fd, …);
            request_proc(…);
            write(conn_fd,…);
            …..
        }
    }
```

上面非常概要地阐述了几种提高服务器吞吐量的方法。在实践中，不仅性能，而且服务器的可靠性也非常重要，例如要特别避免使服务器可能长期阻塞的任何条件，并且要仔细设计一旦这类阻塞发生如何使之有效解除。诸如此类的问题，都是网络服务器软件设计中极具挑战性的问题。

A3.1.4 多协议服务器编程

如果要设计一个网络服务器使之能同时提供多个服务，例如同时提供 Web 与 FTP 服务，应该如何做？请读者看下面这一设计有什么问题(暂不考虑性能问题)：

```
/*创建两个 socket 对象*/
int fd_web=socket(AF_INET, SOCK_STREAM, 0);
int fd_ftp=socket(AF_INET, SOCK_STREAM, 0);
int fd1, fd2;

bind(fd_web, …); /*为一个 socket 对象指派 IP 地址和 Web 服务端口号*/
listen(fd_ftp,…); /*为另一个 socket 对象指派 IP 地址和 FTP 服务端口号*/
while(1){
    fd1=accept(fd_web, ….);
    read(fd1, buff, size);
     /*解析客户 HTTP 请求并响应*/
    request_proc_http(buff,size,…);
    write(fd1,…);
    close(fd1);
```

```
    fd2=accept(fd_http, ….);
    read(fd2, buff, size);
     /*解析客户 FTP 请求并响应*/
    request_proc_ftp(buff,size,…);
    write(fd2,…);
    close(fd2);
}
```

这一服务器以此在两个 socket 对象上等待 TCP 连接，然后顺序服务。这里的问题是，如果某段时间内有 FTP 请求到达，但没有 HTTP 服务请求，由于该服务器总是先等待在 Web-socket 对象上，以至于永远无法响应 FTP 请求，直到有某个 HTTP 请求到达并服务完成后，服务器才会执行到 FTP-socket 对象上的 accept 函数。这样一种设计显然是不可行的。

多协议服务器意味着要同时检测多个 socket 对象上的可用状态，包括一组 socket 对象中哪些已经建立 TCP 连接、哪些当前可读、哪些当前可写等。通过上一节的学习，读者会想到用多线程方案实现，每个线程监视并处理一个 socket 对象上的服务。这固然是一种方法，但对性能要求一般的服务器，如何用简单的单线程结构实现呢？这里向读者介绍一个很有用的函数：select[10]。

该函数的原型是 int select(int ignore, fd_set* refds, fd_set* wrfds, fd_set* exfds, const struct timeval* timeout)，参数 refds、wrfds 和 exfds 是三个文件描述符数组，分别监视可读、可写和发生异常的文件描述符(实际上是这些文件描述符所标识的对象)。如果 refds 中有任何一个描述符可读—对一个准备接受 TCP 连接的 socket 对象，其涵义是其上已建立一个 TCP 连接，对已经建立 TCP 连接的 socket 对象，其涵义是有数据到达；或 wrfds 中任何一个描述符可写；或 exfds 中任何一个描述符发生异常，select 返回这些描述符的数量并设置相应数组项的状态，以便程序检查究竟是哪些描述符的状态发生了变化。参数 timeout 用来指定超时，如果从调用 select 开始经过时间 timeout 后没有任何描述符的状态发生变化，则 select 返回 0。timeout 为 0 表示 select 一直阻塞直到有某个任何描述符的状态发生变化。

利用 select，我们可以实现一个单线程的多协议服务器如下：

```
 /*创建多个 socket 对象*/
int fd_web=socket(AF_INET, SOCK_STREAM, 0);
int fd_ftp=socket(AF_INET, SOCK_STREAM, 0);
…….
```

[10] select 源自 Unix System V, 但在 Linux 和 Windows 上也可用。 Win32 中的对应 API 是 WaitForMultipleObjectsEx, 另外还有一个源自 BSD Unix 的功能相似但用法不同的 API poll。

```
int fd1, fd2,…., fdn;

bind(fd_web, …); /*为一个 socket 对象指派 IP 地址和 Web 服务端口号*/
listen(fd_ftp,…); /*为一个 socket 对象指派 IP 地址和 FTP 服务端口号*/
……
while(1){
    FD_SET(fd_web, &refds); /*FD_SET 和下面的 FD_ISSET 是宏*/
    FD_SET(fd_ftp, &refds);
    ……
    select(FD_SETSIZE, &refds, (fd_set *)0, (fd_set *)0, (struct timeval *)0);
    if(FD_ISSET(fd_web, &refds)){
        fd1=accept(fd_web, ….);
        read(fd1, buff, size);
        /*解析客户 HTTP 请求并响应*/
        request_proc_http(buff,size,…);
        write(fd1,…);
        close(fd1);
    }
    if(FD_ISSET(fd_ftp, &refds)) {
        fd2=accept(fd_http, ….);
        read(fd2, buff, size);
        /*解析客户 FTP 请求并响应*/
        request_proc_ftp(buff,size,…);
        write(fd2,…);
        close(fd2);
    }
    ……..
}
```

事实上，select 并非 socket API 特有的函数，它可以应用于一切文件对象，如管道、共享内存等。在 Windows 平台上，有两个类似的 Win32 API ：WaitForSingleObject 和 WaitFroMultipleObjects，还有两个分别带 Ex 后缀的扩展功能的函数(select 也可以直接应用于 Windows 环境)。

习　　题

3-1 重写 3.1 节的函数 routine，改正其错误。

3-2 为什么单字节溢出攻击对 Big-endian 机器无效？

3-3 请仔细查阅 *Windows* 或 *Unix/Linux* 库函数手册，搞懂函数 system 的功能，然后考虑图 3.5 中的攻击及其讨论，思考为什么函数 system 是个对实施攻击很有用的候选？你还能想出其他一些有用(或危险的！)的库函数吗？。

3-4 比较 3.1-3.4 节这些入侵实例，哪一种更为隐蔽，即：哪一个可供检测的特征最少？

3-5 以下是一些有漏洞的程序(不仅限于溢出漏洞)，均源于很坏的编程习惯或极不专业的编程能力，请找出这些漏洞并加以改正：

(1)
```
#define BUFFERSIZE    64
void func(size_t buffersize, char *buf){
    if ( buffersize < BUFFERSIZE){
        char *pBuff = new char[buffersize – 1];
        memcpy(pBuff, buf, buffersize – 1);
        ……
}
```

提示：若 buffersize 为 0 时如何(在 32 位平台上 0 减 1 等于 0xFFFFFFFF!)？若分配内存空间失败又如何？

(2)
```
#define BUFFERSIZE    256
int    ConcatString( char *buf1, char *buf2, size_t len1, size_t len2){
    char buf[BUFFERSIZE];

    if ( (len1 + len2 ) > BUFFERSIZE ) return(-1);
    memcpy( buf+len1, buf2, len2);
    ……
}
```

提示：看上去这个程序检查了超界问题，但假如 len1 是 0x104、len2 是 0xFFFFFFFC 时 len1+len2 却是 256(为什么？)，即“未超界”!

(3) 下面这个程序需要读者熟悉关系数据库中的 SQL 程序，它取自一个天气预报网站，用户只要输入所在区域的邮政编码就可以获得该地区的当前天气报告，其中变量 strPostCode 的值来自用户从界面编辑框所输入的字符串，程序将其(无条件地！)解释为邮政编码：

```
Function String DBLookupByPostCode(strPostCode) {
    Connection = “server=weatherserver; user=sysadmin; password=xyzzyl”;
    String query = “SELECT * FROM weatherdata WHERE postcode= ‘ ” +
                    strPostCode + “ ’ ”;
    String weather = Connection.ExecuteQuery( query );
    Connection.Close();
    Return weather;
}
```

如果你输入 116600，则程序自动构造的 SQL 语句将是

```
SELECT * FROM weatherdata WHERE postcode= ‘ 116600 ’
```

系统返回作者所在区域的天气情况。可如果你输入的是 116600 ’ OR 1=1 -- (--是 SQL 语句的

注释号)，结果怎样？程序自动构造的 SQL 语句将是(为什么?)

SELECT * FROM weatherdata WHERE postcode= ‘ 116600 ’ OR 1=1　-- ’

这时系统返回的信息将是什么？更可怕的是，如果以上程序不是用来访问天气预报数据库而是访问某企业客户的信用卡数据库，后果如何？

请阅读一本好的安全编程指南，精心磨练你的编程能力。