第24章　系统程序员安全

Linux系统为程序员提供了许多子程序，这些子程序可存取各种安全属性。有些是信息子程序,返回文件属性，实际的和有效的 UID、GID等信息。有些子程序可改变文件属性。 UID、GID等有些处理口令文件和小组文件，还有些完成加密和解密。

本章主要讨论有关系统子程序，标准C库子程序的安全，如何写安全的C程序并从root的角度介绍程序设计（仅能被root调用的子程序）。

24.1　系统子程序

24.1.1　I/O子程序

• creat （）

建立一个新文件或重写一个暂存文件。需要两个参数：文件名和存取许可值（8进制方式）。如：creat（“ /usr/pat/read_write ”，0666） /* 建立存取许可方式为0666的文件 */调用此子程序的进程必须要有建立的文件的所在目录的写和执行许可，置给 creat （）的许可方式变量将被umask （）设置的文件建立屏蔽值所修改，新文件的所有者和小组由有效的 UID和GID决定。

返回值为新建文件的文件描述符。

• fstat （）

见后面的stat （）。

• open （）

在C程序内部打开文件。需要两个参数：文件路径名和打开方式（ I、O、I&O）。如果调用此子程序的进程没有对于要打开的文件的正确存取许可（包括文件路径上所有目录分量的搜索许可），将会引起执行失败。如果此子程序被调用去打开不存在的文件，除非设置了 O_CREAT标志，调用将不成功。此时，新文件的存取许可将作为第三个参数（可被用户的 umask修改）。当文件被进程打开后再改变该文件或该文件所在目录的存取许可，不影响对该文件的 I/O操作。

• read （）

从已由open （）打开并用作输入的文件中读信息。它并不关心该文件的存取许可。一旦文件作为输入打开，即可从该文件中读取信息。

• write （）

输出信息到已由open （）打开并用作输出的文件中。它同 read （）一样也不关心该文件的存取许可。

24.1.2　进程控制

1. exec （）族

包括execl （）、execv （）、execle （）、execve （）、execlp （）和execvp （），可将一可执行模快拷贝到调用进程占有的存贮空间。正被调用进程执行的程序将不复存在，新程序取代其位置。这是Linux系统中一个程序被执行的唯一方式：用将执行的程序复盖原有的程序。

安全注意事项：

- 实际的和有效的UID和GID传递给由exec（）调入的不具有SUID和SGID许可的程序。
- 如果由exec（）调入的程序有SUID和SGID许可，则有效的UID和GID将设置给该程序的所有者或小组。
- 文件建立屏蔽值将传递给新程序。
- 除设了对exec（）关闭标志的文件外，所有打开的文件都传递给新程序。用 fcntl（）子程序可设置对exec（）的关闭标志。

2. fork（）

用来建立新进程。其建立的子进程是与调用 fork（）的进程（父进程） 完全相同的拷贝（除了进程号外）。安全注意事项：

- 子进程将继承父进程的实际和有效的UID和GID。
- 子进程继承文件方式建立屏蔽值。
- 所有打开的文件传给子进程。

3. signal （）

允许进程处理可能发生的意外事件和中断。需要两个参数：信号编号和信号发生时要调用的子程序。信号编号定义在 signal.h中。信号发生时要调用的子程序可由用户编写，也可用系统给的值，如：SIG_IGN则信号将被忽略，SIG_DFL则信号将按系统的缺省方式处理。如许多与安全有关的程序禁止终端发中断信息（BREAK和DELETE），以免自己被用户终端终止运行。有些信号使Linux系统的产生进程的核心转储（进程接收到信号时所占内存的内容，有时含有重要信息），此系统子程序可用于禁止核心转储。

24.1.3 文件属性

1.access（）

检测指定文件的存取能力是否符合指定的存取类型。需要两个参数：文件名和要检测的存取类型（整数）。

存取类型定义如下：

0：检查文件是否存在。

1：检查是否可执行（搜索）。

2：检查是否可写。

3：检查是否可写和执行。

4：检查是否可读。

5：检查是否可读和执行。

6：检查是否可读可写可执行。

这些数字的意义和chmod命令中规定许可方式的数字意义相同。此子程序使用实际的 UID和GID检测文件的存取能力（一般有效的UID和GID用于检查文件存取能力）。

返回值：0:许可 -1:不许可。

2. chmod （）

将指定文件或目录的存取许可方式改成新的许可方式。

需要两个参数：文件名和新的存取许可方式。

3. chown （）

同时改变指定文件的所有者和小组的UID和GID。（与chown命令不同。）

由于此子程序同时改变文件的所有者和小组，故必须取消所操作文件的SUID和SGID许可，以防止用户建立SUID和SGID程序，然后运行chown（）去获得别人的权限。

4. stat（）

返回文件的状态（属性）。需要两个参数：文件路径名和一个结构指针，指向状态信息的存放的位置。

结构定义如下:

st_mode:	文件类型和存取许可方式。
st_ino:	I节点号。
st_dev:	文件所在设备的ID。
st_rdev:	特别文件的ID。
st_nlink:	文件链接数。
st_uid:	文件所有者的UID。
st_gid:	文件小组的GID。
st_size:	按字节计数的文件大小。
st_atime:	最后存取时间（读）。
st_mtime:	最后修改时间（写）和最后状态的改变。
st_ctime:	最后的状态修改时间。
返回值: 0:	成功1:失败。

5. umask（）

将调用进程及其子进程的文件建立屏蔽值设置为指定的存取许可。

需要一个参数: 新的文件建立屏值。

24.1.4 UID和GID的处理

1） getuid（） 返回进程的实际UID。

2） getgid（） 返回进程的实际GID。

以上两个子程序可用于确定是谁在运行进程。

3） geteuid（） 返回进程的有效UID。

4） getegid（） 返回进程的有效GID。

以上两个子程序用于确定某程序是否在运行某用户而不是其他用户的 SUID程序时很有用，可调用它们来检查确认本程序的确是以该用户的SUID许可在运行。

5） setuid（）： 用于改变有效的UID。

对于一般用户，此子程序仅对要在有效和实际的UID之间变换的SUID程序才有用（从原有效UID变换为实际UID），以保护进程不受到安全危害。实际上该进程不再使用 SUID方式运行。

6） setgid（） 用于改变有效的GID。

24.2 标准C程序库

24.2.1 标准I/O

1. fopen（）

打开一个文件供读或写，安全方面的考虑同open（）一样。

2. fread（）、getc（）、fgetc（）、gets（）、scanf（）和fscanf（）

从已由fopen（）打开供读的文件中读取信息。它们并不关心文件的存取许可。这一点同read（）。

3. fwrite（）、put（）、fputc（）、puts，fputs（）、printf（）、fprintf（）

把信息写到已由fopen（）打开供写的文件中。它们也不关心文件的存取许可。这一点同write（）。

4. getpass（）

从终端上读取至多8个字符长的口令，不回显用户输入的字符。

需要一个参数：提示信息。该子程序将提示信息显示在终端上，禁止字符回显功能，从/dev/tty读取口令，然后再恢复字符回显功能，返回刚敲入的口令的指针。

5. popen（）

将在后面的运行外壳中介绍。

24.2.2 /etc/passwd的处理

有一组子程序可对/etc/passwd文件进行方便地存取，可在入口项对文件读取、写入或更新等等。

1. getpwuid（）

从/etc/passwd文件中获取指定的UID的入口项。

2. getpwnam（）

对于指定的登录名，在/etc/passwd文件检索入口项。以上两个子程序返回一个指向 passwd结构的指针，该结构定义在/usr/include/pwd.h中，定义如下：

```
  struct passwd {
 char * pw_name;  /* 登录名 */
 char * pw_passwd;/* 加密后的口令 */
 uid_t  pw_uid;   /* UID */
 gid_t  pw_gid;   /* GID */
 char * pw_age;   /* 代理信息 */
 char * pw_comment; /* 注释 */
 char * pw_gecos;
 char * pw_dir;   /* 主目录 */
 char * pw_shell; /* 使用的外壳 */
  };
```

3. getpwent（）、setpwent（）、endpwent（）

对口令文件作后续处理。

首次调用getpwent（），打开/etc/passwd并返回指向文件中第一个入口项的指针，保持调用之间文件的打开状态。再调用 getpwent（）可顺序地返回口令文件中的各入口项。调用setpwent（）把口令文件的指针重新置为文件的开始处。使用完口令文件后调用 endpwent（）关闭口令文件。

4. putpwent（）

修改或增加/etc/passwd文件中的入口项。

此子程序将入口项写到一个指定的文件中，一般是一个临时文件，直接写口令文件是很危险的。最好在执行前做文件封锁，使两个程序不能同时写一个文件。算法如下：

• 建立一个独立的临时文件，即/etc/passnnn，nnn是PID号。

- 建立新产生的临时文件和标准临时文件 /etc/ptmp的链，若建链失败，则为有人正在使用/etc/ptmp，等待，直到/etc/ptmp可用为止或退出。
- 将/etc/passwd拷贝到/etc/ptmp，可对此文件做修改。
- 将/etc/passwd移到备份文件/etc/opasswd。
- 建立/etc/ptmp和/etc/passwd的链。
- 断开/etc/passnnn与/etc/ptmp的链。

注意 临时文件应建立在/etc目录，才能保证文件处于同一文件系统中，建链才能成功，且临时文件不会不安全。此外，若新文件已存在，即便建链的是 root用户，也将失败，从而保证了一旦临时文件成功地建链后没有人能再插进来干扰。当然，使用临时文件的程序应确保清除所有临时文件，正确地捕捉信号。

24.2.3 /etc/group的处理

有一组类似于前面的子程序处理 /etc/group的信息，使用时必须用 include语句将/usr/include/grp.h文件加入到自己的程序中。该文件定义了 group结构，将由getgrnam（）、getgrgid（）、getgrent（）返回group结构指针。

1. getgrnam（）

在/etc/group文件中搜索指定的小组名，然后返回指向小组入口项的指针。

2. getgrgid（）

类似于前一子程序，不同的是搜索指定的GID。

3. getgrent（）

返回group文件中的下一个入口项。

4. setgrent（）

将group文件的文件指针恢复到文件的起点。

5. endgrent（）

用于完成工作后，关闭group文件。

6. getuid（）

返回调用进程的实际UID。

7. getpruid（）

以getuid（）返回的实际UID为参数，确定与实际UID相应的登录名，或指定一个UID为参数。

8. getlogin（）

返回在终端上登录的用户的指针。

系统依次检查STDIN、STDOUT、STDERR是否与终端相联，与终端相联的标准输入用于确定终端名，终端名用于查找列于 /etc/utmp文件中的用户，该文件由login维护，由who程序用来确认用户。

9. cuserid（）

首先调用getlogin（），若getlogin（）返回NULL指针，再调用 getpwuid （getuid（））。

10. logname

列出登录进终端的用户名。

11. who am I

显示出运行这条命令的用户的登录名。

12. id

显示实际的UID和GID（若有效的UID和GID和实际的不同时也显示有效的UID和GID）和相应的登录名。

24.2.4 加密子程序

1977年1月，NBS宣布一个用于美国联邦政府ADP系统的网络的标准加密法:数据加密标准即DES用于非机密应用方面。DES一次处理64BITS的块，56位的加密键。

1. setkey（）和encrypt（）

提供用户对DES的存取。

这两个子程序都取64bits长的字符数组，数组中的每个元素代表一个位，为0或1。setkey（）设置将按DES处理的加密键，忽略每第8位构成一个56位的加密键。encrypt（）然后加密或解密给定的64bits长的一块，加密或解密取决于该子程序的第二个变元，0:加密 1:解密。

2. crypt（）

是Linux系统中的口令加密程序，也被/usr/lib/makekey命令调用。

crypt（）子程序与crypt命令无关，它与/usr/lib/makekey一样取8个字符长的关键词，2个salt字符。关键词送给setkey（），salt字符用于混合encrypt（）中的DES算法，最终调用encrypt（）重复25次，加密一个相同的字符串。返回加密后的字符串指针。

24.2.5 运行外壳

1. system（）

运行/bin/sh执行其参数指定的命令，当指定命令完成时返回。

2. popen（）

类似于system（），不同的是命令运行时，其标准输入或输出联到由 popen（）返回的文件指针。

二者都调用fork（）、exec（）、popen（）还调用pipe（），完成各自的工作，因而fork（）和exec（）的安全方面的考虑开始起作用。

24.3 编写安全的C程序

24.3.1 需要考虑的安全问题

一般有两方面的安全问题，在写程序时必须考虑:

• 确保自己建立的任何临时文件不含有机密数据，如果有机密数据，设置临时文件仅对自己可读/写。确保建立临时文件的目录仅对自己可写。

• 确保自己要运行的任何命令（通过system（）、popen（）、execlp（）、execvp（）运行的命令）的确是自己要运行的命令，而不是其他什么命令，尤其是自己的程序为 SUID或SGID许可时要小心。

第一方面比较简单，在程序开始前调用 umask（077）。若要使文件对其他人可读，可再调chmod（），也可用下述语名建立一个“ 不可见 ”的临时文件。

```
creat（"/tmp/xxx",0）;
file=open（"/tmp/xxx",O_RDWR）;
unlink（"/tmp/xxx"）;
```

文件/tmp/xxx建立后，打开，然后断开链，但是分配给该文件的存储器并未删除，直到最终指向该文件的文件通道被关闭时才被删除。打开该文件的进程和它的任何子进程都可存取这个临时文件，而其他进程不能存取该文件，因为它在/tmp中的目录项已被unlink（）删除。

第二方面比较复杂而微妙，由于system（）、popen（）、execlp（）、execvp（）执行时，若不给出执行命令的全路径，就能“骗”用户的程序去执行不同的命令。因为系统子程序是根据PATH变量确定哪种顺序搜索哪些目录，以寻找指定的命令，这称为SUID陷井。最安全的办法是在调用system（）前将有效UID改变成实际UID，另一种比较好的方法是以全路径名命令作为参数。execl（）、execv（）、execle（）、execve（）都要求全路径名作为参数。对付SUID陷井的另一方式是在程序中设置PATH，由于system（）和popen（）都启动外壳，故可使用外壳句法。如:

```
system（"PATH=/bin:/usr/bin cd"）;
```

这样允许用户运行系统命令而不必知道要执行的命令在哪个目录中，但这种方法不能用于execlp（）和execvp（）中，因为它们不能启动外壳执行调用序列传递的命令字符串。关于外壳解释传递给system（）和popen（）的命令行的方式，有两个其他的问题:

• 外壳使用IFS 外壳变量中的字符，将命令行分解成单词（通常这个外壳变量中是空格、tab、换行），如IFS中是/，字符串/bin/ed被解释成单词bin，接下来是单词ed，从而引起命令行的曲解。

• 再强调一次:在通过自己的程序运行另一个程序前，应将有效UID改为实际的UID，等另一个程序退出后，再将有效UID改回原来的有效UID。

24.3.2 SUID/SGID程序指导准则

1） 不要写SUID/SGID程序，大多数时候无此必要。

2） 设置SGID许可，不要设置SUID许可，应独自建立一个新的小组。

3） 不要用exec（）执行任何程序。记住exec（）也被system（）和popen（）调用。

• 若要调用exec（）（或system（）、popen（）），应事先用setgid（getgid（）） 将有效GID置加实际GID。

• 若不能用setgid（），则调用system（）或popen（）时，应设置IFS:

```
popen（"IFS=\t\n;export IFS;/bin/ls","r"）;
```

• 使用要执行的命令的全路径名。

• 若不能使用全路径名，则应在命令前先设置PATH:

```
popen（"IFS=\t\n;export IFS;PATH=/bin:/usr/bin;/bin/ls","r"）;
```

• 不要将用户规定的参数传给system（）或popen（）;若无法避免则应检查变元字符串中是否有特殊的外壳字符。

• 若用户有个大程序，调用exec（）执行许多其他程序，这种情况下不要将大程序设置为SGID许可。可以写一个（或多个）更小、更简单的SGID程序执行必须具有SGID许可的任务，然后由大程序执行这些小SGID程序。

4） 若用户必须使用SUID而不是SGID，以相同的顺序记住2）、3）项内容，并相应调整。不要设置root的SUID许可，选一个其他帐户。

5） 若用户想给予其他人执行自己的外壳程序的许可，但又不想让他们能读该程序，可将程序设置为仅执行许可，并只能通过自己的外壳程序来运行。

24.3.3 编译、安装SUID/SGID程序的方法

1）确保所有的SUID/SGID程序是对于小组和其他用户都是不可写的，存取权限的限制低于4755（2755）将带来麻烦，只能更严格。4111（2111）将使其他人无法寻找程序中的安全漏洞。

2）警惕外来的编码和make/install方法。

- 某些make/install方法不加选择地建立SUID/SGID程序。
- 检查违背上述指导原则的SUID/SGID许可的编码。
- 检查makefile文件中可能建立SUID/SGID文件的命令。

24.4 root用户程序的设计

有若干个子程序可以从有效UID为0的进程中调用。许多前面提到的子程序，当从root进程中调用时，将完成和原来不同的处理。主要是忽略了许可权限的检查。

由root用户运行的程序当然是root进程（SUID除外），因有效UID用于确定文件的存取权限，所以从具有root的程序中，调用fork（）产生的进程，也是root进程。

1. setuid（）

从root进程调用setuid（）时，其处理有所不同， setuid（）将把有效的和实际的UID都置为指定的值。这个值可以是任何整型数。而对非 root进程则仅能以实际UID或本进程原来有效的UID为变量值调用setuid（）。

2. setgid（）

在系统进程中调用setgid（）时，与setuid（）类似，将实际和有效的GID都改变成其参数指定的值。

调用以上两个子程序时，应当注意下面几点：

- 调用一次setuid（）（setgid（））将同时设置有效和实际UID（GID），独立分别设置有效或实际UID（GID）固然很好，但无法做到这点。
- setuid（）（setgid（））可将有效和实际UID（GID）设置成任何整型数，其数值不必一定与/etc/passwd（/etc/group）中用户（小组）相关联。
- 一旦程序以一个用户的UID调用了setuid（），该程序就不再做为root运行，也不可能再获root特权。

3. chown（）

当root进程运行chown（）时， chown（）将不删除文件的SUID和/或SGID许可，但当非root进程运行chown（）时， chown（）将取消文件的SUID和/或SGID许可。

4. chroot（）

改变进程对根目录的概念，调用chroot（）后，进程就不能把当前工作目录改变到新的根目录以上的任一目录，所有以/开始的路径搜索，都从新的根目录开始。

5. mknod（）

用于建立一个文件，类似于creat（），差别是mknod（）不返回所打开文件的文件描述符，并且能建立任何类型的文件（普通文件，特殊文件，目录文件）。若从非root进程调用mknod（）

将执行失败，只有建立FIFO特别文件（有名管道文件）时例外，其他任何情况下，必须从 root进程调用mknod（）。由于creat（）仅能建立普通文件，mknod（）是建立目录文件的唯一途径，因而仅有root能建立目录，这就是为什么mkdir命令具有SUID许可并属root所有。一般不从程序中调用mknod（）。通常用/etc/mknod命令建立特别设备文件，而这些文件一般不能在使用时建立和删除，mkdir命令用于建立目录。当用 mknod（）建立特别文件时，应当注意确保所建的特别文件不允许存取内存、磁盘、终端和其他设备。

6. unlink（）

用于删除文件。参数是要删除文件的路径名指针。当指定了目录时，必须从 root进程调用unlink（），这是必须从root进程调用unlink（）的唯一情况，这就是为什么 rmdir命令具有root的SGID许可的原因。

7. mount（）和umount（）

由root进程调用，分别用于安装和拆卸文件系统。这两个子程序也被 mount和umount命令调用，其参数基本和命令的参数相同。调用 mount（），需要给出一个特别文件和一个目录的指针，特别文件上的文件系统将安装在该目录下，调用时还要给出一个标识选项，指定安装的文件系统可被读/写（0）还是仅读（1）。umount（）的参数需要一个可拆卸的特别文件的指针。