December 1996 • $11.50
VOL. 2 NO. 12

INSIDE SOLARIS
Tips & techniques for users of SunSoft Solaris™

in this issue

Securing your system with dialup passwords

By Al Alexander

As the world continues the trend toward remote and mobile computing, an ever-increasing concern is maintaining system security over dialup telephone lines. Once you attach a modem and a phone line to your computer, you expose your system to a variety of external threats.

I can recall one instance when a friend and I discovered a modem leading into our company's VAX mainframe computer. We were calling our VAX System Manager and accidentally dialed the last number of his phone number incorrectly. Suddenly we heard the familiar sound of a remote modem.

Curiosity got the better of us, and we reached over to a workstation, used our modem to call the previous phone number, and found a login message from our company's VAX mainframe. Because it was against company policy to have modems attached to our mainframe, this was a startling discovery, which we kept to ourselves.

In this article, we'll explore the process of adding dialup passwords to your Solaris computer's modem lines. This process adds another layer of security to your incoming modem lines by forcing intruders to break an additional password to gain access to your computer system.

Overview

Adding dialup passwords to your modem lines is such a simple process that I recommend it for anyone who uses inbound modems. In this article, we're assuming that you already have incoming modem lines installed and tested. You need to take only three additional steps to add dialup passwords:

1. Create the */etc/dialups* file, specifying the devices that require dialup passwords.
2. Create the */etc/d_passwd* file, specifying the password for each login shell.
3. Test your configuration.

Once you've configured dialup passwords, the new dial-in login process works as follows: When a user attempts to log in to the computer, Solaris first issues a prompt for the username and password, just like normal. Then, if the */etc/dialups* file lists the port, the login program looks in the */etc/d_passwd* file for the shell that the user is attempting to use. Solaris then prompts the user for the dialup password for the shell before granting access.

Creating the */etc/dialups* file

The first file you must create is */etc/dialups*, which specifies for which

A Publication of The Cobb Group

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
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 Artists .... Marguerite Winburn, Liz Palmer
Editor .... Karen S. Shields
Publications Coordinator .... Linda Recktenwald
Circulation Manager .... Mike Schroeder
Editorial Director .... Linda Baughman
VP/Publisher .... Lou Armstrong
President .... John A. 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.

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

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

ports you want the extra password enabled. You create this file by specifying the incoming ports, one per line, that you want secured with a dialup password.

For instance, let's assume you have a computer with three modems that have dial-in access. These modems are connected to ports */dev/term/b*, */dev/term/c*, and */dev/term/d*. In this case, you should create your */etc/dialups* file to contain the following three lines:

```
/dev/term/b
/dev/term/c
/dev/term/d
```

For security purposes, both the user and group ownership should be set to root. In addition, only root should have read or write access to it. Assuming that you're logged in as the root user, you can use these commands to finish setting up the file:

```
# chown root /etc/dialups
# chgrp root /etc/dialups
# chmod 600 /etc/dialups
```

Creating the */etc/d_passwd* file

The */etc/dialups* file we just created tells Solaris which ports we want to secure. Next, we must set up the dialup passwords. Rather than giving each user another password, you must associate the dialup passwords with the login shell. Each login shell may have its own dialup password.

Each line in the */etc/d_passwd* file contains two fields terminated by a colon (:). The first field specifies the login shell to protect, such as */bin/sh* or */bin/ksh*. The second field contains the 13-character encrypted password for the login shell. The encrypted password field in */etc/d_passwd* uses the same format as the one in */etc/shadow*.

If you don't specify a password for a particular login shell, it will use the same password you've defined for */usr/bin/sh*. If you haven't specified a password for */usr/bin/sh*, then the system won't prompt you for a dialup password.

For security reasons, you should always specify a password for the shell */usr/bin/sh*. You want to do this because if you add a new shell to your system, such as `tcsh` or `bash`, you may forget to add an entry for it in the */etc/d_passwd* file. If that happens when you don't have an entry for */usr/bin/sh*, anyone using the new shell will be able to log in without using the dialup password!

TRICK: If you want to use the same dialup password for all shells, you can just set up the entry for */usr/bin/sh* in */etc/d_passwd*. If, on the other hand, you want to use different passwords for different shells, you should add the entry

```
/usr/bin/sh:*LK*
```

to your */etc/d_passwd* file. This prevents anyone from logging in remotely with any program not explicitly mentioned in */etc/d_passwd*. However, it also prevents you from specifying */usr/bin/sh* as the login shell for any remote user. You can still allow a remote user to use the Bourne shell by using `admintool` and specifying */bin/sh* as the shell, and adding an entry for */bin/sh* in */etc/d_passwd*. This way, you get security as the default behavior and can still use the Bourne shell for remote access.

The final */etc/d_passwd* file that provides protection for the Bourne, Korn, and C shells looks something like this:

```
/usr/bin/sh:*LK*
/bin/sh:i4lUy7yLcj4Ru:
/usr/bin/ksh:Bt1Mc0.ejy.XN:
/usr/bin/csh:d2I163trYS3ab:
```

Now let's step through the process of generating a password for the */usr/bin/ksh* entry. First, we must create an encrypted password. Unfortunately, there's no simple process in Solaris to create a password for a shell. However, since the encrypted password for the */etc/d_passwd* file uses the same format as those in the */etc/shadow file*, we can create a temporary user account, set its password, and then copy its encrypted password from the */etc/shadow* file into the */etc/d_passwd* file.

To do so, you need to be the root user. First, create a temporary account named tempuser:

```
# useradd tempuser
```

Next, set the password for the tempuser account to the dialup password you want for the shell */usr/bin/ksh*. In this case, we're going to use Mark16:

```
# passwd tempuser
New password: Mark16
Re-enter new password: Mark16
```

Now find the encrypted password entry for tempuser in the */etc/shadow* file, using the command

```
# grep tempuser /etc/shadow
tempuser:k8AqzGjr6amOY:9762::::::
```

The second field in the line is the encrypted form of the password Mark16; in this case, it's k8AqzGjr6amOY. Now, edit the */etc/d_passwd* file and create the line

```
/usr/bin/ksh:k8AqzGjr6amOY:
```

Don't forget to follow the encrypted password with a colon!

You can create entries for other login shells, such as the Bourne or C shells, by following this same procedure. For those shells, use the tempuser account to generate your encrypted passwords. When you're finished using the tempuser account, delete it:

```
# userdel tempuser
```

Once you've created the */etc/d_passwd* file, give it the same ownership and permissions as the */etc/dialups* file:

```
# chown root /etc/d_passwd
# chgrp root /etc/d_passwd
# chmod 600 /etc/d_passwd
```

Testing your dialup password

The final step in the configuration process is to test your dial-in modem lines. You can do so by dialing in to your Solaris computer and logging in. If everything is set up properly, your login dialog will involve three prompts, shown below. Please note that, for clarity, we've shown the passwords, even though they don't appear on the terminal:

```
login: marco
Password: zort!poit
Dialup Password: Mark16
Last login: Mon Sep 23 13:09:16 on term/b
    • • •
```

You're prompted for the Dialup password only if you get the user name and account password correct. This way, you don't have to wonder which password you may have typed incorrectly: If you get your user password wrong, you won't get the Dialup password prompt. However, this tells a potential hacker when he or she has guessed the password for your account. To help thwart these hackers, Solaris pauses a second before telling you whether the Dialup password is right (by logging you in) or not. This time delay places an upper limit on the speed a hacker may guess passwords. In addition, after a few failed login attempts, Solaris terminates the connection, forcing a hacker to dial up the computer again, consuming more valuable time.

Once you enter the proper dialup password for your login shell, your login process will proceed as usual. When you pass this login test, your configuration process will be complete. When you test your system, be sure to try each different login shell, as well as one that's not in */etc/d_passwd*, to make sure you're happy with the behavior.

More security on modem lines

If you want to make your computer system even more secure, you can take other approaches, such as using `cron` to change the */etc/d_passwd* file on a periodic basis, for instance on shift changes. You can do this by creating a series of files with the */etc/d_passwd* file format, each containing different passwords. Then you can use `cron` to replace the */etc/d_passwd* file with one of these files each time period. The important part for you and your users is to remember when each password is valid.

You can also protect your dial-in modem lines with third-party hardware devices. These devices connect to your telephone line, between the wall jack and the modem. When a remote user dials your modem's phone number, this device picks up the phone line immediately and simulates a telephone ring.

This simulated ring continues unless the remote user has included a security code in the modem dialup string. For instance, if a user is calling the phone number 555-1212, and has a security device with a code of 1234, a proper dial-in string for the remote modem might be

```
atdt5551212,,,1234
```

You could interpret this dial-in string as, "Call the phone number 555-1212, wait six seconds, then issue the code 1234."

When the user includes this security code, it triggers the third-party device, allowing the user to get through this device and into your modem. If the user hasn't included a security code, all that person will hear is a remote ringing telephone. I've found this to be an outstanding security measure.

You can also purchase third-party software that will add the best security possible: callbacks. These give each user a prearranged number to dial in from. Once a user calls in and identifies himself, Solaris then hangs up on the user and calls back at the prearranged number. So even if a hacker figures out the identification of a valid user on the system, he can't log in remotely unless he's at the correct phone number! The big headache for a callback system is managing phone numbers for users that move frequently, such as sales staff. ❖

Alvin J. Alexander is an independent consultant specializing in UNIX and the Internet. He has worked on UNIX networks to support the space shuttle, international clients, and various Internet service providers. He has provided UNIX and Internet training to over 400 clients in the last three years.

CUSTOMIZING YOUR SYSTEM

Tuning your environment for both X and dumb terminals

By Al Alexander

Does your site use both dumb terminals and X terminals with CDE? Are you always changing your PATH, your tty parameters, etc.? If so, you can do a little tuning on your startup files to greatly simplify your environment. In this article, we'll explore the interaction between the CDE startup file, *.dtprofile*, and your shell's startup file, *.profile*, for the Bourne and Korn shell or *.login* for the C shell.

The CDE startup file

The first time you log in to CDE, it automatically creates a default *.dtprofile* file for you. It places the *.dtprofile* file in your $HOME directory.

Each time you log in to a CDE session, it reads and processes the commands in your *.dtprofile* file. It makes certain environment variables available to all your desktop applications. The *.dtprofile* file uses the Korn shell command syntax, so if you're comfortable with the Korn shell, the *.dtprofile* file is easy to work with.

One simple thing you can do is to define the values of specific environment variables within your *.dtprofile* file. Since a command-line login never reads your *.dtprofile* file, you don't have to worry about CDE-specific items interfering with command-line-specific items. Because the CDE reads the *.dtprofile* file only after you log in, you need to log out of the CDE and back in again to test any changes in your *.dtprofile* file.

The shell startup file

When you do a command-line login, your shell never looks at the *.dtprofile* file. Instead, it uses your shell's login file, either *.profile* for the Bourne or Korn shell or *.login* for the C shell. Many people customize their shell's startup file to customize their path, set terminal preferences, create aliases for commands and directories, and many other things.

The hurdles

Obviously, you're going to have differences between your CDE and command-line login environments. However, in most situations you'll want to keep some things the same. When you do so, you'll find that trying to keep both the *.dtprofile* file and your shell's startup file synchronized can be a tedious chore. If you change your preferences for the PATH variable in one startup file, you may have to make the same changes to the other startup file.

For example, if you install some GNU tools on your system, you'll want both the CDE sessions and your command-line sessions to be able to access them. So, you'll need to update the path in both your *.dtprofile* file and your shell's startup file. You'll need to keep other types of data in sync between CDE and command-line sessions.

The tedium and potential for error are obvious. Every time you change *.profile*, you must decide whether you want the same changes reflected in *.dtprofile*. If so, you need to cut and paste the changes to your *.dtprofile* file. You also need to be sure that you put the changes in the right place. Then you need to test both your command-line login as well as your CDE login to make sure you made the changes correctly. And that's just for the Korn shell.

If you're using the Bourne shell, you may encounter some subtle syntax differences between the *.profile* and *.dtprofile* files. So when you cut and paste, carefully examine the text you're moving to make sure that you're not going to create any errors.

It's even worse for the C shell. The syntax is so different that you can't cut and paste between *.dtprofile* and *.login*, except maybe for comments.

A much simpler way

The people who created CDE realized that this was going to be a problem, so they created a simpler way for you to customize your environment and keep things synchronized. Rather than put all your customizations in *.dtprofile* and in your shell's startup files, you can tell CDE to read your shell's startup files after it reads *.dtprofile*. You can do this by setting the new CDE variable DTSOURCEPROFILE to true in *.dtprofile*.

When this new variable is true, it tells CDE to also read your *.profile* (or to read the *.login* file if you're using the C shell). This way, you can put all your environment customizations in the same file. And if you're using the Bourne or C shell, you don't have to worry about the nuances of the Korn-shell syntax. Now you can manipulate all your environment variables, aliases, custom functions, paths, etc., in the same place.

Keeping the CDE and command-line login differences

This approach has potential drawbacks, however. In using this option, you must be careful that the commands in your shell's startup file don't try to set terminal options (tset, tput, etc.), write output to your terminal, or read input from your terminal when you're performing a CDE login. Because the CDE startup process reads *.dtprofile* (and now *.profile* or *.login*) before defining or opening a terminal window, an erroneous command in your startup file can cause your CDE login process to abort or hang.

You can get around this problem by making your startup file a little smarter. The solution is to make sure you run any terminal-related or I/O-generating commands only when you're not in a CDE login. To do so, you can create an if statement to keep separate all the things you want done only in a CDE window from things you want done only in a command-line login. The CDE environment helps this process along by defining the environment variable DT. If you find that DT is set, you can be sure that you're logging into a CDE session. If, on the other hand, it isn't set, you can be sure that you're in a command-line login. Figure A shows a part of a *.profile* file that contains an if statement as described.

Figure A

```
# Commands that may run in both CDE and
# command line logins:
PATH=/usr/bin:/usr/local/bin:/usr/ucb:/etc
MANPATH=/usr/man:/usr/local/man

if [ ! "$DT" ]; then
  # Commands that should run only during
  # a command-line login
  echo "Enter your terminal type: \c"
  read termType
  export TERM=$termType
  stty erase '^H' intr '^C' start '^Q' stop '^S'
  tput init
  tput clear
  echo "Hello again, Marco!"
else
  # Commands that should only run during
  # a CDE session
  MANPATH=/usr/dt/share/man:$MANPATH
  PATH=/usr/dt/bin:$PATH
fi

# More environment configuration...
export PATH=$PATH:$HOME/bin
export MANPATH
export PS1='$LOGNAME:$PWD> '
export VISUAL=vi
alias l="ls -al"
alias help=man
alias cd..="cd .."
alias cd...="cd ../.."
```

Testing the DT variable lets you keep CDE and command-line login environment customization separate.

One advantage of this method is that you can define CDE-specific environment variables that won't interfere with other logins, such as text-based or OpenLook logins. However, the VISUAL editing feature, like many commands in your *.profile* file, isn't specific to any of these environments and is, in fact, desirable in all three. Given the fact that you don't want to duplicate the code in your *.profile* file with similar code in the *.dtprofile* file, this technique maximizes your customization ability and simultaneously minimizes the amount of effort required.

A real-world example

The code shown in Figure A is an example of a *.profile* file you may actually use. If you log in to your system both with CDE and command-line mode (such as from a remote terminal), then you'll probably want to have different, but similar, values for your PATH and MANPATH environment variables (among others).

First, we edited the *.dtprofile* file to uncomment the `DTSOURCEPROFILE=true` line. Please note that as you're going through the *.dtprofile* file, you'll see that Sun provides example code for making your *.profile* or *.login* file act differently in CDE mode.

Notice that the first thing we did was set up the base values for the PATH and MANPATH environment variables. These happen to be the values that we use for the command-line login. Then, our `if` statement contains two sections. If you're in a command-line login, your shell executes the first section, which asks you for your terminal type, sets it up, and prints a welcome message. This would cause us problems if we executed it in CDE. The second section, executed only during a CDE login, adds the CDE commands to PATH and the CDE `man` pages to MANPATH. Finally, we export the path and set up a few environment variables and some aliases.

Debugging

You need to be careful when you edit your *.dtprofile*, *.profile*, and/or *.login* files. If any of these files contain errors, CDE may not allow you to log in. Luckily, it's easy to recover from this sort of error. You simply select a command-line login from the CDE login screen (or log in as the root user if the changes in *.profile* prevent you from logging in using your account), then correct any mistakes in your startup files.

If you're making some complicated changes, it may be easier to extract the section that you want to change into another file. You can then edit this test file and execute the file to test the changes. Once the changes are working correctly, you can put them back into the appropriate startup file. In the Korn shell, you use the `.` command to execute the test file, while you use the `source` command to do so in the C shell.

Conclusion

When you switch back and forth between X terminals and dumb terminals, you should spend a little time customizing your startup files to make your life simpler. Here, we showed you how the CDE *.dtprofile* file and your shell's startup file (*.profile* or *.login*) interact. ❖

SAFEGUARD YOUR PROCESSES

Allow your commands to continue running after you log out

Sometimes you want to run a time-consuming program but, for security reasons, you don't want to leave your account logged in the whole time it's running. If you've ever tried putting the program in the background and logging out, you've found that Solaris terminates your program anyway. In this article, we'll see why this happens, and we'll show you a few ways to get around it.

Why it happens

When you log out, Solaris sends a hangup signal (SIGHUP) to all processes your login shell started, telling them the terminal is no longer active. Normally, a program relies on the default behavior when it receives a signal. However, the default action for most signals, including SIGHUP, is to abort the program. If we could give our programs a new behavior for

SIGHUP that does nothing, then our programs will continue to run.

How to keep your processes alive

If you're writing your own programs, you have a simple way around this. Rather than let the process terminate when it receives SIGHUP, you can tell it to ignore the signal. You can do this in a C program by using the sigignore() function. Figure A shows a simple C program that loops forever, and it uses the sigignore() function to prevent the SIGHUP signal from terminating the process.

Figure A

```
#include <stdio.h>
#include <signal.h>

int main( int argc, char *argv[] )
   {
   sigignore(SIGHUP);
   while ( 1 )
      {
      puts("Loop...");
      sleep(3);
      }
   return 0;
   }
```

This simple C program sits in a loop forever, ignoring any SIGHUP signals sent to it.

Figure B

```
#! /bin/ksh
trap 'echo "SIGHUP ignored"' 1
while true; do
   echo "Loop..."
   sleep 3
done
```

This Korn shell script operates similarly to the C program shown in Figure A.

You can do something very similar with shell scripts. For example, in the Korn and Bourne shells, you can use the trap command to specify an action when a particular signal comes in. Figure B shows a simple Korn shell script that prints a harmless message on the screen when the SIGHUP message arrives.

When you run either the program shown in Figure A or Figure B and send it the SIGHUP signal, the program ignores the signal and continues to run. If you're running in the CDE or OpenWindows environment and close the display window, you can see that it's still running by using the ps -ef command.

The nohup command

What can you do if you're running a program for which you don't have the source code? In this case, you can't modify the source code. However, Solaris provides a command called nohup that does the trick. You execute it like this:

```
nohup command arguments
```

where *command* is the command you want to execute and *arguments* holds the argument list you want to use.

This program starts your command in a new process, telling that process to automatically ignore SIGHUP. Now your program operates normally, even if you log out or your terminal gets disconnected.

Conclusion

We've shown you to get around the SIGHUP command so you can execute your commands without worrying about leaving your terminal unattended. You can sleep easily in the knowledge that your program will execute properly. ❖

CUSTOMIZING CDE

Manipulating your dtterm window with escape codes

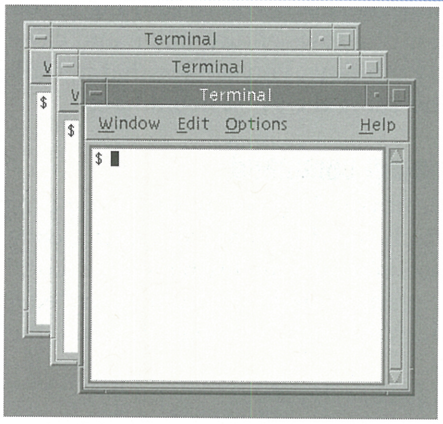
By Al Alexander

In the last issue, we showed you how to change the title of an xterm window in OpenWindows. This month, we'll not only show you how to do the same with a dtterm window, we'll also show how you can change the color of your text and how you can minimize or maximize your dtterm window. Then we'll put together a simple script to allow you to modify your dtterm window. These capabilities can really make things simpler when you're working with many windows open at the same time.

Change your window title

As we described last month, when you work with many windows on the screen simultaneously, it can be difficult to tell them apart at first glance. Even more annoying, the title bar of all the windows defaults to Terminal, which is nearly useless, as shown in Figure A.

Figure A



By default, the title bars convey little useful information.

With descriptive titles on each `dtterm` window, it would be much easier to tell which window was which, whether they were layered on top of each other or presented as icons on the desktop. Unfortunately, the `dtterm` pulldown menus don't offer an option to dynamically change their titles.

This problem is easy to solve if you're comfortable sending escape codes to terminals. As you know, whenever you send text to a terminal, the terminal displays the text. Escape codes are the special case—when the terminal sees an escape code, it performs some special function rather than slavishly drawing the characters on the screen. You can think of escape codes as a way of directly communicating with your `dtterm` window. When you send the right sequence of characters to the `dtterm` window, you'll get the response you want.

For instance, one special sequence of characters tells the `dtterm` window to print text in its title bar. This sequence of characters is `[Esc] ] 0 ; text [Ctrl]-G`. (Please note that we put spaces between the parts of the escape sequence so you can tell them apart. Don't put these extra spaces in your escape sequences.)

The first character, `[Esc]`, alerts `dtterm` to the start of an escape sequence. Next comes], which tells the `dtterm` window that it's the set text parameters escape sequence. The 0 parameter tells the `dtterm` window to use the text parameter as the icon and window title text. The ; separates the first parameter (0) from the second parameter (*text*). The *text* parameter continues until the `[Ctrl]-G`. Thus, to set the window title to *My Title*, you'd send the `dtterm` window the escape sequence

```
[Esc] ] 0 ; My Title [Ctrl]-G
```

The easiest way to send these characters to your `dtterm` terminal is with an `echo` command, as follows:

```
$ echo "^[]0;My Title^G"
```

We surround the escape sequence with quotes (" ") to prevent the shell from interpreting the special characters in the escape sequence. Also note that in order to generate the `^[`, we pressed the [Esc] key, and to get the `^G`, we pressed [Ctrl]-G.

The first parameter, 0, can actually be one of four values, as described in Table A. You may want to take advantage of the fact that you can specify the iconic and window title text separately. When the `dtterm` window is open, you may have plenty of area to describe the window. However, in iconic form, you have little space to describe the window. The final value, 3, is outside the scope of this article, but feel free to experiment with it.

Table A

Value	Description
0	Set window title and icon text
1	Set icon text
2	Set window title text
3	Set current working directory

The set text parameters escape sequence has four modes.

You can put nearly any text value you want in the window title or icon text, so long as it doesn't contain a `[Ctrl]-G` or newline. Thus, if one window contains a telnet session to California, you could set the window text with

```
$ echo "^[]0;California^G"
```

and for your other window, in which you're working on a CAD/CAM project involving a gas range, you can enter

```
$ echo "^[]0;CAD/CAM Gas Range Project^G"
```

Change the text colors

Another way you can help distinguish between windows is with color. It's easy to change the foreground and background colors on the `dtterm` window. For instance, if you want to change one `dtterm` window to use blue as the foreground color, you'd enter

```
$ echo "^[[34m"
```

This special character sequence tells the `dtterm` window to change the foreground text color to blue. Other very similar escape sequences allow you to set the foreground and background colors. Table B shows the colors `dtterm` supports and the corresponding escape code to set the foreground or background color.

Table B

Color	Foreground	Background
Black	`[Esc][30m`	`[Esc][40m`
Red	`[Esc][31m`	`[Esc][41m`
Green	`[Esc][32m`	`[Esc][42m`
Yellow	`[Esc][33m`	`[Esc][43m`
Blue	`[Esc][34m`	`[Esc][44m`
Magenta	`[Esc][35m`	`[Esc][45m`
Cyan	`[Esc][36m`	`[Esc][46m`
White	`[Esc][37m`	`[Esc][47m`
Default	`[Esc][39m`	`[Esc][48m`

You can set the foreground and background colors of the `dtterm` window with these escape sequences.

Manipulate the window state

The `dtterm` window even allows you to change the window status, using the window manipulation escape sequence: `[Esc][`*`paramlist`*`t`. Table C describes the different window states you can specify with the *paramlist* parameter(s). So, if you want to minimize the window, you can use the following command:

```
$ echo "^[[2t"
```

Table C

paramlist	Meaning
1	Restore (de-iconify) the window.
2	Minimize (iconify) the window.
`3;x;y`	Move window to x, y.
`4;h;w`	Resize window to h X w pixels.
5	Put the window in front.
6	Put the window in back.
7	Refresh (redraw) the window.
`8;h;w`	Resize text area to h X w characters.

The window manipulation escape sequence lets you move, resize, and otherwise change the window's state.

By now you must be wondering where these escape codes came from and what other options are available. You can find a complete description of all of them in section 5 of the `man` pages. Just type

```
$ man -s5 dtterm
```

to learn more about the special escape sequences that are available to control the characteristics of `dtterm` windows.

A shell script to make it easy

I've found that it's not easy to remember escape codes, so I wrote a simple shell script to access all the features we discussed in this article. Figure B shows the complete source code for the script `ISOLdtcfg`, which lets you configure your `dtterm` window.

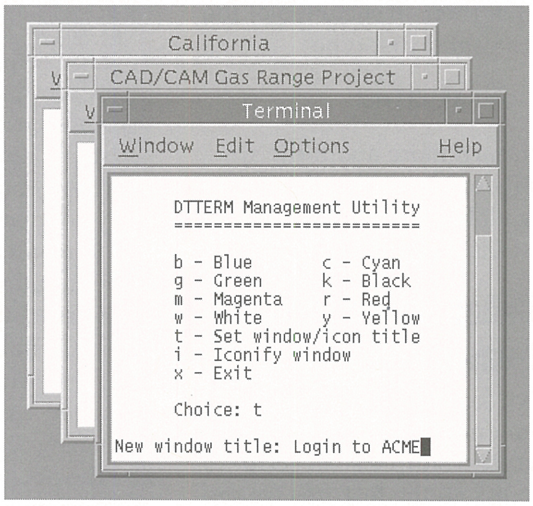
TIP: The window manipulation escape sequence is one of the most exciting escape sequences. To illustrate, suppose you write a script that monitors the system. You don't necessarily want it to take up any room on your screen, so you can tell your window to minimize itself as the first thing in your script. Then, if something happens that demands attention, you can restore the window and put it in front, so it's not obscured by any other windows. You can use this technique to make your shell scripts take screen real estate only when it's required!

Figure B

```
#!/bin/sh
while true
do
   clear
   echo "
      DTTERM Management Utility
      =========================

      b - Blue       c - Cyan
      g - Green      k - Black
      m - Magenta    r - Red
      w - White      y - Yellow
      t - Set window/icon title
      i - Iconify window
      x - Exit

      Choice: \c"

   read CHOICE
   case "$CHOICE" in
      b|B)   echo "\0033[34m";;
      c|C)   echo "\0033[36m";;
      g|G)   echo "\0033[32m";;
      k|K)   echo "\0033[30m";;
      m|M)   echo "\0033[35m";;
      r|R)   echo "\0033[31m";;
      w|W)   echo "\0033[37m";;
      y|Y)   echo "\0033[33m";;
      t|T)   echo "\nNew window title: \c"
              read title
              echo "\0033]0;${title}□";;
      i|I)   echo "\0033[2t"
              exit 0;;
      x|X)   exit 0;;
   esac
done
```

This script, called `ISOLdtcfg`, allows you to interact with `dtterm`'s escape codes.

Figure C



```
DTTERM Management Utility
=========================

b - Blue        c - Cyan
g - Green       k - Black
m - Magenta     r - Red
w - White       y - Yellow
t - Set window/icon title
i - Iconify window
x - Exit

Choice: t

New window title: Login to ACME
```

Here we're telling `ISOLdtcfg` *to change the window title.*

The `ISOLdtcfg` script continuously prompts you for a selection. It exits only when you choose `x` to exit or `i` to iconify (minimize) the `dtterm` window. Otherwise, it sends the appropriate escape code to the `dtterm` window and prompts you for the next command. Figure C shows `ISOLdtcfg` running. Notice that the title on this window is about to be changed to *Login to ACME.*

What else can we do?

When you read the section 5 `man` page for `dtterm`, you'll see that you have quite a lot of control over the `dtterm` window. You can set screen attributes, such as bold and dim. You can move and resize the windows. You can also address the cursor. The escape sequences `dtterm` provides give you flexibility to control and customize your working environment. ❖

C SHELL TIP

Using command-line history with the C shell

Last month, we showed you how to use command-line history in the Korn shell. Using the Korn shell, you use `vi` or `emacs` commands to manipulate previously entered commands. The C shell uses a totally different philosophy. Instead of using editor-like commands to edit the command line onscreen, the C shell provides commands that let you tell it what to do with the command line, but it doesn't show you what it's doing. In this article, we'll explain some of the ins and outs of the C command-line history mechanism.

Command numbering

Each time you enter a command in the C shell, the shell assigns it a number. You can view the last few commands you've entered and the commands' numbers by issuing the `history` command, as shown below. If you prefer to have the latest command displayed first, you can invoke `history` with the `-r` option.

```
Mikado% history
     1  vi ISOL9606a.c
     2  cc ISOL9606a.c -lpthread
     3  ./a.out
     4  gdb a.out
```

Since the C shell keeps the command lines in memory, rather than in a disk file like the Korn shell does, you typically don't have access to much history. The C shell has a shell variable named `history` to tell it how many command lines to remember. You can change this value if you need to do so. For example, if you want the C shell to remember the last 50 commands, just type

```
Mikado% setenv history=50
```

One possible snag that you must be aware of is that the C shell buffers only the command lines that give you the normal prompt. When the C shell issues the auxiliary prompt, it doesn't store them in the history list. Luckily, the C shell issues the auxiliary prompt only when you've given it an incomplete flow control statement. For example, suppose you type the following:

```
Mikado% sync
Mikado% if ( $history >= 50 )
?   echo "Huge history!"
? endif
Mikado% history
     65  sync
     66  if ( $history >= 50 ) then
     67  history
```

As you can see, when we started the `if` statement, the C shell prompted us with the auxiliary prompt (?) until we finished the `if` statement with the `endif` statement. When you examine the history, you see only the first line of the `if` statement. The remainder of the `if` statement doesn't make it into the history list.

How do we use it?

Unlike in the Korn shell, the command-line history mechanism is always active and available to you in the C shell. To access previously entered commands, you use the exclamation mark (!) and follow it with a specifier telling the C shell which command line you want. Table A shows five ways to specify the command line.

Table A

Specifier	Description
!	Use last command entered.
-#	Use the #th previously entered command line.
#	Use absolute command line number #.
string	Use last command starting with *string*.
?*string*?	Use last command line containing *string* anywhere within it.

These are the five basic ways you can specify which command line to use.

The simplest way to specify a command line is to use the ! specifier, which tells the C shell to use the last command line you entered. However, this doesn't allow you to access any command line other than the last one. Here we're going to check the amount of space left on our */tmp* partition, then we'll run the command again:

```
Mikado% df /tmp
/tmp (swap): 126144 blocks   5711 files
Mikado% !!
df /tmp
/tmp (swap): 126144 blocks   5711 files
```

As you can see, after you enter a command line that uses history, the C shell prints the resulting command line before executing the command. Later, when you start using some of the advanced features, the C shell will let you see the results of more complex command lines.

The ! specifier gives you access to only the last command, though. If you want to execute a command other than the last one, you might want to use the -# specifier instead—just replace the # with the number of command lines you want to go back. This allows you to access any of the previous commands. It turns out that the ! specifier is really shorthand for the specifier -1, which tells the C shell to use the previous command. Therefore, you can execute the last command using either the `!!` or `!-1` command. Here, the last command will run the `df /tmp` command again:

```
Mikado% df /tmp
/tmp     (swap):     126144 blocks   5711 files
Mikado% df /proc
/proc    (/proc):    0      blocks   451  files
Mikado% df /dev/fd
/dev/fd (fd):        0      blocks   0    files
Mikado% !-3
df /tmp
/tmp     (swap):     126144 blocks   5711 files
```

If you can see the command you want to access on the screen, you can count the number of commands you've entered since then and use that number for #. If the command you want to use has already scrolled off the screen, and you can't tell how far back to go, you can simply run the `history` command to see the last few commands you've run. Since the `history` command shows the absolute command number, you can use the command number without the hyphen. This way, you don't have to do a mental subtraction.

If the command line you want has scrolled off the screen, you don't necessarily have to use the `history` command and the absolute command-line number. If you remember what the command line started with, and you've used no other commands that start with the same command, you can use the *`string`* specifier. This tells the C shell to find the last command starting with *`string`*.

If you've entered an intervening command line that uses the same command, you can use the ?*`string`*? specifier instead. This specifier lets you locate a command line based on any string in it, rather than just the first one.

In this example, we use `!vi` to edit the *script_program* file. Next, we want to edit the *test_data* file, but we can't use `!vi` or we'll get the `vi script_program` command again, so we'll instead use `!?test?`.

```
Mikado% history
      6  vi test_data
      7  vi script_program
      8  ./script_program
Mikado% !vi
vi script_program
Mikado% !?test?
vi test_data
```

Modify your command lines before executing them

In the examples we've worked through so far, we've just used the command-line history mechanism to re-execute commands. You can actually add to a command line by creating a new command line and inserting the command-line history specifier at the appropriate point. You may even use multiple command-line specifiers in your new command line. The following (admittedly contrived) example shows three commands, where the first two are incorporated into the third one:

```
Mikado% ls
a
Mikado% cp a b
Mikado% echo `!-2` -- !!
echo `ls` -- cp a b
a b -- cp a b
```

Often, you don't really want to execute the same command line; you want to execute one that's similar to a previously executed command line. The most frequent case occurs when you make a typing mistake and don't catch it until the command fails. For this special case, the C shell provides a quick way to repair your command line. Just type `^old^new`, and the C shell will find the first occurrence of `old` on the line, replace it with `new`, then re-execute the command. Here's an example:

```
Mikado% viscript_program
viscript_program: command not found
Mikado% ^vis^vi s
vi script_program
```

Or maybe you don't want to execute the same command, you just want to use one or more of the previous command's arguments. Table B shows you some of the various methods you can use to extract one or more arguments from a command. You use these in conjunction with the command-line specifier.

The C shell breaks the command line into arguments, starting with 0 for the command name and 1 for the first argument of the command, and it numbers the rest sequentially. It uses standard quoting rules, so it usually breaks the command at white space, but quotes allow an argument to contain white space. For example, the command line

```
Mikado% Now is "the time" for
```

contains four arguments. Argument 0 is the command `Now`, and argument 2 holds `"the time"`.

You use an argument specifier by appending a colon and the argument specifier to the end of the command-line specifier. So you can specify the argument `"the time"` in the previous example by using `!!:2` where you want `"the time"` to appear in your new command line.

If you want to use multiple arguments, you can specify them separately or as a group. For example, if you wanted to use the same argument list as the previous example, but with a command named `Then`, you can enter this command:

```
Mikado% Then !!:*
```

Similarly, if you want just the last argument, you can use `!!:$`, or `!!:^` for argument 1. Please remember that the command name is numbered 0.

Table B

Argument specifier	Description
`m`	Use argument *m*.
`m-n`	Use arguments *m* through *n*.
`-n`	Use arguments 1 through *n*.
`m-`	Use arguments *m* through the one before the last.
`m*`	Use arguments *m* through the last one.
`^`	Use argument 1.
`$`	Use the last argument.
`*`	Use arguments 1 through $.

You can tell the C shell to use only a part of the command line with these argument specifiers.

Command-line history in practice

Now you can do some really fancy command-line manipulations. For example, suppose you're developing a program. Normally, you'll go through the standard Edit, Compile, Run, Debug cycle. If you forgo the command-line history, you will type something like this:

```
Mikado% vi Test_Program.c
Mikado% gcc Test_Program.c -lpthread
➥-oTest_Program
Mikado% Test_Program
Mikado% gdb Test_Program
```

Then you'll start the cycle all over again.

Using the command-line history, you can get away with less typing. You can do something like this:

```
Mikado% vi Test_Program.c
Mikado% gcc !!:* -lpthread -oTest_Program
gcc Test_Program.c -lpthread -oTest_Program
Mikado% Test_Program
Mikado% gdb !!
gdb Test_Program
```

Then, instead of typing the same commands for the next go-round, you can just type `!-4` for each command. The first `!-4` starts `vi`, the next `!-4` starts `gcc`, etc.

Advanced features

We still had to type too much in our last example. The culprit in this case is that we need both *Test_Program* and *Test_Program.c*. When you're working with filenames, you'll often want to strip off directory names, extensions, and file modifiers. For this purpose, the C shell allows you to modify individual arguments. Table C shows some of the most-often-used argument modifiers.

Table C

Argument Modifier	Description
`h`	Remove the filename, leaving the directory.
`t`	Remove the directory, leaving the filename.
`r`	Remove the file extension.
`e`	Remove all but the file extension.
`s/x/y`	Substitute *y* for *x*.

You can even edit individual arguments with these argument modifier codes.

To use these argument modifiers, simply add a colon to the end of your argument specifier and follow it with the argument modifier code. The following example will show you how each of these argument modifiers works:

```
Mikado% x /export/home/x.y
Mikado% x !!:1:h
x /export/home
Mikado% x !-2:t
x x.y
Mikado% x !-3:r
x /export/home/x
Mikado% x !-4:e
x y
Mikado% x !-5:s/home/away
x /export/away/x.y
```

Now our Edit, Compile, Run, Debug cycle gets even simpler. The list of commands now looks like this:

```
Mikado% vi Test_Program.c
Mikado% gcc !!:* -lpthread -o!!:1:r
gcc Test_Program.c -lpthread -oTest_Program
Mikado% !!:1:r
Mikado% gdb !!:1
gdb Test_Program
```

And as before, once you start the cycle, you can use `!-4` to access the next command. Now we can perform large sets of related commands with much less typing.

Miscellany

We have a couple more points to make. First, the modifier `:p` tells the C shell not to execute the command after it performs all substitutions and edits. This way, you get the opportunity to examine them before you execute the command. So if you changed the last command in the previous example to

```
Mikado% gdb !!:1:p
gdb Test_Program
```

you'd get to see the substitution, but `gdb` wouldn't actually run. This gets to be very important if you're unsure of a complex edit.

Second, because the ! character tells the C shell that you're about to do something with the command-line history, you need to be sure you escape it (precede it with a backslash) anytime you want to use the ! character rather than access the command-line history.

Conclusion

There are other, finer points that you'll want to investigate after you have a firm grasp on the C command-line history mechanism. When you're ready, execute `man csh`, and then search for the word *History* to get to the relevant section (to do so, just type `/History`). ❖

A quick way to parse a string in a script

When you write scripts, you often must parse the output of a command into tokens. For commands that emit columns of information, your first thought may be to use the `awk` command. This utility does the job neatly: It reads in each successive line of input, parses it into columns, and allows you to specify actions for some or all of the lines. We took a preliminary look at the `awk` command in the May issue in the article "An Introduction to awk."

If you're using the Korn or Bourne shell and you only want to parse a little bit of text, you're going to too much trouble. Not only do you have to tell `awk` what to do, but you're making the computer execute another command. In a resource-critical situation, this method just isn't acceptable.

The simple way

If you have a command that outputs a small amount of data, and you'd like to break the output into fields, there's an easier way. You can instead use the `set` command. By giving the `set` command the `--` option, you're telling it to copy the rest of the arguments of the line to the `$1`, `$2`, `$3`, etc., script variables. Then you can access these variables directly.

All you need to do is pass the results of the command you'd like to parse to the `set` command. You can do so by putting the command in grave accents (i.e., the single back quote `), like this:

```
set -- `command`
```

Suppose, for example, that you want to find out how much disk space is free, in kilobytes, on the root partition. You can do so with the `df` command, like this:

```
$ df -b /
Filesystem          avail
/dev/dsk/c0t0d0s0   356661
```

You'll notice that the `df` command outputs two lines of text, and the number we want is on the second line. That's not really a problem. The `set` command parses tokens on white space boundaries, and the newline is just another white space to `set`. So, putting the two commands together gives

```
set -- `df -b /`
```

After the `set` command executes, `$1` holds *Filesystem*, `$2` holds *avail*, `$3` holds */dev/dsk/c0t0d0s0*, and `$4` holds the value we want, *356661*.

What's the catch?

You may have already noticed the catch in this technique. The `$1`, `$2`, ... script variables normally hold your argument list. So if you use this technique, you lose access to the arguments the user passes to your script. How do we get around this?

You can do this in two primary ways. First, you might just arrange your script such that you no longer need the arguments by the time you use the `set` trick. You can often do so by rearranging some of your script to do some of the work before you need to do the parsing.

Second, you can store your arguments in another location. This is surprisingly easy to do, since the shell variable `$*` contains the entire list of arguments you gave your shell script. All you must do is store the `$*` variable in some other variable, like so:

```
argStorage=$*
```

Then, if you want to restore your argument list to its original condition, just use the `set` trick again, like this:

```
set -- $argStorage
```

There's only one problem with this method: If you didn't pass your script any arguments, then the `set` command will only see `set --`, and it will simply display the environment. It won't restore your argument list properly. You can easily fix this problem by ensuring that the `set` command *always* sees at least one argument, then just remove that argument.

The shell provides us a built-in command named `shift` that removes the first argument and shifts the others down to fill the hole. So all we need to do is give the `set` command a bogus argument that we'll shift out of existence. You can do it like this:

```
set -- bogus $argStorage
shift
```

Now your argument variables `$1`, `$2`, ... are back to their original values.

A simple example

Let's create a simple script to demonstrate the technique. Figure A shows the script `ISOLargs`, which first uses the `set` trick to parse the output of the `df -b /` command. It also demonstrates how to store and recall the original argument list.

Figure A

```
#!/bin/sh
echo "Current argument list is: $*"
argStorage=$*
set -- `df -b /`
echo "Amount of space free on / is ${4}K."
set -- bogus $argStorage
shift
echo "Restored argument list: $*"
```

This script, named ISOLargs, demonstrates the set technique for parsing output from another program.

When you run the `ISOLargs` script, the output looks something like this:

```
$ ./ISOLargs alpha beta gamma
Current argument list is: alpha beta gamma
Amount of space free on / is 356661K.
Restored argument list: alpha beta gamma
```

It even works properly if you don't give it any arguments:

```
$ ./ISOLargs
Current argument list is:
Amount of space free on / is 356661K.
Restored argument list:
```

Conclusion

Whenever you're working on a large data stream, using the `awk` command is certainly the way to go. However, as we've shown in this article, you can handle small jobs in an easier, more efficient way. The `set` trick shown here makes it easy to pick apart the results of many Solaris commands. We hope you give it a try in your Bourne or Korn shell scripts. ❖

LETTERS

Find and grep, revisited

In the September issue, the article "Combining find and grep to Find Any File Anywhere" is useful, but the technique discussed is not very efficient. When combining `grep` with `find`, it's much better to use the `xargs` program so find doesn't have to fork off another `grep` process for every file. You can do it like this:

```
$ find / -type f | xargs grep -il ficonfig
```

Linh Ngo
Boulder, Colorado

It's true that using the `-exec` option to spawn a `grep` job is less efficient than your method. In the method we used, the `-exec` option starts a new process for each file. Since each process has some overhead, this method can cause a serious performance problem, especially when there are a lot of matches. Another problem is that these processes may execute in parallel. Since they're all reading disk files, if your files are on one disk drive, our method may cause severe thrashing on the drive.

Your method avoids these problems. It does so because the `xargs` command collects all the filenames, builds a new argument list for the `grep` job, and runs it only once. This method starts a single process and gives `grep` the complete list of files to search. Not only do you sidestep the overhead of all the individual processes, but the process checks each file sequentially, thereby reducing thrashing on the disk drive.

Another reader mentioned that we shouldn't have used `grep` at all. Using `egrep` instead would speed up the search. With our method, `egrep` might not be the best choice in some situations, because `egrep` consumes more RAM for a complex regular expression. With `xargs`, however, `egrep` is best because it's definitely faster, and with only one process running, the increase in RAM isn't a big deal.

SunSoft Technical Support

(800) 786-7638

PERIODICALS MAIL

/97
[redacted]
0131-1232

Please include account number from label with any correspondence.

Moving a directory branch

In the article "No Matter How You Slice It" in the September issue, the method you presented for moving file hierarchies is bound to lead to trouble, because it doesn't preserve the owner, group, or permissions. Instead, you should use the following commands:

```
$ cd /abc
$ tar cf - def | (cd /xyz; tar xpf -)
$ rm -rf def
$ ln -s /xyz/def /abc
```

David K. Drum
via the Internet

That's a good point. We were so caught up in the act of showing some of the finer points of disk slices that we stubbed our toe on something as simple as this. As you mention, our method will lose all sorts of information about the files, causing security problems and file-access errors. Your method is definitely better, because it preserves all the information *about* each file as well as what's in it.

The first command, `cd /abc`, tells Solaris to move to the parent of the directory we want to move to a new location. The next command is the heart of the technique. A simple copy won't do the trick, so you've built a pipeline of commands.

The first part of the command pipeline, `tar cf - def`, tells `tar` to archive the *def* directory to the file named -. However, `tar` treats the filename - differently than it does normal filenames: The - means to send the archive to the standard output stream, rather than actually write a file.

When this archive flows down the pipeline, it encounters the phrase `(cd /xyz; tar xpf -)`. As you can see, this phrase contains two separate commands in parentheses. We put the commands inside parentheses to start a new subshell, because if we tried to pipe the archive to the `cd /xyz` command, the `cd /xyz` command would simply ignore our archive and pass nothing to the next command. Putting it in a subshell allows the command following `cd`, the `tar xpf -` command, access to the standard stream. The `cd /xyz` command moves us to the parent directory where we wish to move our directory, and the `tar xpf -` directory expands the archive - (now our standard input) in place. All this works because `tar` preserves the ownership and access rights information for all the files in the archive.

The last two commands erase the original directory (`rm -rf def`) then create a symbolic link to the new location (`ln -s /xyz/def /abc`). An interesting point: Since the second `cd` command appeared in the subshell, the directory changed only for the subshell (i.e., the commands inside the parentheses). Once Solaris encounters the right parenthesis, the subshell terminates, and things go back the way they were. ❖

For speed, add RAM during installation

Installing Solaris can take a long time, depending on the machine you're installing on and the features you select. But did you know that the amount of RAM on the machine can significantly affect installation speed?

Installing Solaris on a machine with only a little memory can take a long time. This occurs because Solaris needs to keep track of everything in RAM and doesn't establish the swap space until late in the installation procedure. So when Solaris needs room, rather than swap a little-used piece of data to the swap space, it discards part of the installation program. Then it reloads part of the installation program as it needs to.

It can take significantly longer to install Solaris on a machine with only 16MB of RAM than a machine with 32MB. So if you're trying to get a machine ready as soon as possible, you might want to borrow some RAM to put on the machine during installation. You can remove it when you're finished.

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