UNIX Basics

by Peter Collinson, Hillside Systems

CATHY GENDRON

Over Revisited

Back in May, I wrote an article on Kernighan and Pike's `over` script (see "Over," Page 28, `http://sw.expert.com/C2/SE.C2.MAY.00.pdf`). It never seems possible to predict the response to articles, but this one certainly raised some interest, I had quite a bit of email. Thanks to everyone who took the time to write. Of course, some of the email raised further points and I thought that an article that picked up on some of the questions and queries would be of general interest.

Choice of Shell

First, I was asked why I seem to prefer the Bourne Shell (`sh`) for my scripts when most people using Suns (and perhaps other commercial UNIXes) are using the Korn Shell (`ksh`) as a standard command-line interface to the system. Actually, I wrote a history of the development of shells for this magazine in December '97 (see "A Shell Road Map," Page 26, `http://sw.expert.com/C2/SE.C2.DEC.97.pdf`). The article has aged a little, but most of it remains relevant today.

David Korn, the author of `ksh`, was working at Bell Labs in the mid-80s. He took a version of the Bourne Shell source and created `ksh` from it. His idea was always that Bourne Shell users, both humans and extant scripts, should be able to use the new shell with no difficulty. As a result, all Bourne Shell scripts can be run quite happily using `ksh`. There is one exception, the Bourne Shell allows the use of the caret ^ in place of the vertical bar | for pipes. The caret was allowed for backwards compatibility reasons at the time the Bourne Shell was created but is still there, albeit rarely used and deprecated.

Korn added several features that were present in `csh`, like support for job control and the ability to control per-process limits. These features allowed `ksh` users to interface a little better with the underlying operating system, a privilege denied to Bourne Shell users. He took some features from `csh` that were recognized as being sensible for efficiency reasons. The Bourne Shell, for example, uses a separate command `expr` to perform arithmetic; `csh` supports computations in the shell and `ksh` does, too. He also picked up some of the usability aspects of `csh`, like command history and aliasing.

Korn added a bunch of new features: Some were aimed at the terminal user, like command-line editing; some added new functionality to the shell, like the `select` command that generates a menu of values and allows the user to make a selection from the values; and others attempt to reconcile extant shell differences. The `print` command, for example, was introduced to replace `echo` because the UNIX System V `echo` command is different from the BSD version and the divergence causes a great problem with compatibility for some scripts.

A few of the new features were adopted by the Institute of Electrical and Electronics Engineers (IEEE) POSIX com-

mittee and became a requirement for a POSIX-compliant shell. POSIX avoided the features that were aimed at human interaction. The committee liked the use of `$(...)` to replace the back-quote operator because the new syntax allows nesting. To refresh your memory, the backquote operator allows you to capture the standard output from a command in a shell variable, so

```
dirc=`ls *.c`
```

will result in a list of files ending with `.c` to be placed in the `dirc` variable. In a POSIX-compliant shell, this overly simple example is written as:

```
dirc=$(ls *.c)
```

The use of an opening marker `$(` that matches a closing bracket allows nesting.

The committee also adopted the use of shell arithmetic statements wrapped in double brackets:

```
a=100
b=10
c=$((a/b + 10))
echo $c
```

The above statement will print 20. However, these decisions meant that neither of the extant mainstream shells, Bourne or `csh`, could be a POSIX shell.

There was and is a problem with `ksh`. It's not publicly available. I think that Korn tried very hard to get the source out into the public arena and was successful with early versions. However, the source for later versions was proprietary to AT&T Corp. and hidden. This meant that the shell never really gained what might be called widespread acceptance at a time when it mattered. To get widespread usage, scripts that used the new facilities needed to be portable without any let or hindrance, and they just were not. If you look at my December 1997 article, you'll see that Sun Microsystems Inc. was not supporting `ksh` on SunOS at that time, so if you had a mixed Solaris and SunOS site, you would write in Bourne Shell to ensure that your scripts would work everywhere.

Given the absence of the "real" `ksh` in the public area, several clones were attempted. One of these, `pdksh`, came close for interactive use (and I actually contributed some code to this at some point). However, this shell was built on Charles Forsyth's publicly available version of the V7 shell, and he hadn't quite cloned the idiosyncrasies of the Bourne Shell quote syntax exactly.

The GNU Bourne-Again Shell (`bash`) fills the niche for people who are not concerned about the full ramifications of GNU's Public License. In fact, I use `bash` for day-to-day use because it has extra features that I find convenient. Also, some of the features that were clumsy to use in `ksh` are much easier in `bash`, like command and filename completion.

So, given all this, the question remains. Why do I tend to print scripts written in Bourne Shell? Well, I must confess that I've never really bothered to spend time learning the "newer features" of `ksh`. There has never seemed to be mileage in doing so when some of the features are not available on all the platforms I use.

I know that a script that works in Bourne Shell is likely to work everywhere. Some shell clones may not execute it properly, but I rarely tickle the quoting problems that are the main difficulty. I would guess that every UNIX platform has a shell that will support the syntax that I print in this column. The syntax I use should work in several of the shells on your system. The code should work in Bourne Shell, `ksh`, `dtksh` (the version of `ksh` that is supplied with the Common Desktop Environment with support for X widgets) and `bash`, if you have it. I always try to use a shell language that is a known "lowest common denominator."

As a small postscript, and before you mail in to ask, if you are a `csh` or `tcsh` user, and wonder why I never write scripts in `csh`, then you need to read Tom Christiansen's article "Csh Programming Considered Harmful," which you will find reprinted in *UNIX Power Tools* (see below for full reference).

The sort Command

Another piece of mail commented on the fundamental example that I used in the May article. My take on the mail was that the author had felt the need to be somewhat insulting, so I've broken my usual rule and haven't replied to him. Actually, I am more than happy to be told that I have made oversights and mistakes in the articles I write. To err is human, and one learns from mistakes. Sadly, to be offensive in email also seems to be a human trait. As usual, I digress.

The basic idea of the `over` script is that it copes with the

common situation where you wish to run a command on a file and use the same filename to store its output, as in:

```
command data > data
```

This command pattern always fails because of the way that shells work. All shells will open the output file `data` for writing, creating a zero-length file, before invoking `command`. Essentially, before `command` is run on the `data` file, the file contents have already been destroyed.

In my article, I used the `sort` command as my example, and my correspondent pointed out in a somewhat derisory manner that I didn't need to go to the lengths of using `over` to circumvent the problem. I should write:

```
sort -o data data
```

Here, the `-o` flag nominates an output file for `sort`, rather than the data being sent to its standard output channel. An output file that's defined in this way can be the same file as one of the input files, and the command is not allowed to overwrite the input file until all the sorting has been completed. On my Sun and BSD systems, `sort` will create a temporary file on `/tmp` to store the output before overwriting its source file. My correspondent asserted that on AIX, the program will retain information in memory when presented with the `-o` option.

The `-o` option has actually been around for aeons it seems, and was part of the POSIX Shells and Utilities standard (IEEE Standard 1003.2-1992), so I should have known about it.However, I didn't. It shows how you can happily use the things you know for years, making things happen without the need to expand your knowledge. It pays to RTFM.

Temporary Files

The final piece of mail emanated from an educational establishment in the United States. Its author was concerned that the `over` script was creating predictably named files in `/tmp` and these files could be subverted by a malicious user on a multiuser system.

The potential problem is that the script contained the following line

```
new=/tmp/over.$$
```

This sets the `new` variable to `over.` followed by the process ID of the shell process that is executing the script. Later in the script, a command sends its output to this file using

```
... > $new
```

The idea of adding the process ID to the temporary filename is to allow simultaneous use of the script. If more than one person is using the script at any one time, their shells will have different process ID numbers and the files on `/tmp` will have different names that have a good chance of being unique.

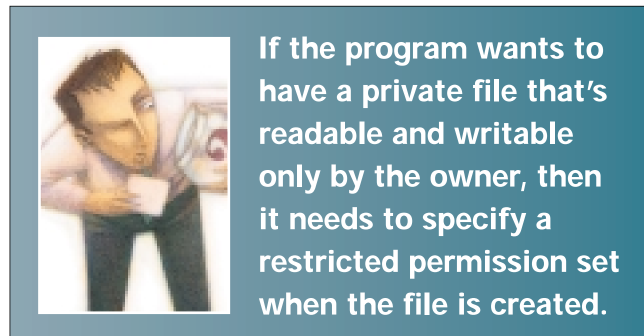
Of course, when one creates such a script, the idea that someone else may pick on these temporary files as a way to damage the system is a long way from your mind. Let's look at `/tmp` and other publicly writable directories and see where the dangers lie.

On the original UNIX systems, `/tmp` was a directory whose permissions were

```
rwxrwxrwx
```

Setting these permissions on the directory means that anyone can create a file in the directory, and anyone can delete a file from the directory. If I create a file in `/tmp`, you can then delete it. The file permissions on the file itself may prevent you from reading or writing my file (see below), but you can still remove it. You can also create a hard link to the file, so that when I delete the file, a copy will remain on `/tmp`.

This situation was altered by Berkeley's 4.4BSD release that defined a new meaning for the "sticky bit" when it is set on a directory. In 4.4BSD, and in Solaris today, the kernel limits access to files that sit in a world-accessible directory that has its sticky bit set. The presence of the sticky bit on the directory prohibits a "mortal" user from deleting (or renaming) a file unless they own it. This means that I can create a file on `/tmp` in the certain knowledge that you cannot delete it.



You perhaps may need an explanation of the term "sticky bit." The standard set of file permissions that I show above occupies nine bits. However, the file mode that's stored with the file and contains the file permissions occupies a 16-bit word. Some of these bits are used to indicate the file type: whether the file is a directory, a regular file, a symbolic link, or a device entry and so on. Two bits are used to store the `setuid` and `setgid` state for executable files, allowing the program that the file contains to be run with the permissions of the owner of the file.

Early UNIX systems used a third additional bit, the sticky bit, to tell the kernel that certain executable files were likely to be in frequent use. The kernel retained the code section of these nominated executable files on the swap space, even when they were not being used. The idea was to permit the process to be loaded quickly when someone invoked them. It was quicker to load the file from swap space, where it was conveniently laid out so that the hardware could pull it into memory using a single direct memory access instruction.

As UNIX moved from swapping to paging, the sticky bit became redundant for executables. Of course, it had no meaning when applied to directories anyway, and was available when

the 4.4BSD team wanted to have some way of getting around the ability for anyone to delete or rename any file on /tmp.

So, the use of the sticky bit sealed one possible security hole. There is actually a danger that the sticky bit is not set on your publicly writable directories like tmp. You can check by

```
$ ls -ld /tmp
drwxrwxrwt   ..............
```

If the "t" is there, then you are OK.

Protecting Files

What about protecting the files themselves? When a program creates a new file, it is required to establish file permissions on that file. If the program wants to have a private file that's readable and writable only by the owner, then it needs to specify a restricted permission set when the file is created.

However, if the file is being created by a general-purpose mechanism such as the > operator in the shell, then the shell will have no idea that the file should be private. In general, the program will create the file so that it's readable and writable by everyone. It will have the following permissions:

```
rw-rw-rw-
```

Ideally we'd like the user to make the decision about who can have access to their files. The umask system call was introduced in UNIX Version 7 to permit the user to control the default permission settings on any new files they create. You need to think of the contents of umask as a binary value; any bit that is set in the mask will remove the corresponding bit from the file permissions that are established on a default file create. Because the file permissions are groups of three binary digits, umask values are usually expressed in octal. For example, a umask value of

```
002
```

(octal 2 is binary 010) will result in files with the following permissions:

```
rw-rw-r--
```

This allows anyone to read the file, and the owner and anyone in the owner's group to write to the file. A umask value of

```
022
```

will result in files with the following permissions:

```
rw-r--r--
```

This removes group write permission. The umask value is passed from parent to child process, so it's normally set in your shell depending on the policy in force at your site. The command

```
umask
```

in most shells will print the current umask value, while

```
umask 022
```

will set it. The two values, 022 and 002 are common settings, but other values are certainly possible. You may wish to remove general access to your files, for example

```
umask 007
```

Be careful not to remove *your* read and write access to files, otherwise odd things may happen.

All this Seems Safe...

Given that file permissions are used sensibly, and the sticky bit is set on /tmp, then things look safe. In some ways, this is *the* problem; things look safe but there still are ways for malicious users to attack your files. If I can predict what filename you will be using on /tmp when you make use of a script, then I can create that file before you arrive. In general, this offers a denial of service attack. If a script does something like:

```
 ... > /tmp/tmpfile
```

Then I can create the named file so that its not writable by you and the script will fail.

A more malicious person can create a symbolic link on /tmp called tmpfile and point it at anything on the system. There are no controls on the contents of any symbolic link that anyone can create. Assuming the person who invokes the script has write access on the target file, then when they run the script, they will unwittingly overwrite the target file. That target file could be something that controls system security like the .rhosts file.

If I, as the script author, add the process ID to the temporary file name, then the problem of creating a suitable symbolic link becomes a little harder. For the over script, I would need to create symbolic links named over.1, over.2 and so on, up to around over.30000. Actually, I suspect that people on a multiuser system would notice if I created 30,000 files in /tmp because most UNIX systems don't handle a huge number of files in one directory very well. So, I'd probably create a small script that hung around waiting for you to log in and then looked at the most recent process ID, and used that information to make a considerably smaller number of symbolic links starting at that known value.

There are two problems at work here. First, symbolic links are perhaps too easy to create. Should I be able to create a symbolic link to your file if I don't have any access permissions for that file? Currently, I can, and pragmatically, if you want to allow me to create a symbolic link to a file that doesn't exist, which you do, then checking on the target's permissions is probably a waste of time. Second, my shell will generally write to a file and overwrite old contents without complaining. You can avoid this by setting the noclobber option in bash, ksh or csh. This stops the shell from overwriting files that already exist. If the script uses the regular Bourne shell, then this

option is not available.

It's necessary to be a little more clever when constructing temporary names so that they will be harder to guess. I note that the `mktemp` C library routine on Solaris applies some function to both the process ID and the time of day to generate a temporary name. The chances of hitting the exact time when I will start a script coupled with the exact process ID it will be using are somewhat more remote.

The other problem with `/tmp` is that it's a shared name space, so programs need to check whether a name they wish to employ is already being used. Traditionally, programs use the `mktemp` routine, which creates a name, checks for its existence on `/tmp` and passes it back to the C program that creates the file. This two-stage operation creates a "race" condition where someone could theoretically "get in first" to create the file.

These problems mean that some UNIX subsystems will prefer to create a temporary directory under the user's home directory rather than use the world-writable shared namespace in `/tmp`. The main disadvantages to using user-specific directories are: `/tmp` is essentially a RAM-disk that runs much faster than magnetic media; `/tmp` is cleared at system bootstrap time and a private `tmp` will slowly be filled with rubbish files.

Well, I started writing this section being a lot more confident in the use of `/tmp` than I'm now. I can't say that I am more than queasy. Most bases seem to be covered. It's somewhat complacent to think that since I am the only user of this machine, then I am safe. There is always the possibility that one day some person will succeed in breaking in and I will not be alone. I don't want my system to offer easy methods for hackers to subvert it.

Further Reading and Thanks

Thanks to Paul B. Henson from California State Polytechnic University for some really helpful email during the course of writing this article. He suggests that you look at `http://www.securityfocus.com` for more details on `/tmp` exploits.

The May 2000 and December 1997 articles are available as PDF files on *SW Expert's* Web site at `http://sw.expert.com`. Also, I maintain a list of all my articles (when I get the time) on my own site at `http://www.hillside.co.uk/articles`. Where possible, this page is linked to online copies.

You'll find Tom Christiansen's `csh` article in *UNIX Power Tools, 2nd Edition*, by Jerry Peek, Tim O'Reilly and Mike Loukides (published by O'Reilly and Associates Inc., 1997, ISBN 1-56592-260-3).

You can get a binary ready-to-run version of `bash` that comes in an easy-to-install package for various Solaris versions from `http://sunfreeware.com`.

Peter Collinson *runs his own UNIX consultancy, dedicated to earning enough money to allow him to pursue his own interests: doing whatever, whenever, wherever... He writes, teaches, consults and programs using Solaris running on an UltraSPARC 10. Email:* `pc@cpg.com`.