为 bin 目录设置保护

确认 $PATH 下的每一个目录都只有它的拥有者可以写入，其余任何人都不能。同样的道理也应应用于 bin 目录里的所有程序。

写程序前，先想清楚

花点时间想想，你想要做的是什么、该如何实行。不要一开始就在文字编辑器上直接写，而且，在它真的开始运作前，要不断地设法测试。错误与失败的优雅处理，也应该设计在程序里。

应对所有输入参数检查其有效性

如果你期待的是数字，那就验证你拿到的是数字。检查该数字是否在正确的范围内。同样的检查也应该出现在其他类型的数据上，Shell 的模式匹配工具可以将这个工作处理得很好。

对所有可返回错误的命令，检查错误处理代码

不在你预期内的失败情况，很可能是有问题的强迫失败，导致脚本出现不当的行为。例如，如果参数为 NFS 加载磁盘或面向字符的设备文件时，即便是以 root 的身份执行，也可能导致有些命令失败。

不要信任传进来的环境变量

如果它们被接下来的命令（例如 TZ、PATH、IFS 等）使用时，请检查并重设为已知的值。ksh93 会在启动时自动重设 IFS 为它的默认值，无论当时环境为何，但其他许多 Shell 就不会这么做了。无论在什么情况下，最好的方式就是明确地设置 PATH 只包含系统 bin 目录，并设置 IFS 为“空格－定位字符－换行字符”（space-tab-newline）。

从已知的地方开始

在脚本开始时，确切 cd 到已知目录，这么一来，接下来的任何相对路径名称才能指到已知位置。请确认 cd 操作成功：

```
cd app-dir || exit 1
```

在命令上使用完整路径

这么做你才能知道自己使用的是哪个版本，无须理会 $PATH 设置。

使用 syslog(8)保留审计跟踪

记录引用的日期与时间、username 等，参见 *logger*(1)的使用手册。如果你没有 logger，可建立一个函数保留日志文件：

```
logger(){
    printf "%s\n" "$*"  >>  /var/adm/logsysfile
}
logger "Run by user "  $(id -un)  "($USER) at "  $(/bin/date)
```

当使用该输入时，一定将用户输入引用起来

例如："$1"与"$*"，这么做可以防止居心不良的用户输入作超出范围的计算与执行。

勿在用户输入上使用eval

甚至在引用用户输入之后，也不要使用eval将它交给Shell再处理。如果用户读了你的脚本，发现你使用eval，就能很轻松地利用这个脚本进行任何破坏。

引用通配字符展开的结果

你可以将空格、分号、反斜杠等放在文件名里，让棘手的事情交给系统管理员处理。如果管理的脚本未引用文件名参数，此脚本将会造成系统出问题。

检查用户输入是否有meta字符

如果使用eval或$(...)里的输入，请检查是否有像$或`（旧式命令替换）这类的meta字符。

检测你的代码，并小心谨慎阅读它

寻找是否有可被利用的漏洞与错误。把所有坏心眼的想法都考虑进去，小心研究你的代码，试着找出破坏它的方式，再修正你发现的所有问题。

留意竞争条件（race condition）

攻击者是不是可以在你脚本里的任两个命令之间执行任意命令，这对安全性是否有危害？如果是，换个方式处理你的脚本吧！

对符号性连接心存怀疑

在chmod文件或是编辑文件时，检查它是否真的是一个文件，而非连接到某个关键性系统文件的符号性连接（利用[-L *file*]或[-h *file*]检测*file*是否为一符号性连接）。

找其他人重新检查你的程序，看看是否有问题

通常另一双眼睛才能找出原作者在程序设计上陷入的盲点。

尽可能用setgid而不要用setuid

这些术语在本章稍后有探讨。简而言之，使用setgid能将损害范围限制在某个组内。

使用新的用户而不是root

如果你必须使用setuid访问一组文件，请考虑建立一个新的用户，非root的用户做这件事并设置setuid给它。

尽可能限制使用setuid的代码

尽可能让setuid代码减到最少。将它移到一个分开的程序，然后在大型脚本里有需要时才引用它。无论如何，请做好代码防护，好像脚本可以被任何人于任何地方引用那样！

bash维护工程师Chet Ramey提供了下列代码的开场白，给那些需要更多安全性的Shell脚本使用：

```
# Reset IFS. Even though ksh doesn't import IFS from the environment,
# $ENV could set it.  This uses special bash and ksh93 notation,
# not in POSIX.
IFS=$' \t\n'

# Make sure unalias is not a function, since it's a regular built-in.
# unset is a special built-in, so it will be found before functions.
unset -f unalias

# Unset all aliases and quote unalias so it's not alias-expanded.
\unalias -a

# Make sure command is not a function, since it's a regular built-in.
# unset is a special built-in, so it will be found before functions.
unset -f command

# Get a reliable path prefix, handling case where getconf is not
# available.
SYSPATH="$(command -p getconf PATH 2>/dev/null)"
if [[ -z "$SYSPATH" ]]; then
        SYSPATH="/usr/bin:/bin"         # pick your poison
fi
PATH="$SYSPATH:$PATH"
```

这段代码使用了许多非POSIX的扩展，这在14.3节里已说明。

15.2 限制性Shell

限制性Shell（restricted Shell）的设计，是将用户置于严格限制文件写入与移动的环境中，用户多半是使用访客（guest）账号。POSIX并未定义提供限制性Shell的环境，“因为它并未提供历史文件中所暗示的安全性限制”。然而，ksh93与bash两者都提供这一功能，我们将在此介绍它们。

当被引用为rksh时（或使用-r选项）时，ksh93即为限制性Shell。你可以让用户的登录受限制，方法是放置rksh的完整路径名称在用户的/etc/passwd里。ksh93可执行文件必须连接到名为rksh之处，以执行此操作。

限制性ksh93的特定限制不允许用户做下列操作。这些功能有部分是仅ksh93适用，要了解更多信息，可见参考书目中的《Learning the Korn Shell》：

- 变更工作目录：cd是没有作用的。如果你尝试使用它，会收到错误信息ksh: cd: restricted。

- 不允许重定向输出到文件：重定向运算符 >、>|、<>，与 >> 都不被允许。这点不包含 exec 的使用。
- 指定新值给环境变量 ENV、FPATH、PATH 或 SHELL，或试图以 typeset 改变它们的属性。
- 标明任何带有斜杠（/）的命令路径名称。Shell 仅执行在 $PATH 里找到的命令。
- 使用 builtin 命令，增加新的内置命令。

类似于 ksh93 的是：当引用为 rbash 时，bash 即扮演限制性 Shell 的角色，而 bash 可执行文件必须连接到 rbash，以执行此任务。bash 的限制性运算列表和 ksh93 很类似。下面列表里的功能，有部分是 bash 所特有的（参考自 *bash*(1)而来），不过我们不在本书多作介绍。要了解进一步信息，见 *bash*(1)手册页：

- 以 cd 切换目录。
- 设置或解除设置 SHELL、PATH、ENV 或 BASH_ENV 的值。
- 标明含有 / 的命令名称。
- 标明含有 / 的文件名，作为 .（点号）内置命令的一个参数。
- 在内置命令 hash 里使用 -p 选项，指定含有 / 的文件名作为参数。
- 在启动时，自 Shell 环境导出函数定义。
- 在启动时，自 Shell 环境解析 SHELLOPTS 的值。
- 使用 >、>|、<>、>&、&>，与 >> 重定向运算符，重定向输出。
- 使用 exec 内置命令，用另一个命令取代 Shell。
- 以内置命令 enable 搭配 -f 或 -d 选项，增加或删除内置命令。
- 使用 enable 内置命令，启用已停用的 Shell 内置命令。
- 为内置命令 command 标明 -p 选项。
- 使用 set +r 或 set +o restricted 关闭限制性模式。

对这两个 Shell 而言，这些限制都是在用户的 .profile 与环境文件被执行之后才生效。即限制性 Shell 下的用户环境全被设置在 .profile 里。这让系统管理者可以适当地配置环境。

要防止用户覆盖 ~/.profile，只把文件权限设置为用户只读是不够的。根目录不应该被用户写入，或是 ~/.profile 里的命令也不应该 cd 到不同的目录下。

建立这类环境常用的两种方式便是设置“安全”命令的目录，然后让该目录为 PATH 里的唯一一个，以及设置命令选单。其中，用户没有离开 Shell 是不能跳离的。无论如何，请确定 $PATH 下的任何目录中没有其他 Shell；否则，用户只要执行该 Shell，就能避开先前列的限制。同时，也要确认 $PATH 下没有任何程序允许用户起始 Shell，像是来自 ed、ex 或 vi 文本编辑器的“Shell 转义（Shell escape）”。

警告： 虽然自原始的 Version 7 Bourne Shell 起，便拥有限制性 Shell 的功能，但被使用的很少，因为设置一个可用又正确的限制性环境其实并不容易。

15.3 特洛伊木马

特洛伊木马是看起来无害，有时甚至会误以为它很有用，但却隐藏危险的东西。

想想下面这样的情况：用户 John Q.（登录名称为 jprog）是一个顶尖的程序设计师，拥有一些个人程序，就放在~jprog/bin里。这个目录出现在~jprog/.profile里PATH变量的第一个。因为他是这样优秀的程序设计师，不久便被提升为系统管理者。

这对他而言是一个全新的领域，而 John 在不注意的情况下，仍将它的 bin 目录保留予其他用户可以写入。这时有个居心不良的 W.M. 先生，建立了这样的 Shell 脚本，名为 grep，放在 John 的 bin 目录里：

```
/bin/grep "$@"
case $(whoami) in                    检查有效的用户 ID 名称
root)   nasty stuff here             危险的操作就放在这!
        rm ~/jprog/bin/grep          隐匿罪行!
        ;;
esac
```

本质上，当jprog以自己的身份在做事时，这个脚本不会有任何危险。问题出在他使用了 su 命令之后。su 命令可以让一般用户切换到不同的身份。通常用法是：让一般用户成为 root（当然，前提是这个用户必须知道密码）。接下来，su 会使用它继承的任何 PATH 设置（注 2）。在这里的情况是：PATH 包括了 ~jprog/bin。现在，当 jprog 以 root 身份工作，执行 grep 时，确实执行的是他 bin 目录下的特洛伊木马版本。这个版本还是会执行真正的 grep，所以 jprog 仍能得到他要的结果。但更重要的是，接下来脚本还会以 root 身份执行一连串 *nasty stuff here* 处所指定的命令。即该 UNIX 会让脚本为所欲为。当一切操作完成，特洛伊木马也删除，不留任何证据。

注 2： 使用 su - *user* 切换用户，就会像用户登录一般，可防止导入已存在的 PATH。

可写入的bin目录为特洛伊木马敞开了大门，如同在PATH里具有点号，（想想看，要是root执行cd切换到含有特洛伊脚本的目录中，而且点号是在root的PATH里且位置又先于系统目录时，会发生什么事）。让可写入的Shell脚本放在任何bin目录下更是另一个大门。就好像你晚上会关闭并锁上家门一样，你应该确定关上系统上的任何大门。

15.4 为Shell脚本设置setuid：坏主意

UNIX安全性上的问题有很多是出在它的一个文件属性上，称为*setuid*（设置用户ID）位。这是一个特殊权限位：当一个可执行文件将它打开时，身份会立即转换为与文件拥有者相同的一个有效用户ID。这个有效的用户ID与进程真正的用户ID并不同，UNIX以进程的有效用户ID进行权限检测。

假设你编写了一个游戏程序，可保留私有分数记录文件，显示前15名系统里的玩家。你不希望这个分数文件任何人都能写入，因为这么一来任何人只要动点手脚，就能让自己成为高分的玩家。如果让你的游戏setuid为你的用户ID，则只有你自己拥有的游戏程序可以更新文件，其他人都不行（游戏程序可以通过查看它的真实用户ID来知道谁在执行它，并使用它来决定登录名称）。

setuid工具对游戏与分数文件来说是一个不错的功能，如果设为root时，它就可能变得相当危险。将程序setuid为root，可便于管理者处理需要root权限的文件（例如配置打印机）。为了设置文件的setuid位，只要输入chmod u+s *filename*即可。对root拥有的文件设置setuid是很危险的事，所以建议不要在chown root *file*后执行chmod u+s *file*。

类似的工具程序，在组层级上也有，也就是*setgid*（设置组ID）。chmod g+s *filename*即可打开setgid权限。当你执行ls -l时，在setuid与setgid的文件上，会出现s权限模式，取代原有的x。例如-rws--s--x的文件指的便是拥有者可读取与写入、任何人可执行，且setuid与setgid位都已设置（八进制模式为6711）。

现代系统管理的智慧认为，设置setuid与setgid的Shell脚本是一个可怕的想法。尤其在C Shell下更受影响，因为它的.cshrc环境文件有太多可供破坏的地方。而且，它也有很多方式可以将setuid的Shell脚本转化成交互式的Shell，而且是以root的有效用户ID。这就是骇客（cracker）的希望：拥有root执行任何命令的能力。我们从*http://www.faqs.org/faqs/unix-faq/faq/part4/section-7.html*借来一个例子：

> …好，假设有个脚本叫作/etc/setuid_script，一开始是这样：
>
> ```
> #!/bin/sh
> ```
>
> 现在我们来看看假设执行了下面的命令，会发生什么事：

```
$ cd /tmp
$ ln /etc/setuid_script -i
$ PATH=.
$ -i
```

我们知道，最后一个命令将重新安排成：

```
/bin/sh -i
```

因此，此命令会给我们一个交谈模式的 Shell，setuid 为该脚本的拥有者！幸好这个安全性黑洞可以通过，将第一行指定为：

```
#!/bin/sh -
```

解决掉。将 - 置于选项列表的结尾：则下一个参数 -i 将被视为文件名，正常让命令读取之。

正因为如此，POSIX 才容许在 /bin/sh 的选项结尾处使用单一 - 字符。

注意：setuid 的 Shell 脚本与一个 setuid Shell 之间的差异必须特别留意。后者为 Shell 可执行版的副本，它是属于 root 并应用 setuid 位。以上一节的特洛伊木马为例，假定 *nasty stuff here* 部分是这样的：

```
cp /bin/sh ~badguy/bin/myls
chown root ~badguy/bin/myls
chmod u+s  ~badguy/bin/myls
```

还记得，这段代码以 root 身份执行，所以它是可以运作的。当心怀不轨的 badguy 执行了 myls，它是一个机器码的可执行文件，且应用 setuid 位。待 Shell 再回到 root 手中时，系统的安全性便荡然无存了。

事实上，setuid 与 setgid 的 Shell 脚本所带来的危险，在现行 UNIX 系统上都必须特别留意。包括商用 UNIX 系统与自由软件（派生自 BSD 4.4 与 GNU/Linux），都停用了 Shell 脚本上的 setuid 与 setgid 位。即便你在文件里应用这些位，操作系统也不会有任何操作（注 3）。

我们也发现现在的很多系统加载时可选择是否针对整个文件系统停用 setuid/setgid 位。这在网络式加载的文件系统上，还有那些可删除式媒体上，例如软驱与光驱，绝对是件好事。

15.5 ksh93 与特权模式

Korn Shell 特权模式的设计就是为了对付 setuid 的 Shell 脚本。这是一个 set -o 选项

注 3：Mac OS X 与最新的 OpenBSD 版本是我们发现的两个例外，如果你在这类系统下做事，请特别留意！我们发现 Solaris 9 只有在文件拥有者非 root 时，才执行 setuid 的操作。

(set -o privileged或set -p)，无论何时当Shell执行之脚本已设置setuid位时，Shell便会自动输入它；也就是说，当有效用户ID与实际用户ID不同时。

在特权模式下，引用一个setuid的Korn Shell脚本时，Shell会执行/etc/suid_profile文件。此文件应写成限制setuid Shell脚本，一如限制性Shell那样。至少，它会将PATH设为只读（typeset -r PATH或readonly PATH），然后设置它为一到多个“安全的”目录。再说一次，这是用以避开引用时的所有陷阱。

因为特权模式是选用的，所以你也可使用set +o privileged（或set +p）将它关闭。然而，这对潜在的系统骇客产生不了帮助：Shell会自动地将它的有效用户ID切换为相同的真实用户ID。也就是说，当你关闭特权模式时，同时也关闭了setuid。

除特权模式外，ksh另提供了一个特殊的“代理”程序，会执行setuid的Shell脚本（或可执行但不可读取的Shell脚本）。

为此，脚本的开头不应以#! /bin/ksh起始。当程序被引用时，ksh会试图以正规二进制可执行文件的方式执行程序。当操作系统无法执行脚本（因为它不是二进制的，及因为它没有#!标明的解译器名称）时，ksh会认为它是脚本，及使用脚本的名称与它的参数引用/etc/suid_exec。除此之外，它还会安排传递一个认证“token”给/etc/suid_exec，指出脚本的有效用户与组ID。/etc/suid_exec会验证执行脚本是否是安全的，再安排以该脚本的适当真实用户与组ID引用ksh。

虽然结合特权模式与/etc/suid_exec可以避免很多setuid脚本上的攻击，但编写一个可供setuid的安全脚本，其实是一门很大的学问，需要很多的知识与经验，应小心对待。

虽然setuid的Shell脚本在现今系统上不能工作，但有时特权模式也是很好用的。特别是它已广泛应用在第三方所提供的程序sudo上，该程序引用自网页上的说法，允许系统管理者给予特定用户（或一群用户），以root或另一个用户身份执行部分（或所有）命令的能力，其官方网站为：*http://www.courtesan.com/sudo*。系统管理者如要了解执行管理性工作的环境，只要执行sudo /bin/ksh -p即可。

15.6 小结

编写安全的Shell脚本也是保全UNIX系统安全的一环。本章探讨不过是皮毛，我们建议你深入研究UNIX系统安全的相关信息（见“参考书目”）。一开始我们就列出编写安全性Shell脚本的提示，这些都是UNIX安全性领域的专家所认可的。

接下来介绍的是限制性Shell，它可以停用许多具潜在危险的操作，其环境构建于用户的

.profile文件里，该文件会在限制性用户登录时执行。实际上，限制性Shell其实很难正确地设置与使用，我们建议你找其他方式设置限制性环境。

特洛伊木马是看似无害但实际上会对系统产生攻击的程序。我们带你看过几种特洛伊木马的建立方式，但其实还有更多。

设置setuid的Shell脚本不是个好主意，几乎所有近期的UNIX系统都已停用它，因为很难关掉它所打开的安全性漏洞。你必须花时间仔细确认你的系统是否已停用它们，如果没有，请定期查找系统里是否还有这类文件。

最后，我们简短地带过Korn Shell的特权模式，它的目的是在解决诸多与Shell脚本相关的安全性议题。