

Advanced Topics in Shell Scripting

by: Colin Sauze, 07/28/2005

<http://www.securitydocs.com/library/3489>

This article focuses on more advanced topics in shell scripting including a number of common utilities which can help you write more versatile shell scripts.

Awk, sed and grep.

You will often here these 3 tools talked about together. They are very useful for getting the output format of other programs into a format which is useful to you in a shell script. Awk is mainly used for splitting up text based around what is called a field separator (e.g. In "a:B:c:d" : is the field separator). Sed is used to alter the contents of a file or the output of a program in some way, its often used to remove unwanted output or to alter files. Grep is used to match patterns within its input or a file. Sed and grep both use a pattern matching language called regular expressions, more will be explained about these later on and additional references are available at the end of this tutorial.

Awk

Awk's main use is to split up text into separate fields allowing you to get hold of just part of the input to a script or the output of another program. Awk is actually capable of doing far more than this, whole books exist about awk and this tutorial cannot hope to fully explain it so will focus on field separation as its main use.

By default awk will split its input using spaces to separate each field, the main way awk gets its input is from a pipe from another program although it can read files itself. As an example here we are going to use some text from the output of the ifconfig program in Linux which displays information about the network cards in a machine. We're just going to use the first line of the output here which is "eth0 Link encap:Ethernet HWaddr 00:50:BE:DD:76:CD".

Now suppose we want to find out what the network interface's name is (its eth0) in order to compare it to a variable we already have in our shell script (perhaps something we got from a user's parameter). If we type "echo "eth0 Link encap:Ethernet HWaddr 00:50:BF:D7:77:CF" | awk '{print \$1}'" we will get eth0 as our output. This is because awk is using spaces as field separators and is printing the first field (\$1). Do not confuse \$1 in awk with \$1 in the shell script, they are different things. If we wanted to get the next word using space as our separator "awk '{print \$2}'" would return link, "awk '{print \$3}'" would return "encap:Ethernet" and so on.

Now suppose we want only the first 2 digits of the MAC address(the 00:50:BE:DD:76:CD bit), this can be done by specifying a : as the field separator. Field separators are specified by using the -F option to awk, separators can be a single character or several characters. So to get the first 2 digits of the MAC address we must first get the whole address from the original text. This can be done by getting field number 5 from the original input which will return 00:50:BE:DD:76:CD. We can now send this output to another awk process with a pipe in order to separate it out using a ":", e.g. Echo "eth0 Link encap:Ethernet HWaddr 00:50:BE:DD:76:CD" | awk '{print \$5}' | awk -F: '{print \$1}'" this will now return 00.

Sed

Sed is used to change the output of a program or file in some way. Sed is very useful for removing unwanted output or parts of output as well as actually making changes to files like configuration files. As with awk, we are generally going to want to use sed by piping input into it, although it can be told to read from a file. Sed works by taking an input line enclosed either by ' marks or by " marks (it doesn't matter which). The first thing after this is the choice of command, this is a one letter command. The most common command is probably s which means substitute, another common one is delete which will delete whole lines. This must be followed by a / which is then followed by the first argument to this sed command, if the command has another argument it is then followed by another / etc. The substitute command takes the format *s/searchtext/replacementtext/*

If we wanted to change the text "the quick brown fox jumps over the lazy dog" to "the quick brown ox jumps over the lazy frog"

then we could use sed's substitute mode to do this. For example if we use the command

```
echo "the quick brown fox jumps over the lazy dog" | sed 's/dog/frog/'
```

then all occurrences of dog are replaced with frog. If we now output this to another sed command using a pipe then we can replace fox with ox:

```
echo "the quick brown fox jumps over the lazy dog" | sed 's/dog/frog/' |
```

This will produce the output:

```
"the quick brown ox jumps over the lazy frog"
```

This use of sed will only replace the first occurrence of the specified pattern, this can be changed by putting a "g" on the end of the command (this means global), for example:

```
sed 's/fox/ox/g'
will replace all occurrences of fox with ox and not just the first.
```

Sometimes we don't know the exact word we want to search for so we use what's known as a regular expression to specify a pattern of text to match instead of specifying the exact text. Here is an example of the same use of sed but matching any word which ends with the letters og.

```
sed 's/[a-z]og/frog/'
```

So if we use

```
echo "the quick brown fox jumps over the lazy bog" | sed 's/[a-z]og/frog/'
```

We still end up with "the quick brown fox jumps over the lazy frog". The [a-z] bit means match any character between a and z once. If I wanted to match this happening multiple times there are several modifiers I can put at the end of the [a-z] to specify how many times I want to match the preceding pattern (which is in the last set of []). Here's a list of a few of these:

Modifier	Meaning
*	Match preceding pattern 0 or more times
?	Match preceding item 0 or 1 times but no more
+	Match preceding item at least once.
{n}	Match the preceding item n times
{n,}	Match the preceding item n or more times
{n,m}	Match the preceding item n or more times but less than m times.

As well as putting [a-z] in the square brackets we can put any pattern we like, usually this is meant to be a list of characters and [a-z] is actually a shorthand for [abcdefghijklmnopqrstuvwxyz]. Here are some other short hand patterns you can use:

pattern	meaning

[A-Z]	Capital letters
[A-Za-z]	All letters
[:alnum:] or [0-9a-zA-z]	Numbers and letters
[:print:]	Any printable character
[:cntrl:]	Any control character (e.g. Tab, space, backspace, newline etc)

When a list of characters is specified as long as at least one character is found its considered a match. Anything outside the square brackets must be matched in the order its shown (so using the expression “ab” will produce a match with the text “ab ab” but it will not match with “ba ba”).

These are just a few of the possible patterns you can match. For more see the links in the references section or the Unix man page to grep.

The other thing regular expressions do is allow us to specify that the text we match must occur at the beginning of the line or the end of the line. This is done by putting a ^ at the beginning of the expression to indicate beginning of the line and a \$ at the end of the expression to indicate the end of the line.

When using sed to replace text we must try to make it match the entire text we want to replace, a partial match is no good. Using the example of the ifconfig output we used with awk (“eth0 Link encap:Ethernet HWaddr 00:50:BE:DD:76:CD”), say we want to change the MAC address from 00:50:BE:DD:76:CD to be FF:FF:FF:FF:FF:FF then we could use the following sed expression:

```
echo "eth0      Link encap:Ethernet  HWaddr 00:50:BE:DD:76:CD" | 's/[0-9A-F]{2}:[0-9A-F]{2}:[0-9A-F]{2}:[0-9A-F]{2}:[0-9A-F]{2}/FF:FF:FF:FF:FF:FF/'
```

The characters are required before the { characters to act as escape characters, if they are not specified then the shell tries to interpret the { character and horrible things can happen!

If we only used on occurrence of the '[0-9A-F]{2}': then we would end up with FF:FF:FF:FF:FF:FF : 50:BE:DD:76:CD as our resulting pattern!

Grep

Grep is used to match lines of text in a file or pipe input which match a certain pattern. It uses exactly the same regular expression format as sed does. Grep takes the pattern as its first argument and will then either read from a pipe or standard input if no other arguments are specified or it can read from a file if name is specified after the pattern.

So we can either use:

```
echo "some text" | grep "text"
```

or

```
grep "text" filename
```

Unlike with sed, grep can often get away with only matching part of a pattern as all grep needs to do is have enough information to find a line of text, so if we take the previous example with the MAC address and matching 2 digits followed by a : was enough to figure out that this is the line we want, then our pattern for grep could be “[0-9A-F]{2}:” instead of “[0-9A-F]{2}:[0-9A-F]{2}:[0-9A-F]{2}:[0-9A-F]{2}:[0-9A-F]{2}” which certainly makes for an easier to read piece of code.

It is quite common to use awk, sed and grep all together. Lets say now that we want to extract the MAC address from ifconfig again but change the first 2 digits to FF. Here is a single line of code to do it:

```
ifconfig | grep "[0-9A-F]{2}:" | awk '{print $5}' | sed 's/[0-9A-F]{2}:/F'
```

The grep matches only lines where 2 digits (or capital letters A-F) are followed by a :. This should give us only the line "eth0 Link encap:Ethernet HWaddr 00:50:BE:DD:76:CD" from the whole output which is several lines long. The awk gives us just 00:50:BE:DD:76:CD and the sed changes the 00: to FF:.

Other text manipulation tools.

Although awk, sed and grep are very common tools they are not the only tools available for text manipulation. Some other useful tools are tr, cut, head, tail and wc. Tr translates characters, so it can be used to change all spaces in a file to _'s, unlike sed it can only match single characters. Cut removes specified characters from a string, for instance I can ask for the first character to be removed or only the second and third characters to be displayed. Head and tail are used to give the beginning or end of a file or whatever they are sent via a pipe. Wc tells you have how many lines are in some output or a file. Wc can be used in conjunction with head and tail to display a certain line within a file.

Tr takes 2 sets of parameters, these are known as sets and each one contains a list of characters. Whenever a character from set1 is found in the input it is replaced by the corresponding character in set2. So if for instance set1 is "abc" and set2 is "123", then every a is replaced with a 1, every b with a 2 etc. Very often tr is used only with 1 character in each set. Example:

```
echo "gfedcba" | tr abc 123
```

Cut takes a list of characters to display, this is specified by doing cut -c followed by the list. The list can take ranges like 1-3 meaning characters 1-3, open ended range like 1- meaning 1 onwards, a single number or several values comma separated like 1,3 means display characters 1 and 3. Character number 1 is the first character in the string. Example:

```
echo "gfedcba" | cut -c 5,7
gives: ca

echo "gfedcba" | cut -c 2-
gives: fedcba
```

Head and tail take a parameter of the number of lines to display, if none is specified they display 10 lines. You can either send input via a pipe or specify a filename. Wc is typically invoked using the -l option meaning lines, it can also take the -w (words), -b (bytes) and -m (characters) options. You can find out how many lines a file has by doing "wc -l *filename*".

Its possible to display a given line of a file by:

```
doing "head -linenumber filename | tail -1".
```

You can take this a stage further, if you need to process every line in a file you can use this simple script:

```
#!/bin/bash

#use awk because wc -l put the number of lines, a space and the filename
NUMBER_OF_LINES=`wc -l $1 | awk '{print $1}'`

for i in `seq 1 $NUMBER_OF_LINES` ; do
    LINE=`head -$i $1 | tail -1`
    #line processing stuff goes here
```

```
done
```

Sending output over the network:

It is possible to run scripts on other computer by using the ssh and rsh commands. In order for this to work from a script you need to have an ssh key based login or an rsh rhosts file setup, this stops the script from prompting for passwords. See the references section for more information on this.

You can then run a program or script on another machine by simply doing "ssh *user@host command*" or "rsh -l *username host command*". Host is the name of the other system you are connecting to (or its ip address), user is the username. The output of this can be stored in a variable or piped to another process.

Scripts can also interface (badly) with remote services like SMTP servers, POP servers etc. This require 2 scripts, one writes out all the commands to send to the remote system and puts in any required delays. The second invokes the first but sends its output to the telnet program. One major flaw of this is that you can't read any information back, you can only send data, so it requires you to have a pre determined sequence for information to be sent (as the SMTP email protocol does) and there's no way to handle errors.

Here's an example used to send email.

sendemail.sh script

```
#!/bin/bash

FROM="$1"
TO="$2"
SUBJECT="$3"
BODY="$4"

echo HELO `hostname`

sleep 2

echo "MAIL FROM:<"$FROM">"

sleep 2

echo "RCPT TO:<"$TO">"

sleep 2

echo DATA

echo Subject: $SUBJECT

echo $BODY

echo .
```

```
sleep 2  
  
echo quit
```

Send_welcome_email.sh

```
#!/bin/bash  
./sendemail.sh me@mycompany.com bob@bobscompany.com "hello bob" "hey bob,"
```

When sendwelcome_email.sh is started it sends all the output of the sendemail script to the telnet program which is connected to my SMTP (outgoing email) sever on port 25 (SMTP's port number). This is not a reliable way to send email, but could be useful if you want a script to email you its status. Unix also includes an email program (called mail) which is designed to send email from scripts but it require you run an SMTP server locally.

A few extra tips:

Debug Mode

You can see whats going on inside your script by running `bash -x` followed by the name of your script.

Xargs

Some programs cannot read input from a pipe and instead require it to be placed on the command line for them. Anything which you pipe to `xargs` is then sent to the command you specify, e.g. "Echo "hello" | `xargs ls`" has the same effect as running "ls hello".

Gnuplot

Gnuplot is a tool for drawing graphs, it can read its input from a file listing all the commands and another containing all the data, it can then output to a graphics file. This is very useful if you want to create shell scripts which graph things like network traffic or system performance. A link to a Gnuplot tutorial is in the references section at the end of this tutorial.

Yes

Sometimes you get a series of questions in a program which require the same answer. The `yes` program will just say "yes" over and over again (or whatever you tell it to if you give it an argument), you can then send this through a pipe to a program of your choice. For example if you type "`rm -r directory`" to remove a sub directory some systems require you to confirm every file by pressing `y`. If you type "`yes y | rm -r directory`" then it says yes to everything. (note: `rm` actually has a `-f` option which suppresses the questions, this is just an example use).

CGI

CGI is a method for programs in almost any language to be used to make web pages. To make a shell script do CGI all you need to do is have a line say "Content-Type: text/html" followed by a blank line and then followed by your HTML. There is a link in the references section to more details on CGI. CGI has lost popularity in recent years with the advent of new technologies like PHP, J2EE and .NET, however its still useful if you just want a simple webpage to do display some system data or give the user a simple interface.

Reading keyboard input

Its possible to prompt the user for input during a shell script and read in whatever they type (or whatever comes through an input redirection) using the `read` command. The user's input is placed into a variable called `REPLY`. The `read` command takes now arguments.

Running scripts on the command line:

If you type the first line of a `for`, `if`, `case` or `while` statement on the command line and press enter you will get a `>` prompt, this lets you type in the rest of the program and have it run there and then. This is useful testing sections of code quickly or helping to do some tasks at the command line.

Other scripting languages

Apart from other shell scripts there are several alternatives to bash. These include:

Perl – Perl is a very powerful language with built in regular expressions processing and is designed to offer more facilities than shell scripts do but less than that of a full programming language.

TCL/TK – TCL (pronounced Tickle!) is a more complex scripting language with far more features than bash. TK is an extension for TCL that gives it graphical abilities.

Expect – expect is designed to “listen” to the output of commands and to take different responses based upon the outputs. It is very useful for wrapping around larger programs like installers, it can also use the output of the telnet program (like in the example on the previous page) to talk to other computers via the network but it can actually handle responses. Expect is also based around TCL.

References

<http://www.tldp.org/LDP/abs/html/index.html> – The Advanced Bash Scripting Guide describes a number of advanced concepts with the bash shell that goes beyond the detail of this tutorial.

<http://www-h.eng.cam.ac.uk/help/tpl/unix/sed.html> – Handy one liners for SED, lots of useful single line commands for Sed.

<http://iis1.cps.unizar.es/Oreilly/unix/index.htm> – An online collection of old books by O'Reilly, this includes a book on sed and awk, and several covering useful unix commands. Newer editions of these books are available in print from most book shops.

<http://nl.ijs.si/GNUsl/tex/tunix/tips/tips.html> – Unix Tips and Tricks, a web page full of invaluable references about the Unix command line covering everything from tar files to printing to X Windows, includes references for non-bash shells as well.

<http://www.unixreview.com/documents/s=8274/sam0306g> – Using the xargs command. More information about this very handy command.

<Http://expect.nist.gov/> - Expect's home page.

<http://t16web.lanl.gov/Kawano/gnuplot/index-e.html> – Gnuplot tips, a very good guide on how to use GNU plot to generate graphs.

<http://www.oreilly.com/openbook/cgi> - An out of date version of O'Reilly's “CGI Programming on the World Wide Web”, although out of date it is still completely relevant if you want to write CGI based shell scripts, it is however rather centred on using Perl. The later edition is entirely Perl focused so anybody planning to write shell scripts as web applications may find this version more helpful.

<http://bhami.com/rosetta.html> and <http://bhami.com/unix-rosetta.pdf> – The Unix Rosetta Stone, used to show how to perform various tasks in different types of Unix (like Mac OSX, Linux, Solaris etc). An indispensable aid for anyone writing scripts that need to run on several different types of Unix.

<http://software.newsforge.com/software/04/03/15/211214.shtml?tid=78&tid=82> – Setting up SSH key based login for password-less login.