软件安全

第8章 文件输入输出

徐国胜
北京邮电大学

本讲内容安排

1. 文件I/O基础
2. 文件访问控制
3. 文件鉴定
4. 竞争条件
5. 缓解策略

1.1 文件系统

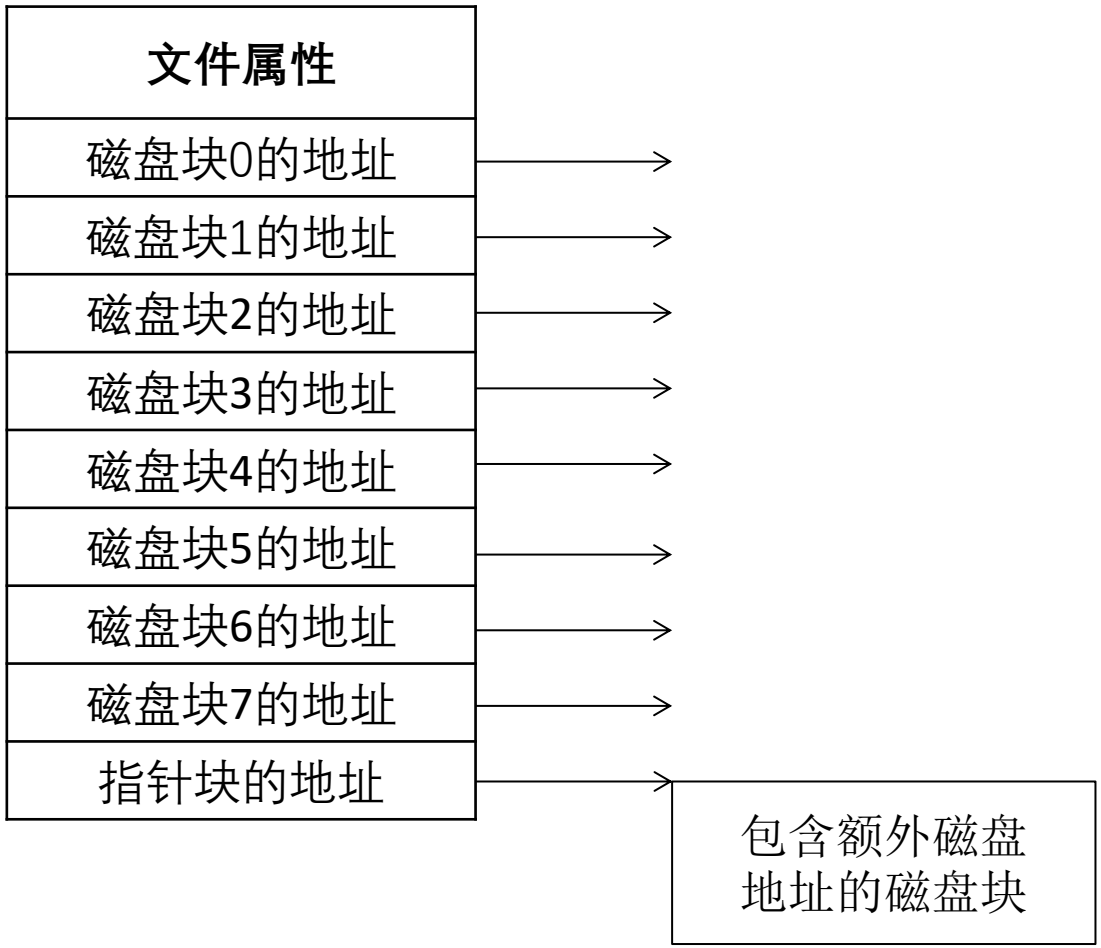
- 程序与文件系统交互方式的不规则性是文件I/O漏洞的根源
  - （1）操作系统、文件系统、文件I/O数量众多
  - （2）每种操作系统能支持多种文件系统、文件I/O接口

- UNIX
  - UFS，UNIX File System
  - 多数厂商改写了UFS以适应自己的用途，增加了可能不被其他厂商的UNIX版本认可的专有扩展
- Linux
  - 早期的MINIX、MS-DOS和ext2文件系统
  - 较新的日志文件系统，如ext4、日志文件系统(Joumaled File System, JFS) 和ReiserFS等
  - 加密文件系统(Cryptographic File System, CFS)和虚拟文件系统/proc

- Mac OS X
  - HFS+：Hierarchical File System Extended Format，分层文件系统扩展格式
  - UPS：BSD标准文件系统格式
  - NFS：Network File System，网络文件系统
  - AFP：AppleTalk文件协议[Mac OS文件共享]
  - UDF：Universal Disk Format，通用磁盘格式
  - ……

- 文件
  - 文件由块(通常在磁盘上)的集合组成
  - UFS的块由与文件关联的i节点(i-node)管理
    - i节点保留所有文件属性，并保持一个固定的地址块数
    - 图8.1是一个i-节点示例，i-节点中的最后一个地址是为一个指向另一个块地址的指针保留的
- 目录
  - 由目录条目的列表组成的特殊文件
  - 目录条目的内容包括目录中的文件名和相关的i-节点的数量

图 8.1 i-节点的示例


文件属性
磁盘块0的地址
磁盘块1的地址
磁盘块2的地址
磁盘块3的地址
磁盘块4的地址
磁盘块5的地址
磁盘块6的地址
磁盘块7的地址
指针块的地址
包含额外磁盘
地址的磁盘块

MS-DOS 8.3文件名

- 用路径(path)名来代替一个文件名
- 路径名包含文件、目录的名称，以及如何浏览文件系统来找到该文件的信息
- 路径名区分绝对路径名和相对路径名，多个路径名可以解析到同一个文件。
- 图8.2显示了路径名的组件

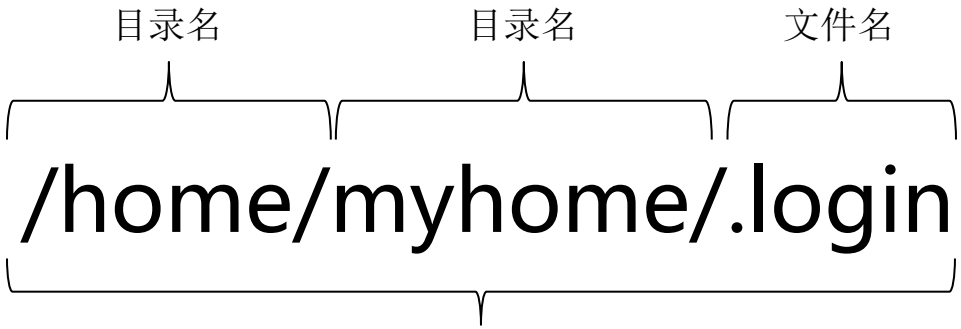


图 8.2 路径名组件

1.2 文件IO接口

- C中的文件I/O包括在<stdio.h>中定义的所有函数
- I/O操作的安全性依赖于具体的编译器实现、操作系统和文件系统
- 较旧的库与较新的版本相比，通常更容易遭受到安全漏洞攻击。

- 字节输入函数
  - fgetc()、fgets()、getc()、getchar()、fscanf()、scanf()、vfscanf()、vscanf()
- 字节输出函数
  - fputc()、fputs()、putc()、putchar()、fprintf()、vfprintf()、vprintf()

- 宽字符输入函数
  - fgetwc()、fgetws()、getwc()、getwchar()、fwscanf()、wscanf()、vfwscanf()、vwscanf()
- 宽字符输出函数
  - fputwc()、fputws()、putwc()、putwchar()、fwprintf()、wprintf()、vfwprintf()、vwprintf()

- 标准C程序在启动时，预定义了三个文本流，操作前不必打开它们：
  - stdin：标准输入(用于读常规输入)
  - stdout：标准输出(用于写常规输出)
  - stderr：标准错误(用于写人诊断输出)

文本流stdin、stdout和stderr是FILE指针类型的表达式。在最初打开时，标准错误流不是完全缓冲的。如果流不是一个交互设备，那么标准输入和标准输出流是完全缓冲的。

1.3 打开、关闭文件
文件打开

- fopen()函数打开一个文件，其名称是由文件名指向的字符串，并把它与流相关联。
  - 1 FILE *fopen(
  - 2 const char* restrict filename,
  - 3 const char* restrict mode,
  - 4);
- 参数mode 指向一个字符串，如果该字符串是有效的，那么该文件以指定的模式打开；否则，其行为是未定义的。

C99支持以下模式

- r:打开文本文件进行读取（只读文件）
- w: 截断至长度为零或创建文本文件用于写入
- a: 追加;打开或创建文本文件用于在文件结束处写入
- rb:打开二进制文件进行读取
- wb: 截断至长度为零或创建二进制文件用于写入
- ab: 追加;打开或创建二进制文件用于在文件结束处写入
- r+: 打开文本文件用于更新(读取与写入)
- w+: 截断至零长度或创建文本文件用于更新
- a+: 追加;打开或创建文本文件用于在文件结束处更新和写人
- r+b或rb+: 打开二进制文件用于更新(读取与写入)
- w+b或wb+: 截断至长度为零或创建二进制文件用于更新
- a+b或ab+: 追加;打开或创建二进制文件用于在文件结束处更新和写入

- C11增加的独占模式
  - wx: 创建独占文本文件用于写入
  - wbx: 创建独占的二进制文件用于写入
  - w+x: 创建独占的文本文件用于更新
  - w+bx或wb+x: 创建独占的二进制文件用于更新

文件关闭

- fclose()函数用来关闭文件流，任何未写入的缓存数据流被传递到主机环境，并被写入到该文件中。任何未读的缓存数据将被丢弃
- 关闭相关文件(包括标准文本流)后，一个指向FILE对象指针的值是不确定的。引用一个不确定的值是未定义的行为
- 长度为零的文件(在它上面没有已写人输出流的字符)是否确实存在是实现定义的
- 关闭的文件可能随后被相同或另一个程序的执行重新打开，并且其内容被回收或修改。
  - 如果main() 函数返回到原来的调用者或如果调用exit() 函数时，所有打开的文件在程序终止之前关闭(且所有的输出流被刷新)。
  - 其他终止程序的路径，如调用abort()函数，不必正确地关闭所有文件。因此，尚未写入到磁盘中的缓冲数据可能会丢失。
  - Linux保证，甚至在程序异常终止时，这个数据也被刷新到磁盘文件。

1.4 C++中的文件I/O

- C++中提供与C相同的系统调用和语义，只有语法是不同的。C++的<ostream>库包括了<cstdio>，后者是<stdio.h>的C++版本。因此，C++支持所有的C的IO函数调用以及<iostream>对象。
- C++中的文件流不使用FILE，而使用ifstream 处理基于文件的输入流，用ofstream处理基于文件的输出流，用iofstream同时处理输入和输出的文件流。所有这些类都继承自fstream 并操作字符(字节)。
- 对于使用wchar_t的宽字符IO，使用wofstream、wifstream、wiofstream、wfstream来处理。

- C++提供下列的
  - cin取代stdin用于标准输入
  - cout取代stdout用于标准输出
  - cerr取代stderr用于无缓冲标准错误
  - clog用于缓冲标准错误，对记录日志有用流来操作字符(字节)
  - 对于宽字符流，使用wcout、wcin、wcerr、wclog

- 例8.1是一个简单的C++程序，它从一个文件test.txt中读取字符数据，并将其写入到标准输出

例 8.1 在C++中读写字符数据

```
1   #include <iostream>
2   #include <fstream>
3
4   using namespace std;
5
6   int main(void) {
7       ifstream infile;
8       infile.open("test.txt",ifstream::in);
9       char c;
10      while (infile >> c)
11          cout << c;
12      infile.close();
13      return 0;
14  }
```

本讲内容安排

1. 文件I/O基础
2. 文件访问控制
3. 文件鉴定
4. 竞争条件
5. 缓解策略

2.1 UNIX文件权限

- 权限与特权
  - 特权(privilege)是通过计算机系统委派的权限，特权位于用户、用户代理或替代，如UNIX进程中。**相对主体而言！**
  - 权限(permission)是访问资源所必要的特权，因此它与资源(如文件)相关。**相对客体而言！**
  - 特权模型往往是特定于系统且复杂的。它们往往会出现“完美风暴”，在管理特权和权限中的错误往往直接导致安全漏洞。

UNIX用户与认证

- 用户名用一个用户ID(User ID, UID) 来确定
  - 把一个用户名映射到一个UID所需的信息保存在/etc/passwd文件中
  - 超级UID(root)的UID为0，它可以访问任何文件
- 每个用户都属于一个组，因此也有一个组ID，或GID。用户还可以属于补充组。
- 用户提供自己的用户名和密码给UNIX系统作身份验证。login程序检查/etc/passwd或shadow文件/etc/shadow来确定用户名是否对应到该系统上的有效用户，并检查提供的密码是否与该UID 所关联的密码对应。

- 文件的特权
  - UNIX文件系统中的每个文件都有一个所有者(UID)和一个组(GID)。所有权用于决定了哪些用户和进程可以访问文件，所有权属于文件的所有者或root，这种特权不能被委派或共享。

- 文件的权限
  - 文件权限包括
    - 读:读一个文件或列出一个目录的内容
    - 写:写入到一个文件或目录
    - 执行:执行一个文件或递归一个目录树
  - 文件权限对于下列每种用户类别可以授予或撤销
    - 用户:该文件的所有者
    - 组:属于文件的组成员的用户
    - 其他:不是文件的所有者或组成员的用户

- 文件权限表示
  - 文件权限一般都用八进制值的向量表示，如图8.4所示。在这种情况下，所有者被授予读、写和执行权限；该文件的组成员的用户和其他用户被授予读取和执行权限。

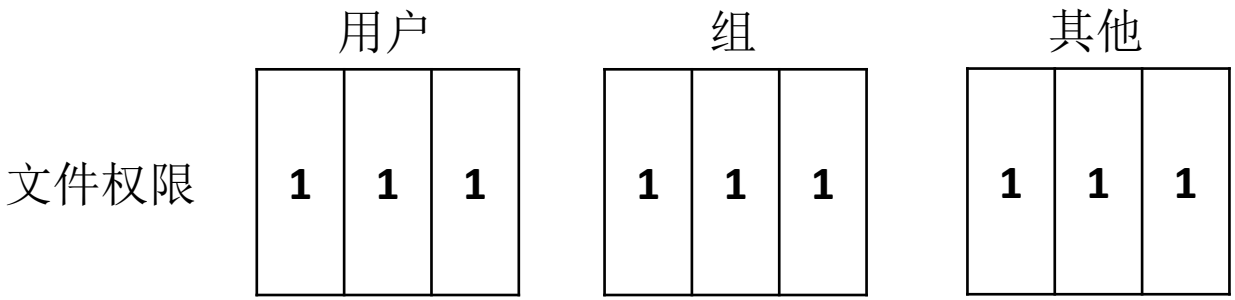


图 8.4 表示文件权限的八进制值向量

- 文件访问控制
  - 当访问一个文件时，进程的有效用户ID (Effective User ID, EUID) 与文件所有者的UID进行比对。如果该用户不是所有者，那么再对GID 进行比较，然后再测试其他。

2.2 进程特权
进程用户ID

- 实际用户ID(RealUser ID, RUID)：RUID是启动该进程的实际用户的ID，它与父进程的实际用户ID是相同的，除非它被改变。
- 有效用户ID(Effective User ID, EUID) 是由内核检查权限时，使用的实际用户ID，因此它确定了进程的权限。如果新的进程映像文件的**设置用户ID模式位被设置**，则新进程映像的EUID被设置为新进程映像文件的用户ID。
- 保存的设置用户ID (Saved Set-User-ID ,SSUID) 是执行时设置用户ID 程序的进程映像文件的所有者ID。

进程组ID，基本上与进程用户ID对应

- 实际组ID (Real Group ID, RGID)是调用该进程的用户的组ID
- 有效组ID (Effective Group ID, EGID)，这是一个由内核检查权限时使用的GID。EGID用于与补充组ID 连用。
- 保存的设置组ID(Saved Set-Group-ID，SSGID) 是执行时设置组ID程序的进程映像文件的所有者的GID。
- 每个进程都维护一个组列表，称为补充组ID (supplementary group ID)，进程在其中有成员关系。当内核检查组权限时此列表用于EGID

进程实例化

- 由C标准system()调用，或由POSIX的fork()和exec()系统调用从父进程继承RUID、RGID、EUID、EGID、补充组ID，以实例化进程。
- 在图8.5所示的例子中
  - 文件的所有者UID是25
  - 一个以RUID 25运行的进程执行文件program 中存储的进程映像，program文件所有者的UID 是18
  - 当程序执行时，它以父进程的权限执行。因此，program程序以RUID和EUID 25运行，并能够访问该UID所拥有的文件

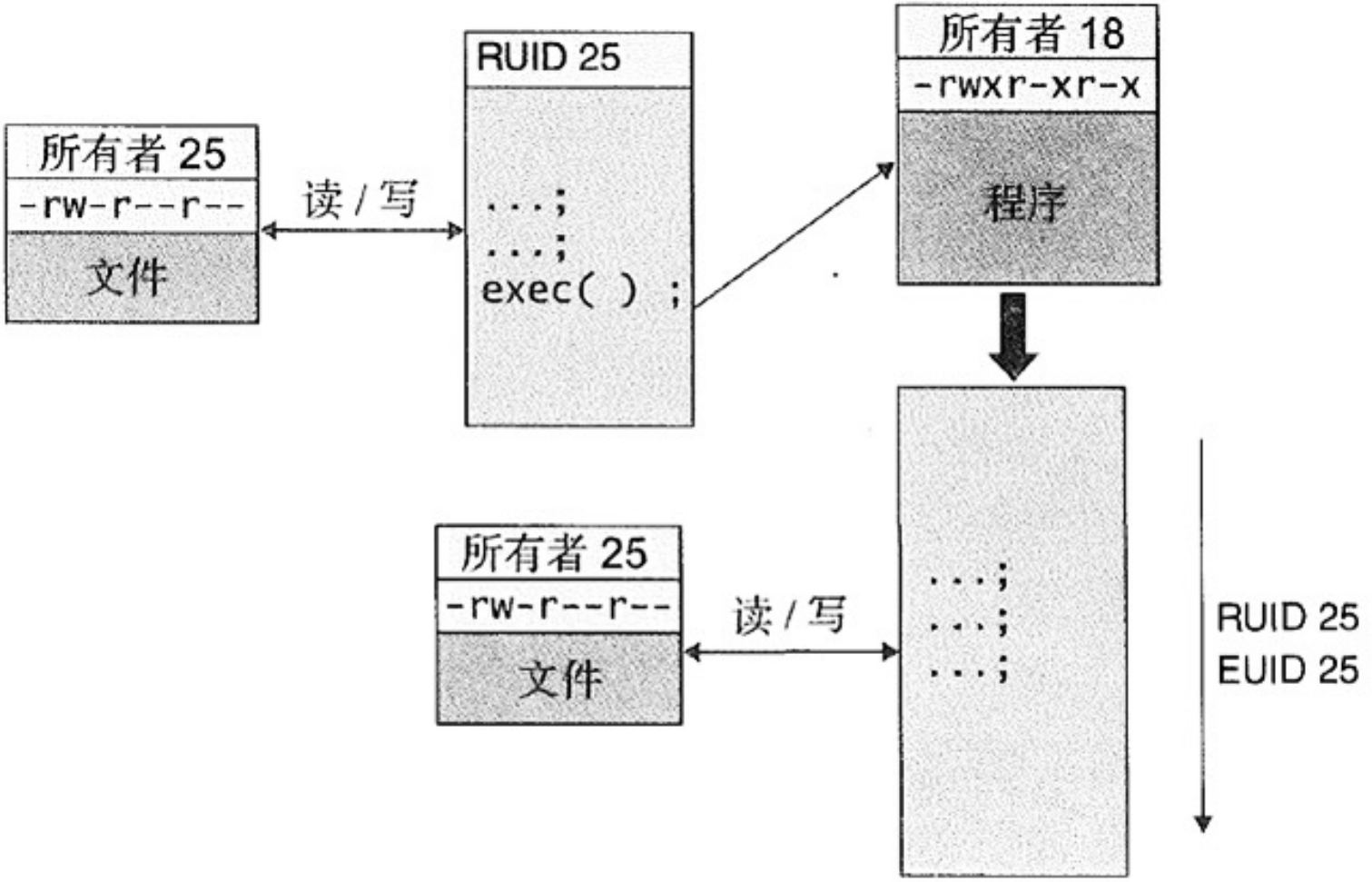


图 8.5　执行非 setuid 程序

- 在如图8.6所示的例子中
  - 一个以RUID 25运行的进程可以读取和写入该用户所拥有的文件。
  - 该进程执行UID18所拥有的文件program中存储的进程映像，但在该文件上设置了执行时设置用户ID位。现在这个过程以RUID 25，但EUID 却是18运行。
  - 此时，程序可以访问UID 18所拥有的文件，但在EUID没有设置为RUID时不能访问UID25所拥有的文件。

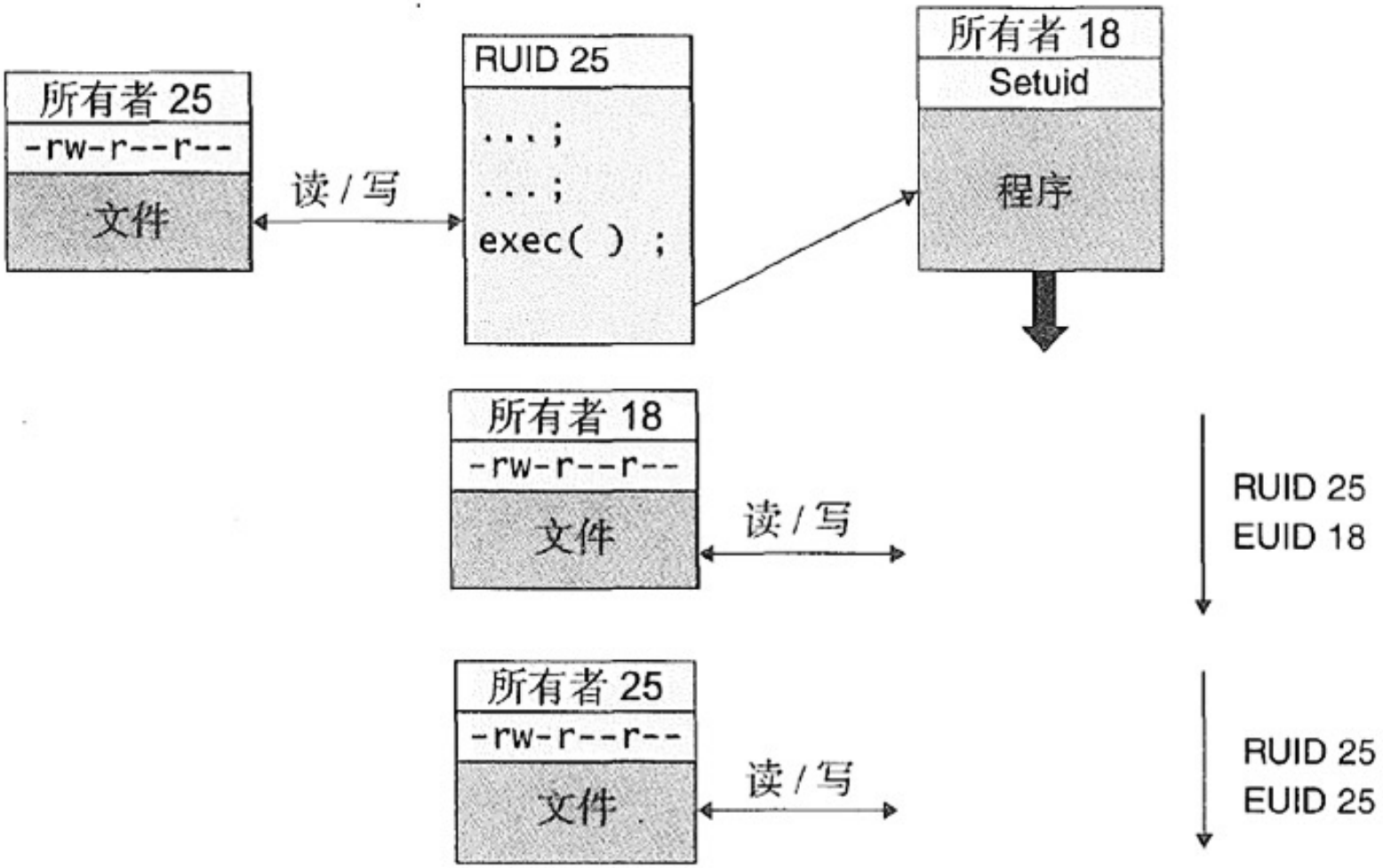


图 8.6 执行 setuid 程序

- 保存的设置用户ID能力允许程序恢复在最后一次exec()调用时建立的EUID。否则，一个程序必须以root身份运行才能执行相同的功能。同样地，保存的设置组ID功能允许一个程序恢复在最后一次exec()调用时建立的有效组ID
- 若要永久放弃特权，则在调用exec()之前把EUID设置为RUID，以使提升的特权不传递给新程序。

2.3 更改特权 - 漏洞利用根源之一

- 如果你的进程正在以提升的特权运行，并访问共享目录或用户目录中的文件，则你的程序就可能会被利用，使得它在程序的用户不具有相应特权的文件上执行操作。
- 提升的特权，可以通过把EUID设置为RUID暂予撤销，它使用操作系统底层权限模型来防止用户执行任何他们没有权限来执行的操作。然而，对于依赖于一个不同的访问控制机制的文件系统，这种方式仍然可能导致不安全的程序（如AFS）。

一个漏洞

- 随FreeBSD 4.4(或更早版本)发布的OpenSSH 版本存在一个特权提升漏洞。这个版本以root特权运行，但并不总是在打开文件之前删除特权

```
1   fname = login_getcapstr(lc, “copyright”,NULL,NULL);
2   if  (fname != NULL && (f=fopen(fname,”r”)) != NULL) {
3       while (fgets(buf, sizeof(buf), f) != NULL)
4           fputs(buf,stdout);
5       fclose(f);
6   }
```

- 此漏洞允许攻击者通过在用户的~/.login_conf 文件中指定配置选项来读取文件系统中的任何文件，例如:copyright=/etc/shadow

2.4 管理特权

- “setuid程序”、"setgid程序“
  - setuid程序是由自己的执行时设置用户ID位设置的程序，setgid程序也是由自己的执行时设置组ID位设置的程序。
  - 不是所有调用setuid()或setgid()的程序都是setuid或setgid程序。setuid程序可以以root身份运行(set-user-ID-root设置用户ID为root)或以更受限制的特权运行。
  - 非root的setuid和setgid 程序通常用于执行有限或特定的任务。这些程序只限于把EUID更改为RUID和SSUID。在可能的情况下，系统应采用这种方法设计，而不是创建设置用户ID为root的程序。

- passwd 程序
  - 由用户运行来更改该用户密码的。它需要打开特权文件，并执行受控的更改，而不允许用户改变存储在同一文件的其他用户的密码。
  - passwd 程序被定义为一个设置用户ID为root的程序:
    - $ ls -1 /usr/bin/passwd
    - -r-sr-xr-x 1 root bin [...] /usr/bin/passwd

Ping程序

- ping程序是一个计算机网络管理实用程序，用于测试主机在互联网协议(Internet Protocol, IP) 网络上是否可达，并测量从源主机发送到目标计算机的消息的往返时间。
- ping程序也是一个设置用户ID为root 的程序:
  - $ ls -1 /sbin/ping
  - -r-sr-xr-x 1 root bin [...] /sbin/ping
- 这个程序做了正确的事并在当它不再需要提升的权限时删除它们，如例8.2所示。

例 8.2 ping程序片段

```
1   setlocale(LC_ALL, “”);
2
3   icmp_sock = socket(AF_INET,SOCK_RAW, IPPROTO_ICMP);
4   socket_errno = errno;
5
6   uid = getuid();
7   if (setuid(uid)) {
8       perror(“ping: setuid”);
9       exit(-1);
10  }
```

setuid程序带来的风险

- 如果所有者是root身份，那么任何事情都是可能的。当编写一个setuid程序时，必须确保对不受信任的用户不采取行动，并确保不给不受信任的用户返回特权信息。还需要确保你遵循了最小特权原则，并在不再需要root权限时把EUID 变更为RUID。
- 它们由用户运行并且在用户(例如，攻击者)控制文件描述符、参数、环境变量、当前工作目录、资源限制、定时器和信号的环境中操作。受控项目清单因UNIX 版本而异，所以很难编写可移植的代码来清理一切。因此，setuid程序要对一些本地可利用的漏洞负责。

本讲内容安排

1. 文件I/O基础
2. 文件访问控制
3. 文件鉴定
4. 竞争条件
5. 缓解策略

3.1 目录遍历

- 特殊文件名
  - “.”指的是目录本身
  - “..”指的是目录的父目录，“..”在根目录中可能指的是根目录本身
  - 在Windows系统上，还可能提供驱动器盘符(例如C:)，以及其他特殊文件名，如“...”，它相当于“../..”。

- 目录遍历漏洞
  - 没有进行足够的验证的路径名会导致目录遍历漏洞
  - 接受“../”形式的输入而没有适当的验证，会允许攻击者遍历文件系统来访问任意文件
    - **例如，/home/../etc/shadow会被解析为/etc/shadow**

许多特权应用程序利用合并用户提供的数据动态地构建路径名

- 例如，假设下面的程序片段作为特权进程的一部分执行，以用于一个特定目录中的文件操作。

```
1  const char *safepath = “/usr/lib/safefile/”;
2  size_t spl = strlen(safe_path);
3  if (!strncmp(fn, safe_path, spl) {
4      process_libfile(fn);
5  }
6  else abort();
```

- 如果该程序所需的文件名参数fn来自非受信的来源(如用户)，那么攻击者可以提供如下的文件名来绕过这些检查。
  - /usr/lib/safefiles/../../../etc/shadow

净化机制

- 用于删除一个目录遍历攻击的特殊文件名，如“.”和“../”
- 然而，攻击者可以试图愚弄净化机制而以一种危险的形式清理数据。例如，假设一个攻击者在文件名里面注入了“.”例如，sensi.tiveFile)，而净化机制删除了这个字符，就会产生有效的文件名sensitiveFile。
- 如果输入的数据现在被认为是安全的，那么该文件可能会受到影响。因此，如果净化被不正确地执行，它可能是无效的或危险的。

- 数据净化举例
  - path = replace(path,"../","");
    - 导致形式为“..../ /”的输入被转换为“../”。
  - path = replace(path, "../", "");path = replace(path, "./", "");
    - 导致形式为 “.../..../ //” 的输入被转化为 “../”

3.2 等价错误

- 当一个攻击者提供不同但等效名字的资源来绕过安全检查时，就会发生路径等价漏洞。
- 做到这一点的方式有很多种，其中有许多是经常被忽视的。
  - 例如，路径名结尾的文件分隔符可以绕过不期望这个字符的访问规则，从而导致一台服务器提供它通常不会提供的文件。
  - 比如：“http://host/./admin/”在功能上等价于 http://host/admin/，但遗憾的是，在这种情况下，它绕过验证。

- 等价错误的另一大类来自区分大小写的问题
  - Macintosh HFS+不区分大小写，因此/home/PRIVATE == /home/private
  - 遗憾的是，Apache的目录访问控制是区分大小写的，因为它是专为UFS设计的，因此/home/PRIVATE != /home/private

- Mac的HFS
  - 数据和资源分支用于存储有关文件的信息。数据分支提供文件的内容，而资源分支存储元数据，如文件类型。
    - sample.txt/..namedfork/rsrc，用于**访问资源分支**
    - sample.txt/..namedfork/data，用于访问数据分支
  - 此字符串相当于sample.txt，可以被用来在能够识别数据分支的操作系统上绕过访问控制。例如，CVE-2004-1084 描述了在苹果的HFS+文件系统上运行的Apache漏洞，远程攻击者能够通过该漏洞直接访问文件数据或资源分支内容。

- 其他等价的错误包括前导或尾随空白、前导或尾随文件分隔符、内部空格(例如，file name)，或星号通配符(例如， pathname*)

3.3 符号链接

- 符号链接(symbolic link)是一个方便的解决文件共享的方案
  - 创建符号链接实际上创建了一个具有独特的i-node节点的新文件。符号链接是特殊的文件，其中包含了实际文件的路径名。

图8.9说明了一个符号链接的例子。在这个例子中，一个i-node为1000的目录包含两个文件条目。第一个文件条目为指向i-node 500的fred1.txt，它是包含各种属性和某种数据的一个正常文件。第二个文件条目是fred2.txt，这是指向i-node 1300的一个符号链接文件。符号链接是一个实际的文件，但此文件仅包含一个到另一个文件的引用，该引用存储为用文本表示的路径。了解这种结构对于理解函数在符号链接上的行为非常有用。

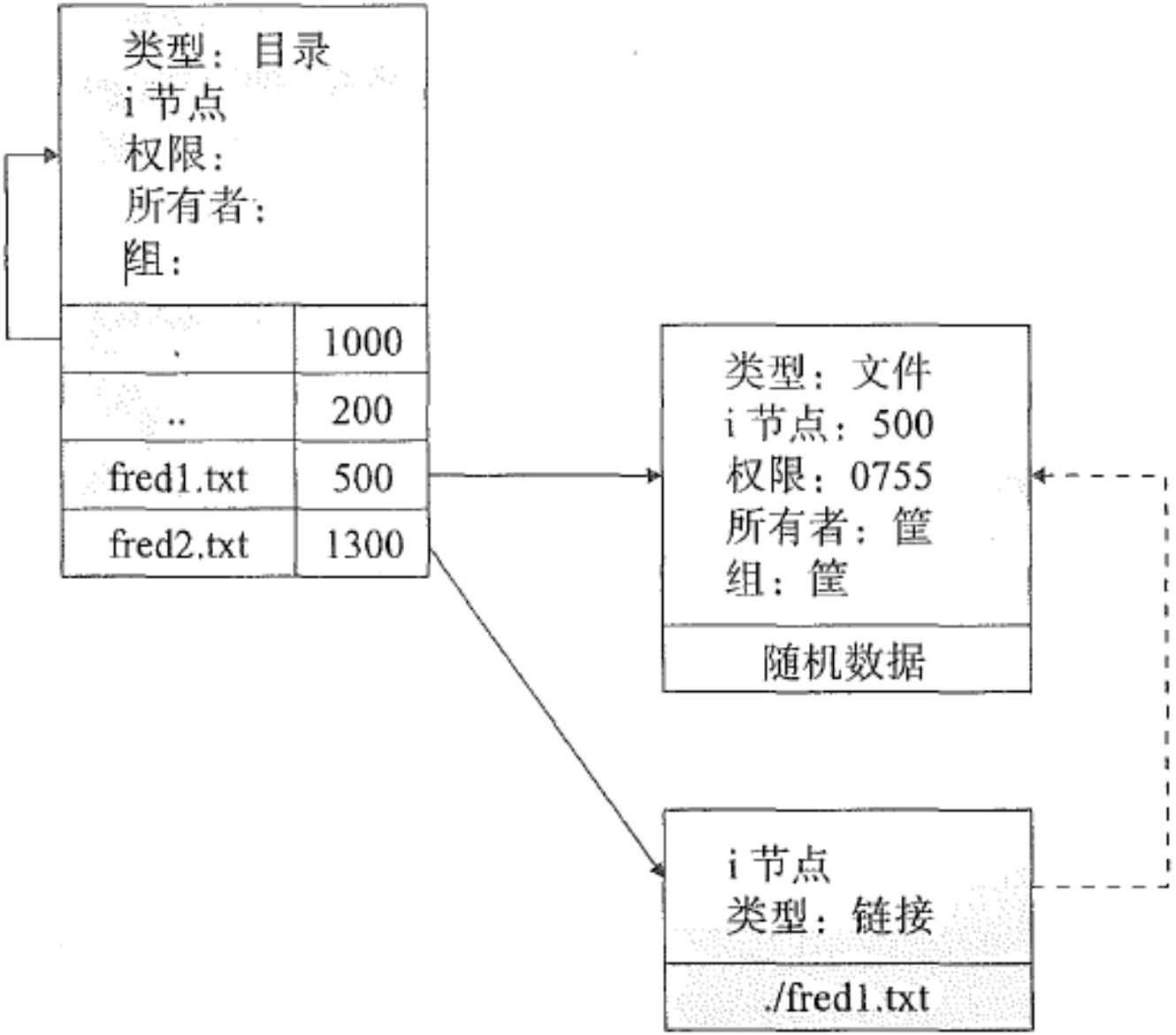

类型：目录
i 节点
权限：
所有者：
组：
.
1000
..
200
fred1.txt
500
fred2.txt
1300
类型：文件
i 节点：500
权限：0755
所有者：筐
组：筐
随机数据
i 节点
类型：链接
./fred1.txt

图 8.9　符号链接

符号链接操作

- 如果路径名称解析过程中遇到符号链接，则用符号链接的内容替换链接的名称。例如，一个路径名/usr/tmp，其中tmp是一个指向../var/tmp 的符号链接，则它被解析为/usr/../var/tmp，这进一步被解析为/var/tmp
- 符号链接上的操作与普通文件操作相似，除非所有下列情况为真：
    - 该链接是路径名的最后一个组件
    - 路径名没有尾随斜线，而且函数需要在符号链接本身上起作用

- 下面的函数在符号链接文件本身，而不是它引用的文件上操作
  - unlink()删除符号链接文件
  - lstat()返回符号链接文件的有关信息
  - lchown()更改符号链接文件的用户和组
  - readlink()读取指定的符号链接文件的内容
  - rename()重命名from参数指定的符号链接，或重写to参数指定的一个符号链接文件

- 例如，假设以一个设置用户ID为root的程序运行下面的代码，它具有有效的root特权
  - 1 fd = open(“/home/rcs/.conf"，O_RDWR);
  - 2 if (fd < 0) abort();
  - 3 write(fd , userbuf , userlen);
- 以上还假定，攻击者可以控制在userbuf 中存储，并在write()的调用中写人的数据。

- 攻击者创建一个从.conf到/etc/shadow认证文件的符号链接:
  - % cd /home/rcs
  - % ln -s /etc/shadow .conf
- 然后运行有漏洞的程序，它以root身份打开文件进行写入，并把攻击者控制的信息写入到密码文件:
  - % runprog
- 例如，这种攻击可以用于创建一个新的、没有密码的root账户。然后，攻击者可以使用su命令切换到root 账号进行root访问:
  - % su
  - #

- 无论是善或恶，符号链接都可以是强大的工具
  - 例如，你可以创建任意文件的链接，甚至在看不到的文件系统中，或不存在的文件中创建。符号链接可以链接到跨分区和磁盘边界的文件，并且符号链接在它们所指向的文件已被重命名、移动或删除后继续存在。更改一个符号链接可以更改使用中的应用程序的版本甚至更改整个网站。
  - 在一个安全的目录如/home/me (主(home) 目录在默认情况下通常具有安全权限设置)，符号链接攻击不是一个问题。如果在一个共享目录，如/tmp目录中操作，或者如果以提升的权限(例如，以管理员身份运行防病毒程序)在一个不安全的目录中操作，就会面临风险。

本讲内容安排

1. 文件I/O基础
2. 文件访问控制
3. 文件鉴定
4. 竞争条件
5. 缓解策略

4.1 一个例子

- 竞争条件可以产生自受信(trusted)或非受信的(untrusted)控制流。受信的控制流包括同一程序内紧密耦合的执行线程。非受信的控制流是一个单独的、并发执行的应用程序或进程，它们的起源往往是未知的。
- 任何支持多任务处理共享资源的系统，都具有源自非受信控制流的竞争条件的可能性。文件和目录通常作为竞争对象。一个文件在一段时间内由独立的函数调用打开、读取或写入、关闭，可能重新打开的文件访问序列，容易造成竞争窗口。打开的文件可以被同等的线程共享，而文件系统可以由独立的进程操纵。

- 在GNU文件实用工具中，曾发现了一个微妙的竞争条件[Purczynski 2002]。软件故障的本质参见例8.12 中的代码。这段代码依赖于一个路径为/tmp/a/b/c 的目录存在。正如注释所指出，竞争窗口在第4行和第6行之间。如果在这个竞争窗口期间执行以下shell 命令，则会构成一个利用:
  - mv /tmp/a/b/c /tmp/c
- 编写这段代码的程序员假定，第6行会将当前目录设置为/tmp/a/b。然而，如果利用代码成功执行，那么第6行的执行会将当前目录设置为/tmp。当代码继续执行到第8行时，也许会在无意之中删除文件。如果这个进程是以root或其他较高的权限执行的，这就特别危险。

例 8.12 GNU 实用文件工具（4.1版）中的竞争条件

```
1  …
2  chdir(“/tmp/a”);
3  chdir(“b”);
4  chdir(“c”);
5  // 竞争窗口
6  chdir(“..”);
7  rmdir(“c”);
8  unlink(“*”);
9  …
```

4.2 检查时间和使用时间

- 文件I/O期间可能出现检查时间和使用时间(Time Of Check, Time Of Use , TOCTOU)竞争条件。首先测试(检查)某个竞争对象属性，然后再访问(使用)此竞争对象，TOCTOU竞争条件形成一个竞争窗口。
- TOCTOU漏洞可能是首先调用stat()，然后调用open()，或者它可能是一个被打开、写入、关闭，并被一个单独的线程重新打开的文件，或者它也可能是先调用一个access()，然后再调用fopen()

- 如例8.13所示。在这个例子中，第7行调用access()函数检查文件是否存在并具有写权限。如果满足这些条件，那么第9行打开该文件用于写入。在这个例子中，access()函数的调用是检查(check)，而fopen()函数的调用是使用(use)。

例 8.13 含有文件打开TOCTOU条件的代码

```
1   #include <stdio.h>
2   #include <unistd.h>

3
4   int main(void) {
5       FILE *fd;
6
7       if (access("a_file",W_OK) == 0) {
8           puts ("access granted.");
9           fd = fopen("a_file","wb+");
10          /* 写入文件 */
11          fclose(fd);
12  }
13  …
14  return 0;
15   }
```

- 在这段代码中的竞争窗口很小，只是在文件已经通过调用access()测试之后，但在打开它之前的第7~9行之间的代码。在那段时间里，一个外部进程有可能在竞争窗口中用某特权文件的符号链接来取代a_file。例如，这可以由一个单独的(非受信)用户在竞争窗口中执行以下shell命令实现：
  - rm a file
  - ln -s /etc/shadow a_file

4.3 创建而不替换

- 例如，下面的代码使用POSIX open()函数打开一个文件用于写入。

```
1   char *file_name;
2   int new_file_mode;
3
4   /* 初始化 file_name 和 new_file_mode */
5
6   int fd = open(
7       file_name, O_CREAT | O_WRONLY,  new_file_mode
8   );
9   if (fd == -1) {
10   /* 处理错误 */
11  }
```

- 如果在open()调用执行时file_name已经存在，那么打开该文件，并截断它。如果file_name是一个符号链接，那么该链接引用的目标文件被截断。攻击者所有需要做的事就是在此调用之前在file_name创建一个符号链接。假设这个有漏洞的过程有相应的权限，那么目标文件将被覆写。
- 使用open()函数的一个解决方案是使用O_CREAT和O_EXCL标志。在一起使用这些标志时，它们指示如果file_name中指定的文件已经存在，那么open()函数就会失败。

```
1   char *file_name;
2   int new_file_mode;
3
4   /* 初始化 file_name 和 new_file_mode */
5
6   int fd = open(
7       file_name, O_CREAT | O_EXCL | O_WRONLY,
    new_file_mode
8   );
9   if (fd == -1) {
10   /* 处理错误 */
11  }
```

- 检查文件的存在并在它不存在时创建文件，这相对于其他正在对同一个目录中相同名称的文件名执行设置O_EXCL和O_CREAT的open() 的线程是原子的。

- 如果设置了O_EXCL和O_CREAT，且file_name是一个符号链接，那么不管符号链接的内容是什么，open()都会失败，并把errno设置为[EEXIST]
- 如果设置了O_EXCL但未设置O_CREAT，那么结果是不确定的。
- 对远程文件系统使用O_EXCL时应小心，因为它不能在NFS版本2上工作。NFS 版本3 中增加了对O_EXCL模式的open()的支持

- 例8.14 显示了在C++中打开一个流之前，测试文件存在的一种常见用法。

代码背后有缺陷的思维是，如果可以打开一个文件用于读取，那么它必须存在。当然，还有其他与文件的存在性无关的原因导致无法打开文件用于读取。
此代码还包含一个TOCTOU漏洞，因为在第9 行测试文件的存在和第13行的打开文件都使用文件名。
再次，这段代码可以通过在执行9行和13行之间的过程中的竞争窗口创建一个具有相同文件名的符号链接来利用其漏洞。

例 8.14 在打开文件时有TOCTOU漏洞的代码

```
1   #include <iostream>
2   #include <fstream>
3
4   using namespace std;
5
6   int main(void) {
7       char *file_name /* = 初始值 */;
8
9       ifstream fi(file_name); // 尝试作为输入文件打开
10      if(!fi) {
11          // 文件不存在，所以它是安全的[原文]
12          // 创建并写入它
13          ofstream fo(file_name);
14          // 写入到file_name;
15          //…
16      }
17      else { //文件存在，关闭并处理错误
18          fi.clise();
19          // 处理错误
20      }
21  }
```

4.4 独占访问

- 由独立的进程产生的竞争条件不能用同步原语来解决，因为这些过程不可能访问共享的全局数据(如一个互斥变量)。
- 通过将文件当作锁来使用，仍可以同步这类并发控制流。例8.15包含两个函数，它们实现了一个Linux文件锁机制。对lock()的调用用于获得锁，而对unlock()的调用则可以释放锁

例 8.15 Linux中的简单文件锁定

```
1   int lock(char *fn) {
2       int fd;
3       int sleep_time = 100;
4       while (((fd = open(fn, O_WRONLY | O_EXCL |
5         O_CREAT, 0)) == -1) && errno ==EEXIST) {
6           usleep(sleep_time);
7           sleep_time *= 2;
8           if (sleep_time > MAX_SLEEP)
9               sleep_time = MAX_SLEEP;
10      }
11      return fd;
12  }
13  void unlock(char *fn) {
14      if (unlink(fn) == -1) {
15          err(1, “file unlock”);
16      }
17  }
```

- lock()和unlock()都接收一个文件名作为共享锁对象
  - 锁文件(lock file)用作锁的代理。如果文件存在，则锁被获得；如果文件不存在，则锁被释放。
  - 这种锁实现机制的一个缺点是open()函数不会阻塞。因此，lock()函数必须反复调用open()函数，直到可以创建文件为止。这种重复有时称为忙等(busy form of waiting)或自旋锁(spinlock)。
  - 例8.15中的代码任意选择了一个100 微秒的初始睡眠时间。这个时间在每次对锁的检查后都翻倍(一直升至某个用户定义的常量时间MAX_SLEEP) 以试图减少对计算时间的浪费。

- 这种类型的锁定的另外一个弊端是，如果持有锁的进程调用unlock()失败，则文件锁将一直被保持。例如，进程崩溃就可能会造成这种情况。常用的修补方案是修改lock()函数，将锁定进程的ID(PID)写到锁文件中。当发现一个现有的锁时，新版lock()将检查文件中保存的PID，并与活动进程列表进行比较。在锁定文件的进程已经终止的情况下才可以获得锁并更新锁文件以包含新的PID。

- 尽管使用这种技术在进程崩溃后进行清理的方法听起来还不错，然而它至少存在3点风险:
  - 1.终止进程的PID有可能被重用。
  - 2.除非非常谨慎地进行实现，否则修补方案可能会包含竞争条件。
  - 3.被锁保护的共享资源可能由于进程的崩溃也受到破坏。

4.5 共享目录

- 当两个或更多用户，或一组用户都拥有对某个目录的写权限时，共享和欺骗的潜在风险比对几个文件的共享访问情况要大得多。
- 程序员经常在对所有用户都是可写(如UNIX上的/tmp和/var/tmp目录和Windows上的C:\TEMP)并可以定期清除(例如，每天晚上或重启时)的目录中创建临时文件。

- 这是一个危险的做法，因为一个在共享目录中的众所周知的文件很容易被攻击者劫持或操纵。缓解战略包括以下内容：
  - 使用其他低级别的IPC (进程间通信)机制，如套接字或共享内存
  - 使用更高级别的IPC 机制，如远程过程调用。
  - 使用只能被应用实例(确保在同一平台上运行的应用程序的多个实例不存在竞争)访问的安全目录或jail
- 在共享目录创建临时文件没有完全安全的方式。为了降低风险，可以把文件创建为具有独特并且不可预知的文件名、仅当文件不存在时打开(原子打开)、用独占访问模式打开、用适当权限打开，并在程序退出之前删除。

独特的并且不可预知的文件名

- 在所有用户可写的目录中创建临时文件的特权程序可以被利用以覆写受限文件。能预测由特权程序创建的文件名称的攻击者，可以创建一个符号链接(与程序所使用的文件具有相同的名称)，指向一个受保护的系统文件。临时文件的名称都必须是唯一的(所以它们不与现有的文件名冲突)并且对攻击者是不可预知的。如果攻击者可以发现随机数生成算法和种子，那么即使是使用随机数发生器为文件命名可能都是不安全的。

- 创建而非替换
  - 仅当此文件不存在时才应建立临时文件。确定该文件是否存在的测试和打开文件必须作为原子操作执行，以消除任何潜在的竞争条件。
- 独占访问
  - 独占访问给予锁定过程无限制的文件访问，而拒绝所有其他进程访问，并消除锁定区域上潜在的竞争条件。
- 适当的权限
  - 应该用执行所需操作(通常由文件的所有者读取和写入)的必要权限的最小集合来打开临时文件。

终止前删除

- 不再需要临时文件时删除它们，允许文件名和其他资源、(如辅助存储器)得以回收。在异常终止的情况下，没有确定的方法，可以保证删除孤立的文件。出于这个原因，广泛采用由系统管理员手动调用或由一个守护进程运行的临时文件来清理实用程序以定期清扫临时目录，并删除旧文件。然而，对于基于文件的攻击，这些工具本身是脆弱的，并经常需要使用共享目录。在正常运行过程中，程序有责任确保删除临时文件，或通过使用库例程，如tmpfile_s()，保证程序终止后删除临时文件
- 表8.4列出了常见的临时文件的函数和它们各自符合这些标准的情况。

表 8.4 临时文件创建函数比较

	tmpnam	tmpnam_s	tmpfile	tmpfile_s	mktemp	mkstemp
	(C)	(Annex K)	(C)	(Annex K)	(POSIX)	(POSIX)
不可预测名字	不可移植	是	不可移植	是	不可移植	不可移植
唯一名字	是	是	是	是	是	是
创建而不替换	否	否	是	是	否	是
独占访问	可能	可能	否	如果操作系统支持	可能	如果操作系统支持
适当权限	可能	可能	否	如果操作系统支持	可能	不可移植
删除文件	否	否	是[a]	是[a]	否	否

注：a 如果程序异常终止，则这个行为是实现定义的。

本讲内容安排

1. 文件I/O基础
2. 文件访问控制
3. 文件鉴定
4. 竞争条件
5. 缓解策略

5.1 关闭竞争窗口

- 互斥缓解方案
  - UNIX和Windows支持很多能够在一个多线程应用程序中实现临界区的同步原语，包括互斥变量、信号量、管道、命名管道、条件变量、CRITICAL_SECTION(临界区)对象以及锁变量等。
  - 线程间的同步可能引入死锁的潜在威胁。当进程从饥饿状态转变为恢复执行时，存在相关联的活锁(live lock) 问题。避免死锁的标准措施是要求资源的获取按照特定的顺序进行。从概念上说，所有要求互斥的资源都可以被编号为r1、r2、…、rn。只要保证进程在捕获资源rk之前已经捕获了所有的资源、rj (其中j < k)，就可以避免死锁。

- 线程安全的函数
  - 在多线程应用程序中，仅仅确保应用程序自己的指令内不包含竞争条件是不够的，被调用的函数也有可能造成竞争条件。当宣告一个函数为线程安全的时候，就意味着作者相信这个函数可以被并发线程调用，同时该函数不会导致任何竞争条件问题。不应该假定所有函数都是线程安全的，即使是操作系统提供的API。当要使用的函数必须为线程安全时，最好去查阅它的文档以确认这一点。
  - 如果必须调用一个非线程安全的函数，那么最好将它处理为一个临界区，以避免与任何其他代码调用冲突。

- 使用原子操作
  - 同步原语依赖于原子(不可分割的)操作。当调用了EnterCriticalRegion()或pthreadmutex-lock()之后，本质上直到函数运行完成为止，该函数都不会被中断。如果一个EnterCriticalRegion()调用允许与另一个EnterCriticalRegion()调用(也许是由另一个线程调用的)重叠，或者与一个LeaveCriticalRegion() 调用重叠，那么这些函数内部可能会存在竞争条件。正是这种原子属性使得这些函数在同步操作中非常有用。

- 重新打开文件
  - 重新打开一个文件流一般应避免，但对于长期运行的应用程序，这可能是必要的，以避免消耗可用文件描述符。由于文件名在每次打开时重新与文件关联，因此无法保证重新打开的文件就是原始文件。

- 例8.16中展示的一种解决方法是使用一个检查-使用-检查(check-use-check) 模式。在这个解决方案中，使用open()函数打开该文件。如果成功地打开了文件，则用fstat()函数把有关该文件的信息读入到orig_st结构。在关闭文件，然后重新打开该文件后，把有关该文件的信息读入new_st结构，并对orig_st和new_st 中的st_dev和st_ino域进行比较，以提高识别的正确性。

例 8.16 检查 – 使用 – 检查模式

```
1   struct stat orig_st;
2   struct stat new_st;
3   char *file_name;
4   /* 初始化file_name */
5   int fd = open(file_name, O_WRONLY);
6   if (fd == -1) {
7       /* 处理错误 */
8   }
9   /*……写入文件……*/
10  if (fstat(fd, &orig_st) == -1) {
11      /* 处理错误 */
12  }
13  close(fd);
14  fd = -1;
15      /* … */
16  fd = open(file_name, O_RDONLY);
17  if (fd == -1) {
18      /* 处理错误 */
19  }
20  if (fstat(fd, &new_st) == -1) {
21      /* 处理错误 */
22  }
23  if ((orig_st.st_dev != new_st.st_dev) || (orig_st.st_ino != new_st.st_ino)) {
24     /* 文件被篡改了！ */
25  }
26  /* ……从文件中读取…… */
27  close(fd);
28  fd = -1;
```

- 这使得程序能够识别出一个攻击者是否在第一次close()和第二次open()之间交换了文件。但是，这个程序不识别该文件是否已被修改。

- 检查符号链接。
  - 例8.17中，“检查-使用-检查”模式的另一个用途是检查符号链接。POSIX的lstat() 函数收集符号链接的信息而不是它指向的目标的信息。该示例使用lstat()函数来收集有关文件的信息，检查st_mode域，以确定该文件是否是一个符号链接，如果它不是一个符号链接，那么打开该文件。

例 8.17 利用检查 – 使用 – 检查模式检查符号链接

```
1   char *filename = /* 文件名 */;
2   char *userbuf = /* 用户数据 */;
3   unsigned int userlen = /* userbuf 字符串长度 */;
4
5   struct stat lstat_info;
6   int fd;
7   /*...*/
8   if (lstat(filename, &lstat_info) == -1) {
9       /* 处理错误 */
10 }
11
12 if{ ! S_ISLNK(lstat_info.st_mode)) {
13      fd = open(filename, O_RDWR);
14      if (fd == -1) {
15          /* 处理错误 */
16      }
17 }
18 if (write(fd, userbuf, userlen) < userlen) {
19      /* 处理错误 */
20 }
```

这个代码在lstat()调用和随后的open()调用之间包含一个TOCTOU竞争条件，因为这两个函数都对同一个文件名进行操作，而该程序的执行可以对此文件名进行异步操作。

- 应用"检查-使用-检查"模式可采取如下步骤：
  - 1. 对文件名调用lstat()
  - 2. 调用open()来打开文件
  - 3. 对open()返回的文件描述符调用fstat()
  - 4. 比较调用lstat()和fstat()返回的文件信息以确保文件是相同的
- 此解决方案如例8.18所示。

例 8.18 利用检查 – 使用 – 检查模式检测竞争条件

```
1    char *filename = /* 文件名 */;
2    char *userbuf = /* 用户数据 */;
3    unsigned int userlen = /* userbuf 字符串长度 */;
4
5    struct stat lstat_info;
6    struct stat fstat_info;
7    int fd;
8    /*...*/
9    if (lstat(filename, &lstat_info) == -1) {
10       /* 处理错误 */
11   }
12
13   fd = open(filename, O_RDWR);
14   if (fd == -1) {
15       /* 处理错误 */
16   }
17
18   if (fstat(fd, &fstat_info) == -1) {
19       /* 处理错误 */
20   }
21
22   if(lstat_info.st_mode ==fstat_info.st_mode &&
23       lstat_info.st_ino == fstat_info.st_ino &&
24       lstat_info.st_dev == fstat_info.st_dev) {
25       if (write(fd, userbuf, userlen) < userlen) {
26           /* 处理错误 */
27       }
28   }
```

- 尽管此代码并未消除竞争条件，但它可以检测任何试图利用此竞争条件把文件替换为符号链接的行为。因为fstat()适用于文件描述符，而不是文件名，所以传递给fstat()的文件与过去打开的文件必须是相同的。lstat()函数不理解符号链接，但open()理解。对于一个符号链接检查，使用st_mode域比较这些模式是足够的。

5.2 消除竞争对象

- 软件开发人员也应该消除对系统资源不必要的使用，以尽量减小漏洞的暴露。比方说，Windows的ShellExecute()函数尽管是为打开一个文件提供了便利的方式，但是这个命令依赖于注册表来选择一个将要应用于文件的应用程序。显而易见，调用CreateProcess()并显式指定应用程序的做法比依赖注册表更可取。

使用文件描述符而非文件名

- 在一个与文件有关的竞争条件中的竞争对象通常不是文件，而是文件所在的目录。比方说，一个符号链接漏洞利用依赖于改变目录条目或是在目录树中更高层的条目，从而改变文件名所指代的文件。一旦一个文件被打开，只要是通过其文件描述符而非文件名的目录(这是竞争的对象)对其进行访问的，该文件就不易受到符号链接攻击。
- 通过使用fchown()代替chown()、使用fstat()代替stat()、使用fchmod()代替chmod()，可以消除很多与文件有关的竞争条件。必须小心使用那些不接受文件描述符的POSIX 函数包括link()、unlink()、mkddir()、rmdir()、mount()、unmount()、lstat()、mknod()、symlink()以及utime()等，并且将它们视作产生竞争条件的潜在威胁。Windows 中仍有可能存在与文件有关的竞争条件，不过概率小得多，因为WindowsAPI 鼓励使用文件句柄而非文件名

5.3 控制对竞争对象的访问

- 最小特权原则
  - 可以通过减少进程的特权来消除竞争条件，而其他时候减少特权仅仅可以限制漏洞的暴露。无论如何，最小特权原则都是一种缓解竞争条件以及其他漏洞的明智策略
  - 竞争条件攻击通常涉及一个策略，攻击者借以使受害代码执行本来没有(也不应该有)权限执行的函数。当然，极端的情况是当受害代码拥有root特权的时候。另一方面，如果正在执行竞争窗口的进程没有比攻击者更高的特权，那么攻击者可以利用的漏洞就很少了。

- 暴露
  - 避免通过用户接口或其他的API暴露你的文件系统的目录结构或文件名。一个更好的方法可能是让用户指定一个键作为标识符。然后，此键可以通过一个哈希表或其他数据结构映射到文件系统中一个特定的文件，而不把文件系统直接暴露给攻击者。

- 存在若干种方式可以将最小特权原则应用于缓解竞争条件
  - 在任何可能的时候，避免运行具有高级特权的进程。
  - 当一个进程必须使用高级特权时，在获取对共享资源的访问前应该使用POSIX特权管理函数或者CreateRestrictedToken()或AdjustTokenPrivileges()(Windows)来去除这些特权。
  - 当创建了一个文件后，应该将权限限制为该文件的所有者(若有必要，稍后可以通过文件描述符调整文件的权限)。某些函数，如fopen()和mkstemp()，要求首先调用umask()来建立创建权限(creation permission) 。

- 安全目录
- 用以检验文件访问权限的算法必须检查的东西不仅仅包括文件自身的权限，还包括从父目录开始，向上至文件系统根目录的每一个包含目录的权限。他们的代码可以避免产生进程间的竞争条件，这是通过“使用-检查-使用”检查模式实现的，以确保当前目录的每一个上级自录都能够被遍历到。这种对当前目录的使用在多线程应用中需要小心实现，以避免出现竞争条件，在其他情况下这是一种检验目录可信度的有效途径。保证文件操作在安全目录中执行也包含验证一个指定目录安全的解决方案。

容器的虚拟化

- 容器提供轻量级的虚拟化技术，让你隔离进程和资源，而不需要提供指令解释机制和其他完全虚拟化的复杂性。容器可以被看作是jail的高级版本，它隔离文件系统、单独的进程ID 、网络命名空间等，并限制诸如内存和CPU 资源的使用。这种虚拟化形式带来的开销通常很少或根本没有，因为在虚拟分区中的程序使用操作系统的正常系统调用接口且不需要仿真或在中间的虚拟机中运行，就像全系统虚拟化技术，如VMware 的情况。容器的虚拟化可用于Linux (lxc , Open VZ)、Windows (Virtuozzo) 和Solaris。标准的Linux支持仍在完善过程中，并包含已知的安全漏洞。商业版本或OpenVZ (Linux 的 fork) 是更加高级的。

5.4 竞争检测工具

- 静态分析
  - 静态分析工具并不通过实际执行软件来进行竞争条件软件分析。这种工具对软件源代码(或者，在某些情况下，二进制执行行文件)进行解析，这种解析有时依赖于用户提供的搜索信息和准则。静态分析工具能报告那些显而易见的竞争条件，有时还能根据可察觉的风险为每个报告项目划分等级。

- 竞争条件侦测已经被证明是一个NP完备问题[Netzer 1990]。因此，静态竞争侦测工具都提供近似的判断。另外，C和C++都是难以进行静态分析的语言，部分原因是C中的指针和指针算术以及C++中的动态调度(dynamic dispatch) 和模板等特性。因此，所有静态分析算法都被证明会导致少许漏报(false negative) (未识别出漏洞)以及频繁的误报(false positive)(被错误地识别的漏洞)。误报的情况需要软件开发人员介入分析。

- 动态分析
- 动态分析工具通过将侦测过程与实际的程序执行相结合，克服了静态分析工具存在的一些问题。这种方式的优势在于可以使工具获得真实的运行时环境。只分析实际的执行流具有一个额外的好处，即可以减少必须由程序员进行分析的误报情况。
- 动态侦测的主要缺点包括：(1) 动态工具无法侦测未执行到的路径; (2) 动态检测通常会带来巨大的运行时开销。

检测工具-IntelThreadChecker

- 一个众所周知的商业工具是来自英特尔公司的。
- Thread Checker对Linux和Windows 上的C++代码的线程竞争和死锁执行动态分析。

检测工具-Helgrind

- Helgrind是Valgrind包的工具之一。它捕获涉及POSIX 线程库的错误，但可能与其他线程原语(例如，在Linux 中的futex() 系统调用)的用法混淆。Helgrind 可以分析用C 、C十十和Fortran 语言编写的程序。
- Helgrind 可能会使性能下降100 倍，因此，它仅对于测试目的有用，而不能作为一个运行时的保护计划。

谢谢大家