COBB

September 1996 • VOL. 2 NO. 9
US $11.50

INSIDE SOLARIS

Tips & techniques for users of SunSoft Solaris™

in this issue

http://www.cobb.com/sun/

Combining find and grep to find any file anywhere

By Al Alexander

Have you ever needed to find a file that contains a particular string? It gets worse when you don't even know exactly in which directory your file resides. Recently, I had to find a file among hundreds of source files in a dozen directories. If you've ever wondered whether you can easily accomplish this under Solaris, the answer to this question is an emphatic "Yes, you can!"

This is the typical situation: You've lost an important file in the maze of files within your home directory structure. You can't remember the filename, but you do remember some of the contents of the file. Specifically, you remember that the words "treasure map" are somewhere inside that file. If only you could find it!

Search files fast with grep

As you probably know, the `grep` utility will do a large part of this job. You can tell `grep` to search a set of files for a particular string that matches a pattern. For example, if we wanted to find all the scripts in the */etc/rc2.d* directory that execute the `ndd` command, we'd use the following commands:

```
$ cd /etc/rc2.d
$ grep ndd *
S69inet: set /dev/tcp
➥tcp_old_urp_interpretation 1
S69inet: set /dev/ip ip_forwarding 1
S69inet: set /dev/ip ip_forwarding 0
```

In this case, we're lucky. The `ndd` command is used only in the *S69inet* script. If the `ndd` command was used frequently in many files, the screen would quickly become unreadable.

We're interested only in the names of the files that contain the string. Luckily, `grep` provides the `-l` option, which tells `grep` to print the filenames containing the string, one per line. If you don't know whether the string is capitalized or not, you can also add the `-i` option to let `grep` ignore case. Now, let's find all the scripts in the */etc/rc2.d* directory that change the path. We'll do so with these commands:

```
$ cd /etc/rc2.d
$ grep -il path *
S47asppp
S72autoinstall
S88sendmail
```

Using `grep`, our search for the "treasure map" becomes simpler. In each directory, all we need to do is execute the command

```
grep -il 'treasure map' *
```

If only you could instruct `grep` to search multiple directories. As it turns out, you can. After you specify the pattern, you can tell `grep` the filenames that you want to search. You can specify pathnames here to instruct `grep` to search multiple directories. For example, if you want to find the names of all the

INSIDE SOLARIS
Tips & techniques for users of SunSoft Solaris

Inside Solaris (ISSN 1081-3314) is published monthly by The Cobb Group.

Prices
U.S. ..... $115/yr ($11.50 each)
Outside U.S. ..... $135/yr ($16.95 each)

Phone and Fax
US toll free ..... (800) 223-8720
UK toll free ..... (0800) 961897
Local ..... (502) 493-3300
Customer Relations fax ..... (502) 491-8050
Editorial Department fax ..... (502) 491-4200
Editor-in-Chief ..... (502) 493-3204

Address
Send your tips, special requests, and other correspondence to:

The Editor, *Inside Solaris*
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Internet: inside_solaris@merlin.cobb.zd.com.

For subscriptions, fulfillment questions, and requests for group subscriptions, address your letters to:

Customer Relations
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220
Internet: cr@merlin.cobb.zd.com.

Staff
Editor-in-Chief ..... Marco C. Mason
Contributing Editor ..... Al Alexander
Production Artist ..... Liz Palmer
Editors ..... Linda Recktenwald
Karen S. Shields
Circulation Manager ..... Mike Schroeder
Editorial Director ..... Linda Baughman
VP/Publisher ..... Lou Armstrong
President ..... John Jenkins

Back Issues
To order back issues, call Customer Relations at (800) 223-8720. Back issues cost $11.50 each, $16.95 outside the US. We accept MasterCard, Visa, or American Express, or we can bill you.

Advertising
For information about advertising in Cobb Group journals, contact Tracee Bell Troutt at (800) 223-8720, ext. 430.

Postmaster
Second class postage paid in Louisville, KY.
Postmaster: Send address changes to:

Inside Solaris
P.O. Box 35160
Louisville, KY 40232

scripts that hold the string `cat` in the */etc/rc0.d*, */etc/rc1.d*, and */etc/rc2.d* directories, you could type

```
grep -il cat /etc/rc0.d/* /etc/
➥rc1.d/* /etc/rc2.d/*
```

As you can see, this is an unwieldy technique when you have many directories and subdirectories to specify.

Adding more power with find

When you want to find a file but don't know where it is, the `find` command is the tool of choice. It will search a directory tree for files based on their names, ages, or other attributes. It turns out that `find` offers a unique and underused option called `-exec` that, in combination with the `grep` command, gives you all the power you need to locate your file.

The `-exec` option of the `find` command will tell `find` to execute a command on each file it finds. Thus, we can use the `find` command to find all the files in the directories we care about, then instruct `grep` to search through these files. If we want to search every file on the hard disk for the phrase "treasure map," we can do so like this:

```
find / -type f -exec grep -li
➥'treasure map' {} \;
```

The `find` command begins at the root and finds all files that *are* files (i.e., not directories, soft links, character devices, etc.). Then, for each file found, it executes the `grep` command to search for the phrase "treasure map." If the phrase is found, `grep` prints the name of the file.

The unusual syntax after the "treasure map" string is part of the required syntax for the `-exec` option. The escaped semicolon (`\;`) signals the end of the `-exec` clause. The curly braces are a placeholder for each file that `find` is currently considering. The brackets hold each filename, one at a time, while `grep` searches through that file. You can then visualize the command as running like this:

```
grep -il 'treasure map' file1
grep -il 'treasure map' file2
        .
        .
        .
grep -il 'treasure map' fileN
```

Given my notoriously bad memory, I use this command often. For instance, recently I was looking for all of the files on my system that contained the `ifconfig` command. I couldn't remember where I had made some network modifications, but I knew my changes involved the `ifconfig` (interface configuration) command. All I needed to type to find the list of candidate files was

```
find / -type f -exec grep -il
➥ifconfig {} \;
```

Once I found the files containing the `ifconfig` command, finding the correct one was simple.

Other -exec applications

In addition to searching for text strings within files, the `-exec` option opens up many other possibilities. Consider for a moment that you can now run any Solaris command on every file that matches your search criteria, and you'll see that you can come up with some powerful combinations. For instance, if you're short on disk space, how about compressing every file in the home directory that hasn't been accessed in over 90 days. You can do so by typing

```
find /home -type f -atime +90 -exec
➥compress {} \;
```

Another example might be to find all of the core dumps on the system and remove them as follows:

```
find / -name core -exec rm {} \;
```

Or, if you're concerned about system security and setuid files owned by root, you can generate a long listing of all of these files by typing the following command:

```
find / -user root -perm -4000 -exec ls -ld {} \;
```

Of course, in addition to built-in Solaris commands, you can also run your own programs after the `-exec` option, including shell and Perl scripts. You'll probably want to read the `man` page for `find` to figure out other uses for the `-exec` option. Combining multiple search parameters and the `-exec` option gives you a powerful tool for your toolbox.

Conclusion

We've shown how to combine the `grep` and `find` commands with `find`'s `-exec` option to search for text strings within files throughout directory trees. This is a useful way to locate files when you know part of the file contents, but not necessarily the filename.

More importantly, we've also shown you other general applications of the `-exec` option that provide extensibility to the `find` command. You now have the power to say "find every file in the file system that matches my search criteria and run my command on only those files." This a powerful tool for all of your file-oriented operations. ❖

TAR TIPS

Working with archives

If you're just starting with `tar`, reading the `man` page gives you a good idea how it works and how you can use it. However, the `man` page doesn't describe some of the conventions that experienced system administrators use for their `tar` archives. In this article, we'll show you two important rules that you should use when you create archives with `tar`.

One archive, one directory

The most common mistake beginners make with `tar` is to create an inconvenient archive. Most `tar` archives, when you extract them, create a directory with the same name as the `tar` file, but without the *.tar* extension. All the files are placed in this directory. This is the expected behavior of a `tar` archive.

The most inconvenient type of `tar` archive is one that doesn't create the subdirectory and instead places all the files in your current directory. This can fill a neatly crafted directory tree with lots of undesired undergrowth.

For example, on many Solaris installations, the home directories for new users are often placed in a specific location, like */export/home*. Thus, your */export/home* directory may look something like this:

```
$ ls /export/home
marco   linda-r liz     alvin   linda-b bin
misc    progs   martha  jeff    august  amy
```

Suppose Liz decides to leave the company. You might save everything in a `tar` archive just in case someone needs a file that Liz created. If you're inexperienced with `tar` archives, you might create the `tar` archive like this:

```
$ cd /export/home/liz
$ tar cvf /archive/liz *
a letter.1 1K
a letter.2 4K
a letter.3 1K
a data/ 0K
a data/addresses 1K
```

Here, you're inside Liz's home directory, and you're saving all the files in it to the file */archive/liz*. The problem comes into play when you need to reinstate Liz's directory. You can do so by typing

```
$ mkdir /export/home/liz
$ cd /export/home/liz
$ tar xvf liz
tar: blocksize = 18
x letter.1, 893 bytes, 2 tape blocks
x letter.2, 3584 bytes, 7 tape blocks
x letter.3, 219 bytes, 1 tape blocks
x data/, 0 bytes, 0 tape blocks
x data/addresses, 773 bytes, 3 tape blocks
```

However, experienced UNIX users expect the `tar` file to contain a directory, with

files and subdirectories beneath it. An experienced UNIX system administrator who wanted to restore Liz's directory would use the following commands:

```
$ cd /export/home
$ tar xvf /archive/liz
tar: blocksize = 18
x letter.1, 893 bytes, 2 tape blocks
...
```

At this point, the system administrator will start cursing, because the */export/home* directory is now being loaded with files and subdirectories. You should create a `tar` archive that other people may possibly use, like this:

```
$ cd /export/home
$ tar cvf /archive/liz liz
```

This way, when you expand the archive in the expected way, `tar` will create a directory named *liz* to hold all the files. None of the files in the archive will mix with files in the current directory or in other directories.

This is important to system administrators because you usually want to restore an archive only for a short period of time. Once you're finished with the data, you may want to remove the data from your disk to save space.

When an archive drops files in your current directory, removing them can be difficult if there are already files in the current directory. And the difficulty is compounded if there are multiple subdirectories. How will the system administrator know which files to remove? Luckily, if this happens, you can use a nifty trick to remove the files: See the sidebar "Removing an Inconvenient Archive" following this article.

Consistent naming

Another way you can avoid surprising behavior is to give your `tar` file the same name as the directory that you're copying. Thus, your archive should be named *liz* if you're archiving a directory named *liz*. This way, other system administrators automatically know what directory name `tar` will create when it extracts the files from the archive.

Conclusion

Many people have to learn these conventions the hard way. But you don't have to. If you use these two simple rules for all your `tar` archives, you'll make life simpler—both for you and your successor. ❖

Removing an inconvenient archive

What happens if you get an archive in an inconvenient format that puts files all over the current directory and creates multiple subdirectories? It turns out that you can use this easy trick to get rid of all these files. Simply execute the command line

```
$ tar tf filename.tar | xargs rm -Rf
```

where *filename.tar* is the name of the archive containing all the files and directories you want to remove. This tells `tar` to print the list of files and subdirectories found in the file archive and send them to the `xargs` command, which executes the `rm -Rf` command for each name in the archive. This way, if you accidentally get a `tar` archive that dumps junk all over your file system, you can quickly and easily remove it from your hard drive.

The only disadvantage of this technique is that filenames with strange characters in them can cause problems. The typical example of a strange character in a filename is a space. This causes the `rm` command to interpret the filename as two files. Thus, not only will the file still exist after the operation, but if you have a file with the same name as one part of the filename, then `rm` will remove it.

As an example, let's assume that you have a subdirectory named *data*, and your archive file has a file named *data files*. After you use this trick, you'll lose your *data* subdirectory, and be left with the file *data files*.

If you're going to use this trick and would like to be careful, first execute the `tar tf` *filename.tar* command by itself. You can then examine the list of files and directories for names that may cause problems. If you find some, here's a simple workaround to this problem. Type the command

```
tar tf filename.tar > temp
```

Now you have a file named *temp* that contains all the files. Delete any that contain spaces or other problem characters. Then delete the remaining files in the archive via the command

```
xargs rm -Rf <temp
```

You can now delete the remaining problem files, as well as the *temp* file.

How do you erase a file so it can't be recovered?

In a secure environment, it's often important to be able to erase a sensitive file without a trace. You don't want someone else to view the contents of the file. How do you erase this file so that it can't be recovered?

What does the rm command do?

When you use the `rm` command, Solaris simply deletes the directory entry for the file. If it's the last directory entry pointing to the file, then it erases the inode for the file as well. However, the data remains on your hard drive. In this state, it's very difficult to recover the data. Though difficult, it's not impossible to read much of this information. (You may want to read the article "Hard and Soft File Links" in the July issue for more information about how Solaris stores disk files.)

Destroy the data before you destroy the file

If you want to erase a file so that there's no chance of it being recovered by someone else, you need to destroy its contents before releasing it. The simplest way to do this is to zero the file before releasing it. You can do this with the following command:

```
$ dd if=/dev/zero of=fileName count=fileSize
➥bs=1
```

This tells Solaris to read *fileSize* bytes from */dev/zero* and write them to *fileName*. The file */dev/zero* is a special file that always returns zero for each byte read. Thus, this command copies zeros on top of your sensitive data, destroying it. Then you're free to `rm` the file, knowing that the file can't be recovered.

When you use this technique, you must be sure that someone else doesn't have a link to the file. To do so, you should examine the output of `ls` to see how many other places the file is referenced. Otherwise, you'll zero the file, and the other references will now point to a file containing only zeros. Similarly, you want to be sure that you use the correct value for *fileSize*, or you may leave some of the confidential data intact.

In order to make things simpler, Figure A shows a simple shell script that will do all the work for you. It will warn you about requests to wipe a directory or a file that's linked to other locations. If the file passes these tests, the *wipefile* script will destroy the data in the file and unlink it from the directory.

Let's examine the highlighted lines to see what the *wipefile* script does. The first three blue lines of code find the size of the file in

Figure A

```
#! /bin/ksh

usage()
{
echo "wipefile <filelist> - destroy the data in the"
echo "files <filelist> and then remove them."
echo "Note: Will not remove a directory."
echo "Note: Will not destroy/remove a file that has"
echo "      additional links."
exit
}

# If invoked with no arguments, show usage
if [ $# -lt 1 ]; then usage; fi

# For each filename on the command line, wipe the file
for i in $*; do

   # Fail if the specified file is a directory
   if [ -d $i ]; then
      echo "$i: directory: Not removed."
      continue
   fi

   # Find # links to file, size of file
   LINKS='ls -al $i | awk '{ print $2,"x",$5 }''
   FILE_SIZE='expr ${LINKS#*x } / 512'
   LINKS='expr ${LINKS% x*}'

   # Warn user that the file has extra links
   if [ $LINKS -gt 1 ]; then
      echo "$i: additional links: Not removed."
      continue
   fi

   # File isn't a directory, or multiply-linked,
   # so destroy it.
   echo "$i: Clearing & deleting file"
   dd if=/dev/zero of=$i count="$FILE_SIZE"
      ➥bs=512 >/dev/null 2>&1
   rm $i

done
```

This script will destroy, then remove, a file so it cannot be recovered.

blocks (i.e., 512-byte increments) and the number of links to it. We don't want to destroy a file that's in use in another location, so if it has more than one link, we won't destroy the file.

The final blue lines do all the work. We use the `dd` command to copy the */dev/zero* file on top of the file we want to destroy, then we remove the file. The */dev/zero* file is special: It acts as if it were an infinitely long file containing only zeros. Thus, when you read it, you'll always get zeros and never encounter the end of the file.

The `dd` command in the script reads `FILE_SIZE` blocks from */dev/zero* and writes them to the file we want to destroy. We discard the standard output and standard error output of the `dd` command, so it doesn't pollute the output of our script. After we've overwritten the file, we remove it with the `rm` command.

Other security hazards

Note that this script has a shortcoming. While it does everything that it's programmed to, it doesn't totally remove the risk that someone else will be able to recover data from your files. You must be aware that some text editors and word processors leave backup files on your system.

Even worse, many operations may create temporary files that hold parts of your data files. Thus, even when you use *wipefile* to destroy files, there's the chance that the partial remains of your confidential files exist somewhere on your disk drive.

When you allocate disk storage, Solaris doesn't zero it. Thus, it's possible to write a program that allocates disk storage and then looks for "interesting" information inside it. If you use the *wipefile* script, you'll lessen the likelihood that nefarious users will be able to find your information. Keep in mind, however, that some of your data may linger on the swap area or on some unused disk area due to updates or temporary files.

If you're particularly worried about security of your data, you may want to clear all the unused space on your file system that contains your sensitive files. You can do this by using the `mkfile` command to allocate all the free space on the file system containing your sensitive files, then use the *wipefile* script to destroy all the data on it.

For example, suppose your sensitive files are on a separate file system mounted at */conf*. If you want to clear all the free blocks, you can do so by first finding out how much space is free on the file system, as shown in Figure B.

In Figure B, you can see that the */conf* file system has 97243 blocks free. We'll tell `mkfile` to create a file 97243 blocks long named */conf/filler*, and then destroy it like this:

```
$ mkfile -v 97243b /conf/filler
/conf/filler  49788416 bytes
/conf/filler: No space left on device
$ ./wipefile /conf/filler
/conf/filler: Clearing & deleting file
```

Note that this process can take a long time, depending on the size of the */conf* file system. Since this process will take a long time and involve a lot of disk I/O, you'll probably want to schedule it when no one else is using the system.

Better security

Since Solaris doesn't zero disk blocks when they're allocated, it's possible for a clever hacker to find your confidential data. You can make it harder for a hacker to steal your data by keeping all your confidential data on a special disk partition and restricting access to only those who need it.

You may also need to reconfigure the tools that access the confidential data to prevent them from leaving data around in deleted temporary files. However, some tools may not allow you to specify where they place temporary files, so you may have to restrict access to those tools on your most sensitive data.

The best method of security is to isolate the entire machine from hackers. Place your machine behind a good firewall and only allow access to the users who work with the data. ❖

Figure B

```
$ df
/       (/dev/dsk/c0t0d0s0 ): 684710 blocks 199944 files
/usr    (/dev/dsk/c0t0d0s6 ):  86664 blocks  94549 files
/proc   (/proc             ):      0 blocks    447 files
/dev/fd (fd                ):      0 blocks      0 files
/var    (/dev/dsk/c0t0d0s3 ): 145390 blocks  39150 files
/export (/dev/dsk/c0t1d0s7 ): 737400 blocks 244401 files
/tmp    (swap              ): 122744 blocks   5704 files
/conf   (/dev/dsk/c0t1d0s2 ):  97243 blocks    611 files
```

The `df` command can show you how many blocks are available on the file systems holding your confidential data.

Trimming your log files

Once your Solaris system is stable and working, you can almost ignore it. However, if you ignore it completely, you'll run into problems. You need to check on your system periodically.

One thing that you really need to watch is disk space. If you run out of disk space, your system will fail. As Solaris runs, it logs various information for future use. Solaris stores these log files in various directories in the */var* branch of your file system. The longer your system runs, the bigger these log files get.

One pair of log files, *wtmp* and *wtmpx*, keeps track of who has logged into the system and for how long. If you need to see who has logged into your computer over the past few days, you can run the `last` command to find out, as shown in Figure A.

However, Solaris never clears the *wtmp* and *wtmpx* files. Thus, as the system runs, the log files increase in size. Eventually you need to clear them out to ensure that the system can continue to operate.

If you just delete these files, the system won't be able to perform any logging. The login program expects these files to exist and won't create them. In addition, the *wtmp* and *wtmpx* files need to have a particular owner and set of permissions or the login program can't update them. This is why you can't just `rm` the files and create new ones.

One way you might clear the log files is to create a new empty file with the correct permissions and owner. Then you could delete *wtmp* and *wtmpx*, and then copy this file to *wtmp* and *wtmpx*. However, when you copy a file on top of one that already exists, the overwritten file keeps its permissions, owner, and group. Thus, a simpler way to trim the log files is to copy the */dev/null* file on top of the files you want to clear. The easiest way to clear the *wtmp* and *wtmpx* log files is like this:

```
$ cp /dev/null /var/adm/wtmp
$ cp /dev/null /var/adm/wtmpx
```

When you do this, you erase all the log information about the users who've logged into your computer. Therefore, you must first make sure that you don't need this information or that you have a backup copy somewhere.

Figure A

```
$ last
marco    console      Wed Jul 31 13:23   still logged in
marco    console      Tue Jul 30 13:13 - 13:23 (1+00:10)
marco    console      Tue Jul 30 13:07 - 13:12  (00:05)
reboot   system boot  Tue Jul 30 12:49
marco    console      Tue Jul 30 11:52 - 12:47  (00:54)
reboot   system boot  Tue Jul 30 11:51
marco    console      Tue Jul 30 11:14 - 11:15  (00:00)
root     console      Tue Jul 30 09:45 - 11:14  (01:29)
reboot   system boot  Tue Jul 30 09:32

wtmp begins Wed Jul 30 09:32
```

The `last` command shows you who has logged in to the computer and for how long.

Not all log files require that the owner and permissions remain the same. Some programs, if you delete their log files, will simply create new ones, but some won't. However, if you clear your log files by copying */dev/null* over them, you won't have to worry about which log files require special treatment.

You should browse around the */var* hierarchy to find other log files your system uses. When you do so, you'll see that some files grow faster than others. For example, */var/adm/messages* grows each time you boot your system—this file keeps track of all the messages your system displays while booting. The aforementioned */var/adm/wtmp* file grows as people log in. And the more you use scheduled jobs on your system, the faster */var/cron/log* grows. Once you're aware of which log files are on your system and what they log, you can decide how often to clear them. ❖

We'd love to hear from you

If you've come across an interesting Solaris tip, have questions about articles you've seen in *Inside Solaris*, or have ideas for topics you'd like us to cover in future issues, you can send mail to

Editor-in-Chief, Inside Solaris
The Cobb Group
9420 Bunsen Parkway, Suite 300
Louisville, KY 40220

Or you can reach us via the Internet at *inside_solaris@merlin.cobb.zd.com*.

Creating a backdoor entrance to the root account

By Al Alexander

In this article, I want to explore an idea I've seen implemented at several different sites. The idea is to create a backdoor entrance to the root account. You can use this backdoor in the event of a severe system emergency that otherwise would disable access to the root account, such as a corrupt or missing password file. You can also use it to give other users root privilege so they can assist in your administration duties.

While these may be highly desirable features, this procedure has the side effect of creating a potentially serious security loophole. Backdoor access to the root account can allow savvy Solaris users to break into your system. If you're connected to the Internet or other WANs, you don't want to pursue this method. In this article, we'll discuss the process required to implement this solution, and we'll also examine the pros and cons of the approach.

Discussion

I've seen two reasons why people are willing to create such a dangerous security loophole. First, they've been in a situation where they couldn't log in as the root user and ran into major problems. Perhaps they went on a long vacation and forgot the password, or maybe the password or shadow file became corrupt. In either circumstance, normal root access via a login or `su` process won't work properly.

I've also seen sites where system administrators have created a backdoor program to allow other users to assist in system management. For example, let's say you've created the world's best printer management shell program. Regardless of how good your shell program is, the person using this program must have root permission to be able to cancel another person's print jobs. This situation ordinarily presents a significant problem.

However, if you create a printer management program named `PrinterMgmt.sh` and a backdoor program named `RunAsRoot`, other users can help manage the printer queues by typing

```
RunAsRoot PrinterMgmt.sh
```

Your `RunAsRoot` program can run any command with the root user permissions. In this case, the `PrinterMgmt.sh` program is run with the necessary root permissions.

While this approach can be very valuable, the problem is that people can also type

```
RunAsRoot ksh
```

to run a Korn shell with root permissions! This allows them to do anything they want as root.

Despite these significant security concerns, many sites create this backdoor access because for them the pros outweigh the cons. Maybe they aren't attached to any external networks, and they trust their employees beyond question. Or maybe they take the chance of creating such a program and not telling anyone about it. For instance, if such a program existed on your machine, would you be able to find it? (If not, don't worry, I didn't either for a long time; I'll show you how to do it shortly.) If you do know how to find this type of file, do you look for it on a regular basis?

Implementation

Implementing this solution involves the following steps:

1. Create a C program that executes `setuid(0)` and the desired program.
2. Set the SUID bit on the program.

I'll assume that you are already logged in as the root user to implement this solution. Specifically, you'll need to be logged in as root to set the SUID bit on the program and give it proper ownership.

Figure A shows a rudimentary C language program that will execute programs as if they were executed by the root user. Please note that this program does no error-checking and makes no attempt at disguising its purpose. It simply invokes the `setuid(0)` system call to change your user ID to 0 (i.e., root), then runs the program name that you type at the command line, using the `system()` function call. While you can make this program far more elaborate, this is the minimum required. If this

file is named *emergency.c*, I would compile it like this

```
cc -o emergency emergency.c
```

to create an executable program named `emergency`.

The next step is necessary but often overlooked. You must set the SUID bit on the file using the `chmod` command. This special setting tells the operating system that it's okay to execute the `setuid()` function call. In this case, the proper command to issue is

```
chmod 4755 emergency
```

This command sets the permissions to 755 and also sets the SUID permissions bit. If you look at this file with the `ls -l` command, you'll see output similar to the following:

```
$  ls -l emergency
-rwsr-xr-x   1 root     sys        6636 Jul  8
➥22:44 emergency
```

The s character (the fourth character in the permissions column) indicates that the SUID bit has been set on this file. Again, this means that when a user runs this file, he or she will run this command with the effective user ID of the root user.

Now you've done all that is necessary to open a backdoor to the root account. The final step in the process involves trying to hide this backdoor somewhere in your file system. Be aware, however, that no matter where you hide it, an intelligent user will be able to find this backdoor program by issuing the following command:

```
find / -perm -4000 -print
```

You can read this command as: "Find all files with the setuid bit set." The hyphen before the 4000 tells `find` to treat the 0s as wildcard characters, similar to the meaning of the question mark in a filename specification. Actual file permissions can be 4755, 4544, or anything else beginning with a 4; that's all this `find` command looks for, which is one of the reasons this approach creates such a big security loophole. Any user who stumbles onto this file can essentially gain access to the root user permissions.

Making it more secure

If you're a good C language programmer, you can come up with other programming methods to prevent people from easily gaining access to the root user's privileges. These methods include embedding passwords into the program and forcing the use of unadvertised command-line options, such as `-r` or `-c` to actually run a command. If the user doesn't include these options, you simply terminate the program without running `setuid(0)`.

In this scenario, if a user types

```
RunAsRoot ksh
```

the program does nothing but return the user to the prompt. However, if the user runs the command as

```
RunAsRoot -r ksh
```

the program succeeds and sets the user ID to root.

Another way to improve the security of this program is to modify it to allow the user to run only certain programs. Thus, you could make the program execute only your `PrinterMgt.sh` program, for example. However, if you do so, keep in mind that users could

Figure A

```
/*  RunAsRoot.c  */

#include <stdio.h>
#include <string.h>
#define BUFFER_SIZE 8192
int main(int argc, char *argv[])
    {
    int     i;
    char    command[BUFFER_SIZE+1];

    setuid(0);

    command[0]='\0';

    for(i=1; i<argc; i++)
        {
        strcat(command,argv[i]);
        strcat(command," ");
        }

    exit(system(command));
    }
```

This program can run a program specified on the command line as root because it sets the user ID to root.

supply their own shell scripts with the desired name that does what *they* want to accomplish as root. One way around this is to reset the path in the `RunAsRoot` program to a known path that users can't change. You could also specify each program by its complete path.

These improvements won't make the `RunAsRoot` program totally secure. By its very nature, it's an insecure program. However, in a trusted environment, it can be a valuable tool, because it can allow some people to perform simple administrative tasks without taking your time.

Conclusion

If you are working in a trusted environment, this method of creating backdoor access can provide emergency access to the root account and can let you have users run your other custom programs (shell, C, Perl, or other) with root permissions. There are certainly times when having additional system administration assistance is helpful! ❖

Alvin J. Alexander is an independent consultant specializing in UNIX and the Internet.

KORN SHELL SCRIPT TIP

String extraction in the Korn shell

When writing shell scripts, you need a large toolbox of tricks to help you get jobs done. Sometimes, you have to select the shell in which to write your script based on the tools you have in each shell's toolbox.

One task that you often need to do in a shell script is to take apart a string. In this article, we'll describe the Korn shell's pattern-matching operators and demonstrate how you can use them in a script to break strings apart.

The pattern-matching operators

The Korn shell provides four pattern-matching operators you can use to take apart string values: `%`, `%%`, `#`, and `##`. You use all these operators in the form

```
${varXpattern}
```

where *var* is the variable name, X is the operator, and *pattern* is the pattern to match. Table A describes the meaning of the operators.

Now let's see these operators in use. The following shell script, named `banana`, shows how these operators will work in your scripts:

```
#! /bin/ksh

TEST=banana
echo TEST=$TEST
echo \${TEST%n*}=${TEST%n*}
echo \${TEST%%n*}=${TEST%%n*}
echo \${TEST#*n}=${TEST#*n}
echo \${TEST##*n}=${TEST##*n}
```

Now run the script as follows. When you do so, it demonstrates how easily you can break a string apart with the pattern-matching operators.

```
$ ./banana
TEST=banana
${TEST%n*}=bana
${TEST%%n*}=ba
${TEST#*n}=ana
${TEST##*n}=a
```

Table A

Operator	Description
%	Remove shortest pattern match from rightmost part of variable.
%%	Remove longest pattern match from rightmost part of variable.
#	Remove shortest pattern match from leftmost part of variable.
##	Remove longest pattern match from leftmost part of variable.

You can use these four operators to disassemble shell variables in your shell scripts.

In our example, we use *banana* as the string we want to take apart. By using the `%` operator with the pattern *n*`*`, we get *bana* as a result. The *n*`*` pattern matches the letter *n* followed by any number of other characters. Since the `%` operator takes the shortest possible match from the right of the string, it removes the rightmost *n* with any characters following it. The `%%` takes the longest possible match, so it takes the leftmost *n* with all the characters following it, so we get *ba* as the result.

Specifying a pattern

You can specify a pattern in the same way that you do elsewhere in Solaris. You can use numbers and letters to represent themselves, `?` to represent any single character, and `*` to represent any combination of zero or more characters.

Thus, if you want to specify a pattern for the letter *n* followed by any character followed by another *n*, you can use the pattern *n*`?`*n*. Similarly, if you want to find any string that ends with *ing*, you can use the pattern `*`*ing*. You can find more information about specifying patterns in the `man` pages for `grep` and `regex`.

General usage

These operators are very handy when you're in a situation where you need to remove a suffix or prefix from a string. These situations appear very often. You may, for example, want to break a filename into a path and base filename. To do so, you can get the path by removing all the characters to the right of the last /, and you can get the filename by removing the last / and all characters preceding it, like this:

```
PATH=${FNAME%/*}
NAME=${FNAME##*/}
```

Conclusion

If you need the ability to remove a prefix or suffix from a string, you may want to use the Korn shell for your script. The pattern-matching operators provide a simple way to break apart your strings. ❖

DISK MANAGEMENT

No matter how you slice it...

One topic that Solaris users frequently debate is the proper way to slice up a hard drive. Some users prefer to carefully slice their hard drive into many pieces to allow detailed storage management. Other users don't slice the hard drive at all. Many of us take the middle ground.

None of these positions is wrong—each allows you to select one set of tradeoffs. In this article, we'll present some of the tradeoffs you'll face when you slice your next hard drive, so you can make the most suitable decision.

A single-slice system

A single-slice system contains a single file system on a single slice. This makes disk management simple, in that there's none to do. For development systems, very simple systems, or very lightly used systems, this isn't a bad choice. It allows you to concentrate your efforts on other things.

You don't need to worry about rearranging slices when part of your file system fills up. Similarly, you don't have to worry about where to install a new software package. You simply determine whether you have enough space and install it if you do.

The disadvantages of a single-slice system are that you have limited control over the file system. If you enable disk quotas, you enable them for the entire file system. If you need to run `fsck` to check the file system, you'll need to reboot your computer.

Few computers have only one slice. Many have two, however, using one for swap space and the other for everything else. Please note that you'll have to start considering disk space management once you add a second hard drive to a system. Even though you don't have to partition your drives into many pieces, you'll still have to decide where to put the additional storage in your file system hierarchy.

Multiple slices

Managing a system with multiple disk slices is more difficult, but it offers more benefits. Splitting a file system into multiple slices gives you finer control over all aspects of disk

management. For example, you can help protect your mission-critical applications, add disk quotas to some file systems, and simplify backing up and restoring. Let's take a brief look at some of the costs and benefits of using multiple slices.

Mission-critical applications

If you have several mission-critical applications running, you can put their data areas on their own slices. Thus, if another job starts consuming prodigious amounts of storage on another disk slice, it won't prevent the mission-critical application from getting the disk space it needs.

You need to keep in mind that a mission-critical application may require space in other file systems as well. If your application does so, you need to be sure that no runaway applications fill that file system. As an example, your application may use temporary files at */tmp*. For the best protection of your mission-critical applications, you may want to force your mission-critical applications to put their temporary files on their own file system.

Disk quotas

You can also ensure that users don't use too much disk space by enforcing quotas on certain file systems. This way, one user won't be able to lock out other users by consuming all available disk space. It also prevents problems when a process goes awry and starts writing vast streams of data to the disk.

Another way to manage disk space is to put groups of users or departments on their own slices. This technique doesn't even require you to turn on the disk quota system.

Ease of backup

When your file systems are partitioned into slices, you can simply back up only selected parts of the file system at any given time. For example, you could back up only your applications file systems after each installation. You could back up frequently changing or critical data nightly, without consuming the additional time or tape backing up your applications would require.

Restoring a particular file is also easier, since each backup set is smaller. Thus, if a user loses a file, the time needed to find and restore the file in your backup set will be shorter.

Read-only partitions

Another advantage of a multiple-slice system is that you can mount some of the file systems as read-only, allowing you to store applications where they won't accidentally be damaged. Mounting slices as read-only also provides an impediment to hackers. Before they can change any files on the specified file system, they must first get the root password so they can unmount the partition and remount it as read-write.

File system maintenance

Splitting a file system into multiple pieces allows you to perform some basic file system maintenance functions without bringing down the entire system. For example, you can run `fsck` on a single file system, other than root, by simply unmounting it and running `fsck`.

Disk space management

One of the disadvantages of a multiple-slice system is that occasionally a slice will fill up and need to be expanded. You can do so in several ways. One way is to back up the affected slice, repartition your drive, and then restore the data.

You may be able to avoid the backup/restore route. If you have disk space available, you can create a new slice and mount it at a temporary location, then copy all the data from the old slice to the new slice. Then you unmount the new file system from the temporary mount point and the old file system from its current mount point. Finally, you mount the new file system at the old location.

Alternatively, you could simply move some of the data by moving a directory to another slice. To do so, create the directory on the new slice and copy the data to it. Then delete the directory on the old slice, and create a symbolic link to it to point to the new slice. For example, suppose you want to move the directory */abc/def* to */xyz/def*. To do so, you can follow these steps:

```
$ cp -r /abc/def /xyz
$ rm -r /abc/def
$ ln -s /xyz/def /abc/def
```

Here we're telling `cp` to recursively copy the contents of the */abc/def* directory to */xyz/def*. If you omit the `-r`, then `cp` won't copy any subdirectories. The second step removes the */abc/def* directory and all its subdirectories. In the third step, the `ln` command links the newly created */xyz/def* directory to */abc/def*. This allows your scripts and programs that expect to see directory */abc/def* to continue to work.

The biggest disadvantage to a multiple-slice system comes when you want to install a substantial application that won't fit in any of

the file systems. It's especially annoying when the aggregate free space on all your file systems is more than adequate to hold the package.

How big should your slices be?

Deciding how big each of your disk slices should be is a complicated issue. If you're just setting up a machine, it can be harder still. How many applications are you going to install? How many users are there going to be? How much space are they going to require?

Some slices are fairly easy to specify. The root, for example, doesn't need to be very large. You don't normally store much in the root except subdirectories and mount points for other file systems. Many people make their root slice fairly small, such as 50MB.

The directories under */var* typically hold logging and transient data, so this slice doesn't need to be very large. Please note, however, that your log files tend to grow. (See the article "Trimming Your Log Files" on page 7.) The larger this partition is, the longer you can go without purging your logging data. Many system administrators make this partition small as well.

Finally, the */tmp* file system is built in your swap space. This means that you'll run out of space in */tmp* whenever processes require a lot of memory or temporary file space. Thus, if you expect your system to require *X* megabytes of RAM and *Y* megabytes of temporary files, you'll need to make your swap partition at least *X+Y* megabytes. If you're running mission-critical applications on your computer, you need to be especially careful with this swap */tmp* dichotomy. If your system runs out of swap space, Solaris will start killing processes.

Computing the sizes of other slices isn't quite so simple. You'll have to draw on your experience, make some "back of the envelope" calculations, and make some guesses as well.

Conclusion

For some file systems, it makes sense to use just a single slice. Other systems have many users, many drives, and many uses, so they must be partitioned into many pieces. No matter what your requirements are, think about how much control you want to exert over your system before slicing it. ❖

QUICK TIP

Documenting file system mount points

If you manage a system with many disk drives and disk slices, you may find it a bit confusing to keep track of which file system mounts where. Once you're experienced with the system, it's not a big problem—you'll know where to find each file system.

But what happens when you're sick or on vacation? Or, if your system operates on multiple shifts, how do you communicate mounting information between system administrators? Post-It™ notes, however useful, just aren't reliable enough.

In this article, we'll describe a standard method of documenting your file system mount points. Then we'll show you a simple documentation method. Using this method, you can be confident that all system administrators on all shifts will know where to look and what to do.

Create a DiskLayoutInfo file

One method you could use is to create a *DiskLayoutInfo* file in each mount point directory that describes which file systems you should mount there and for what circumstances. However, this method causes you to have lots of little files scattered across many file systems. An additional disadvantage is that when you mount a file system at the mount point, you'll no longer be able to see the *DiskLayoutInfo* file that describes the file systems.

A better technique would be to create a single file that holds all the information. You can describe each file system, where it mounts, and the circumstances for mounting and dismounting it. Then, if you desire, you can link this file at each mount point of interest. (You may want to do this so that when you list a directory, you can tell if it's an unused mount point by the presence of that file.) Using a single file allows all the administrators to find and update the information quickly and easily.

The /etc/vfstab file

Solaris provides the starting point for you: the */etc/vfstab* file. This file contains the defaults

for various file systems. It tracks the default mount points and mounting options of various file systems. If you have multiple file systems that may mount at a particular location, they work well in */etc/vfstab*. However, a file system that may mount at several locations won't work as well.

This situation doesn't preclude you from using the */etc/vfstab* file for documenting your system. Since */etc/vfstab*, like most configuration files, allows you to add comments, you may add as much additional information as you want. For example, if you have a file system on which you store confidential information prior to running a particular job, you could add an entry to */etc/vfstab* like this:

```
#Confidential geological data. Mount to run
➥geosurvey and dismount afterwards!
/dev/dsk/c0t3d0s4  /dev/rdsk/c0t3d0s4  /conf/
➥geodata  ufs   2     yes       -
```

Here, we have a normal file system description augmented with a comment that tells us when to mount and dismount it.

Conclusion

A large system can quickly become difficult to manage. By adding information to the */etc/vfstab* file inside comments, you can make managing the system a little easier. Putting the documentation here makes sense, since you need to be familiar with the */etc/vfstab* file anyway. ❖

LETTERS

Adding a tape drive to your system

I've recently added a SCSI tape drive to my system (a Connor TSM4000R), but Solaris doesn't seem to recognize it. How can I get the system to use it?

Antonio Vaca
via the Internet

Most SCSI drives will work with Solaris after they're configured. We'll first show you how to reconfigure Solaris to recognize the tape drive. After that, we'll show you a quick test you can use to see if your tape drive is operating properly. This procedure assumes that you've installed the tape drive correctly.

Telling Solaris to reconfigure itself

You can tell Solaris to reconfigure itself on bootup via two methods. In the first method, you simply restart your computer and tell it to boot with the `boot -r` command. For you Solaris x86 users, after you select the disk partition you want to boot from, the secondary boot screen gives you a five-second window to enter any boot options you want. You can enter `b -r` here to tell Solaris to reconfigure itself on bootup. In either case, booting with the `-r` option tells Solaris to reconfigure itself on bootup.

However, this method has a disadvantage: If you miss your window of opportunity, Solaris will boot up, and you must repeat the process. If you're in an office where you're easily distracted, this isn't the method you'll want to use.

An even better way to tell Solaris to reconfigure itself is to create the file */reconfigure*. When Solaris recognizes this file, it automatically reconfigures itself. Once it's finished its reconfiguration, Solaris removes this file. You can easily create this file (when you have root privileges) with the command

```
# touch /reconfigure
```

If everything goes well, then you'll see a new item in your */dev* directory—the *rmt* subdirectory. The */dev/rmt* directory contains multiple entries; each is a different mode for accessing the tape drive. The following is a directory listing of */dev/rmt* on one of our machines:

```
$ ls
0   0cb  0hb  0lb  0mb  0u
0b  0cbn 0hbn 0lbn 0mbn 0ub
0bn 0cn  0hn  0ln  0mn  0ubn
0c  0h   0l   0m   0n   0un
```

Each file's name depends on the tape drive number, the storage density, whether to use BSD behavior, and whether to rewind the tape after use. The exception is file *0*, which

simply accesses the tape drive with the default density, mode, etc.

The first digit in these filenames specifies the tape drive number. Therefore, all the files in the preceding directory refer to the same tape drive, drive 0. If there are any additional characters in the filename, the next letter specifies the tape density using the letters *l*, *m*, *h*, *u*, or *c*. These stand for low, medium, high, ultra, and compressed. Please note that ultra and compressed are synonyms for the same density. The appearance of a *b* means to access the tape with BSD behavior. The final letter possibility is *n*, which tells the tape drive not to rewind after the operation is complete. Also note that if you already have one tape drive on your system, Solaris won't create the */dev/rmt* directory—instead, it will simply create new entries, beginning with 1, for the new tape drive.

If the reconfiguration doesn't find your tape drive, check to be sure that you've configured the tape drive correctly. Make certain you have the SCSI ID set correctly, that your cables are firmly seated, and that your SCSI bus is terminated at each end, but nowhere else.

Using your tape drive

After establishing that your */dev/rmt* directory exists and contains the proper list of files, you're ready to test your tape drive. To do so, we'll save a copy of the */usr/bin* directory to tape. Then we'll read the directory to the */export/tape_test* directory.

Before we run our test, let's reset the tension on our tape. (Please note that you'll occasionally want to reset the tension of your tape during normal use as well. Read the manual on your tapes and tape drive for further details.) You can reset the tension on your tape using the `mt` command as follows:

```
$ mt -f /dev/rmt/0 retension
```

The `mt` command allows you to give direct commands to your tape drive. In this way, we specify the tape drive to use with the `-f /dev/rmt/0` option. Then we tell `mt` which command we want the tape drive to execute—`retension`, in this case. If everything works well, your tape drive should fast-forward your tape to the end and then completely rewind the tape. This may take a few minutes, so be patient.

Once you've reset the tension on your tape, it's time to write some files to tape. For our test, we'll use the `tar` command. First, let's save the */usr/bin* directory to tape:

```
$ cd /usr/bin
$ tar cv *
a acctcom 52 tape blocks
a adb 229 tape blocks
a addbadsec 30 tape blocks
a addbib 17 tape blocks
...
```

When `tar` finishes, we'll create the */export/tape_test* directory and extract the archive from the tape to our new directory. To do this, type the following commands:

```
$ mkdir /export/tape_test
$ cd /export/tape_test
$ tar xv
x acctcom, 26580 bytes, 52 tape blocks
x adb, 117064 bytes, 229 tape blocks
x addbadsec, 15008 bytes, 30 tape blocks
x addbib, 8596 bytes, 17 tape blocks
...
```

If you've managed to get this far with no trouble, congratulations! Your tape drive is now ready for use. ❖

A shell function to read a password

The little shell script shown in Figure A on the next page is based upon the tips you described in the article "The Secret of Reading Keyboard Input One Key at a Time" in the July issue. In this case, I wanted to get a string of characters, rather than a single character. Also, I wanted to echo an asterisk (*) for each character entered. I terminate the input on receipt of a carriage return.

I encountered only two difficulties when modifying your script to my needs. First, because I'm entering a string of characters in the loop, I must disable character echoing before entering the while loop. If I don't disable character echoing, then a rapid string of characters will sometimes echo their true value. I solved this by moving the relevant code for storing the `tty` settings, switching to no-echo and raw mode, and resetting the `tty` settings to within the main program body.

The second problem was a little harder. It took me a while to figure out how to get

the `case` statement to compare `userChar` to a carriage-return character. The first case statement below has an embedded carriage return (^M).

In `vi`, you can embed a control character while in input mode by typing [Ctrl]V followed by the control character to embed. Therefore, [Ctrl]V[Ctrl]M will embed a carriage return at the current location. *[Note: We printed all the control characters in blue in the script.]*

I hope this helps someone else.

Shawn Clifford
Miami, Florida

Figure A

```
#! /bin/ksh -f
#----------------------
# getpass - get a password from the user
# Shawn Clifford
#----------------------

# Prepare the terminal settings
oldTtySettings="'stty -g'"
stty -echo raw

# Prompt the user for the password
userPwd=""
echo "Enter your password: \c"

while true
do
   # Get the next key from the user
   userChar="'dd bs=1 count=1 2> /dev/null'"
   case $userChar in

     # We're done when user presses [Enter]
     # NOTE: case pattern is ^M
     ^M)    break;;

     # Process [Backspace] & [Del]
     # NOTE: case pattern is ^H^?
     ^H^?) if [ -n $userPwd ]; then
              userPwd=${userPwd%?}
              echo "^H ^H\c" >&2
           fi;;

      # Add any other key to the password
      *)    userPwd=$userPwd$userChar
            echo "*\c" >&2;;
   esac
done

# restore the TTY settings
stty $oldTtySettings

# print the password to the screen
print "\nPassword=[$userPwd]"
```

This script file allows a user to enter a password without leaving any trace on the screen.

Thanks for the *getpass* script, Shawn. As you can see, we modified your script a little. Specifically, we added the case where we handle the [Backspace] and [Delete] keys. To do so, we first told the script that it had to be run under the Korn shell. The reason we did this is that we wanted the [Backspace] and [Delete] keys to remove the last character entered.

We don't know of a way to remove a single character from a string in a Bourne shell script. In the Korn shell, however, there are facilities that let you take a string apart. We describe these fully in the article "String Extraction in the Korn Shell" on page 10. In this specific case, the interesting line is this one:

```
userPwd=${userPwd%?}
```

Here, we use the Korn shell's `%` pattern matching operator. This operator removes a string from the right of the specified variable. Specifically, it removes the shortest string that matches the `?` pattern. Since the `?` pattern matches any single character, this line tells the Korn shell to remove the rightmost character from `userPwd`.

Just as you mentioned, in `vi` you need to precede any control character with [Ctrl]V before entering it. This is how you get a control character in your file. We used it to put the [Backspace] and [Delete] keys (shown as ^H and ^?) in the script. ❖

Printed in the USA
This journal is printed on recyclable paper.