

Introduction to Shell Scripting

by: Colin Sauze, 06/27/2005

<http://www.securitydocs.com/library/3402>

This series focuses on shell scripting in Unix/Linux and focuses on using the bash scripting language. The assumption is that you are new to both Unix and programming. However you will still find things easier to understand if you have some programming experience in a high level language like Basic, C/C++, Java or Pascal/Delphi and some experience with Unix or Linux.

What is a shell?

In Unix (that includes Linux, OSX, BSD etc) , after you have been authenticated, you are presented with an interactive shell. If your login is a text based one then you will probably be presented with your shell as soon as you've logged in, if you use a graphical logon as many modern Unix systems do then you can access a command shell through a graphical front end such as X-Term, KDE's konsole or gnome-terminal. The shell allows you to issue commands to the system and instigate other processes. Each user on a Unix system has a default shell, (specified in '/etc/passwd') that they will be launched into at login time. For most systems this is either bourne shell, also known as sh or the GNU Bourne Again Shell often known as bash, which is an extension upon the original bourne shell. There are several other common shells including csh, ksh and zsh, tcsh, these all offer pretty much the same features as bash and only tend to be used by people who started using Unix before bash was mature (mid 1990s). You can find out what your default shell is by typing "echo \$SHELL" at the command line. From here on this tutorial will focus on bash.

What is a shell script?

A shell script is a file which consists of commands you could normally type into the shell. By running the shell script all of these commands are then executed automatically as if you had typed them in one by one. Shell scripting languages include a number of commands similar to those found in most programming languages, these include features to allow you to loop to make an action execute several times, to test if a condition is true or not and perform different actions either way and the ability to jump to another part of the program rather than having to run the program in the exact order the commands are written.

Most shells also provide a number of facilities to ease interactive use (where you type commands one by one), this tutorial doesn't aim to describe these features in any great detail. However a few things you may find useful:

- If you start typing the name of a file or command and press TAB the shell will either finish it off for you or list several possible suggestions if there is more than one.
- Pressing up shows you the previous command you typed, pressing up again the one before that and so on.
- You can make command aliases, so typing one thing actually runs a very different command. This is useful for commonly used commands which are very long or for making one shell seem like another (a common example is aliasing dir which is a Dos command to ls its Unix equivalent).

Why would I need a shell script?

Shell scripts are ideal for writing programs to automate tasks that require you to type in sequences of commands to your shell. Some examples of this might be backing up a directory, compressing it, encrypting it and copying it to another computer or removing a log files older than a certain date. Shell scripts are in use all over the place in most Unix systems, the best example being when the system boots. When most Unix systems boot up they execute a series of shell scripts to start-up all the services they require (known as daemons) such services might include web servers, email servers and hardware

monitoring programs (e.g. Programs to look handle the user connecting a new USB device). Shell scripts are very good at manipulating text data, processing the output of other programs and “gluing” bigger programs together to do something much more useful. They often prove to be a much quicker way of writing a program than using a high level language such as C or Java. They also tend to be very portable between different Unix compatible operating systems (e.g. Between Linux and Solaris) and between different types of hardware, this is because shell scripts are interpreted by the shell as they are run whereas most programs are compiled to a machine code which only certain types of processor understand.

When shouldn't I use a shell script?

Shell scripts do not execute particularly fast and are not well suited to number crunching programs such as data analysis tools, compression, image manipulation etc. They offer little in the way of support for graphical user interfaces or even text mode interfaces and shouldn't really be used to build graphical applications although there are some exceptions to this which will be discussed in part 3. Shell scripts often don't offer the best memory efficiency and if your application really must run with the absolute least amount of memory (or its dealing with so much data that this make a big difference) then you should consider writing it in a compiled programming language like C.

What are the main features of shell scripting languages?

Like most programming languages shell scripts can perform loops so that a given piece of code is run multiple times without the programmer having to write it out multiple times. They can perform logic and take one course of action if something is true and a different one if its false. They can be used to launch other programs, other shell scripts or other parts of the same script. Some shells (such as bourne shell) cannot actually perform arithmetic operations on their own but must enlist the support of extra helper programs to do this, this is not the case with BASH, however for certain arithmetic operations, a lightweight tool such as 'bc' is always useful.

Brief introduction to the Unix/Linux command line

Here are a few common Unix command line commands. Unless otherwise mentioned these work on all versions of Unix and on any shell. This is by no means a complete list.

File manipulation commands:

cat – displays the contents of a file. For example “cat testfile” displays the contents of testfile.

ls – lists the names of files in the current directory. Additional detail can be gained by using ls -l. You can list the contents of a directory your not currently in by typing ls followed by that directory name.

pwd – displays the current directory.

Rm – removes the specified file. For example “rm testfile” will remove a file called “testfile”. Some systems will prompt for confirmation on this, other won't, you can guarantee a file will be removed without confirmation by using rm -f. Directories can be deleted using rm -rf.

Mv – moves or renames a file. This takes 2 arguments, the first is the name of the file that you want to move, the second is the destination. You must make sure the destination file doesn't already exist

Cp – copies a file. Like mv it takes 2 arguments, the file you want to copy and the second is the destination file. If the destination file already exists it gets overwritten. If you specify “.” as the destination it copies the file to the current directory. Using cp -r copies an entire directory structure including any subdirectories.

Ln – links a file to a new location, similar to shortcuts in Windows, accessing the link file gives you the file it refers to. There

are 2 types of links, hard links and symbolic (or soft) links. With a hard link it becomes impossible to decide which is the original file and which is the link. Hard links only work within the same file system. Symbolic links you have one file as the original and the other as the link. To make a symbolic link use "ln -s filename linkname" with filename being the name of the file you want to link to and linkname being the name of the link. If you remove the file the link will still exist but obviously won't work (unless you recreate the file).

Du - tells you how much disk space the current directory takes up in blocks. Du -h gives human readable output (e.g. 1G instead of 1000000)

df - tells you how much disk space is free. Df -h gives human readable output.

Wc - tells you how many words are in a file, can also be made to tell you how big a file (wc -b) is or how many lines of text are in it (wc -l).

head - displays the beginning of a file. Use head -n numberoflines filename to specify exactly how many lines to display.

Tail - displays the end of a file. Use tail -n numberoflines filename to specify exactly how many lines to display.

Text manipulation commands:

awk - awk is used to split up input text into different fields. More details on awk will be explained in part 3.

cut - removes parts of text. More details on cut will be explained in part 3.

Grep - looks for patterns in text and displays lines which match. For example if I had this article in a file and wanted to display all the lines containing the word "script" I could use grep "script" article. Grep is often used in conjunction with pipes (see next section). Grep matches using a pattern matching language called regular expressions, regular expressions will be explained in more detail in part 3.

more - the more program displays a file one page at a time, once a page has been displayed it will say "--More--" at the bottom left of the screen and no more output is displayed until the user presses a key. More is often used in conjunction with pipes (see next section of this article). If it is used alone it takes one argument, the name of the file to display.

Sed - sed is used to alter the text its given. Sed uses the same syntax as grep to match text (regular expressions). Sed will be explained in more detail in part 3.

sort - sorts the text its given into alphabetical order.

telnet - Telnet is traditionally used to connect to another Unix machines and issue commands as if you were sat at that machine. Telnet is now losing popularity in favour of SSH which does the same thing but encrypts all data to prevent data being intercepted by others. Telnet still provides a very useful program for shell scripting as it can be used to connect to other internet services such as email servers and web servers, more details on this will be explained in part 3.

Other commands:

echo - prints out messages on the command line, e.g. Echo "hello world" will cause "hello world" to be displayed. Can also use it with variables to display their values. Two versions of echo exist, one is what is called a shell built in and is part of the shell itself, the other exists as an executable in its own right, the shell built-in will usually run first unless you give the full path (e.g. /bin/echo) of the stand-alone version.

Export - exports an environment variable such as the system path. Environment variables are global settings which can be accessed by any application you launch. A common example is the \$PATH variable which determines where the system should look for binaries when you type a command.

Su - Changes who the current user is, just running su without arguments attempts to change you to the super user (root), if

you follow the command with a username then it will change to that user. Unless you are a super user you will be prompted for a password. For scripts the `-c` parameter is very useful as this allows a command to be run instead of giving a new shell as that user. For example `“su nobody”` will change to user nobody and give you their default shell, `“su nobody -c myprogram”` will run myprogram as nobody, this can be useful for increasing security.

Time – times how long a program takes to run. It requires you to put the program name as a parameter, e.g. `“time ls”` will tell you how long the ls command takes to run. Time will typically display the result in 3 parameters being displayed, user, real and system. User is the amount of time this command spent running in user space, system is the amount of time spent in kernel space and real is the total. All ordinary applications run in user space but make calls to kernel space code to do things like manipulate files, send output, read input etc.

Halt, Reboot and Shutdown. These commands turn off or reboot your system. Halt will either take the system to a state where power can safely be removed or switch it off (it depends on your hardware), reboot will restart the system and shutdown is a longer way of doing these commands (`shutdown -h` halts, `shutdown -r` reboots) with more options, for instance you can specify a deal before shutting down or send a custom message to all logged on users at shutdown time.

Whoami – Displays the name of the user your currently logged on as.

Tar – creates a tar (tape archive) file which is a single file that contains a series of other files. This is similar to a zip file but offers no compression, its common to compress tar files with the gzip compression system. To create a tar file user `tar cvf tarfilename.tar files` e.g. `Tar cvf test.tar file1 file2` will create a tar file called test.tar containing file1 and file2. To extract a tar file use `tar xvf tarfilename.tar` e.g. `Taf xvf test.tar` will extract all files from test.tar. Tar is very well suited to backing up directories or preparing them to be sent via email.

Pipes, Output and Input redirection

Input and Output Redirection

Normally when you run a command line program it has 3 data streams associated with it. These are known as standard input, standard output and standard error. Standard input is by default the keyboard, Standard output and standard error are the console (your monitor). It is possible to redirect these streams to files, this can be useful for providing the same input to a program repeatedly or capturing its output.

Output redirection example:

If I type:

```
“echo “hello world” “
```

I will see “hello world” on my screen.

However if I run

```
“echo “hello world” > helloworld.txt”
```

The file helloworld.txt now contains the text “hello world”.

If I call `echo “hello world again” > helloworld.txt`, then this file will be overwritten with the text “hello world again”, if instead of `>` we use `>>` then the file will be appended instead of being overwritten. This is an example of standard output redirection.

Input redirection is done using the `<` symbol. An example might be:

```
“read < helloworld.txt”
```

The read command reads standard input and stores the response in an environment variable called REPLY, we can display this by typing echo \$REPLY at the command line. If we execute the example above having already executed echo "hello world again" > helloworld.txt then REPLY will be set to "hello world again". If the file contains multiple lines, read will only take in the first line, a program which needed multiple lines of input will read as many lines as it needs.

Standard error can be redirected by using the notation 2>destination. So if we type "blah 2>error.txt" would send all errors for this command to error.txt, as blah isn't a valid command we will see no error output on screen, instead it will go to error.txt. Sometimes in scripts we don't want any error output as its of no help to the script, in this case errors can be suppressed by sending error output to /dev/null (a special file which discards all data sent to it), for example 'blah 2>/dev/null'. Another common need in scripts is to send error output to standard out and treat the 2 as if they are the same thing, this done by specifying 'blah 2>&1'.

Pipes

Pipes allow the output of one program to be sent to another as its input. One common use of pipes is the more program which will only display one screen's worth of text before prompting the user to press a key to see any more. A pipe is represented with a | symbol. An example might be "cat reallylongfile | more", the result of this will be that the first page of reallylongfile is displayed and the "--More--" appears at the bottom of the screen, pressing any key will show us the next page. Another common use of pipes is the grep program, grep looks for patterns in the text it is supplied with and only displays lines containing that pattern. Grep will be explained in more detail in part 3 of this series. A simple use of grep is to simply specify the word you are looking for as an argument. e.g. 'myprogram | grep "pipes example"', this example will run myprogram and send the output to grep, grep will only display lines containing the words "pipes example". Output can be piped more than once, for instance 'myprogram | grep "pipes example" | more' will only display lines containing "pipes example" from the output of myprogram and if there is more than a page worth of output it will prompt me before showing the next page. Pipes provide a very useful way to send output from one process to another and are indispensable in writing shell scripts. Not all programs support reading from a pipe and some need to be told they are reading from a pipe by specifying a - as the input file if they have an argument for an input file. In part 3 we will look at ways around the problem of programs which won't read from a pipe.

References

<http://www.cf.ac.uk/psych/CullingJ/pipes.html> - More information about Unix pipes and input/output redirection.

<http://education.vsnl.com/cyberciti/linuxcom.htm> - Linux Command Line Reference, a quick guide to the Linux command line, including a number of commands mentioned in this series.

<http://computing.fnal.gov/cd/unixlinux/unixatfermilab/html/shells.html> - A general introduction to Shells.

<http://education.vsnl.com/cyberciti/linuxcom.htm> - What can the shell do for you? A short guide to some features of Unix shells to help your general productivity. This isn't totally related to shell scripting but will help you use shells in general and increase your productivity.