Linux Academy

Hands-on Lab

Working on the Command Line - Environment Variables

Contents

Lab Connection Information

- Labs may take up to five minutes to build
- The IP address of your server is located on the Hands-on Lab page
- Username: linuxacademy
- Password: 123456
- Root Password: 123456

Related Courses

LPIC-1: System Adminstrator - Exam 101

Related Videos

Environment Variables

Need Help?

Linux Academy Community

... and you can always send in a support ticket on our website to talk to an instructor!

In this lab, we'll continue learning about Bash, with a focus on using environment variables. Environment variables are shortcuts that allow us to store filepaths, commands, or other information in a system-accessible location.

To get started, log in to the server using the credentials provided on the Hands-on Lab page. We'll start out as the *root* user, so switch to it if needed:

```
[linuxacademy@ip] sudo su -
```

View Variables

We'll start by viewing all of our environment variables *and* shell settings for the current session, piping the output to `more` for easier reading:

```
[linuxacademy@ip] set | more
```

This returns a long list of values. By convention, variables are written in all capital letters, with words separated by underscores. Some of the variables will be empty, which we can see by their empty parentheses. Others will have one or more values associated.

For example, the `BASH` value represents the path to the Bash binary:

```
BASH=/bin/bash
```

This allows us to use its value in any application in the current environment by referencing `$BASH`.

Environment Variables

We can also view *only* environment variables:

```
[linuxacademy@ip] env | more
```

Shell Settings

Conversely, we can view *only* shell settings:

```
[linuxacademy@ip] shopt
```

Notice that these values are either *on* or *off*. We can modify any of the shell settings using the `set` command.

Set Shell Options

Next, we'll learn how to change a shell option from *off* to *on* and vice versa.

We'll use the `noclobber` option as an example since it's common to be included on the LPIC exam. This setting allows you to prevent redirects from overwriting files. For example, let's direct the current directory's contents to a test file:

```
[linuxacademy@ip] mkdir tmp
[linuxacademy@ip] ls -al > tmp/test.txt
[linuxacademy@ip] cat tmp/test.txt
```

The contents of the test file are a list of files in your working directory. If we redirect something different to the same location, however, the default behavior is to silently overwrite the previous file.

```
[linuxacademy@ip] ls -al /etc > tmp/test.txt
[linuxacademy@ip] cat tmp/test.txt
```

The test file now contains a list of files and directories in `/etc`.

We can see what shell settings are currently enabled:

```
[linuxacademy@ip] set | grep SHELLOPTS

SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
```

To turn on the `noclobber` setting:

```
[linuxacademy@ip] set -o noclobber
```

Now when we check the settings, we'll see `noclobber` included in the list of active shell options.

```
[linuxacademy@ip] set | grep SHELLOPTS
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor:noclobber
```

We can test the effects by trying to redirect the contents of another directory to our test file:

```
[linuxacademy@ip] ls -al /var > tmp/test.txt
```

This time, we'll see a message telling us we cannot overwrite an existing file, which is exactly what we expected.

To turn the `noclobber` option off again:

```
[linuxacademy@ip] set +o noclobber
```

If we check the shell options again, we'll see that it's been removed from the list of active settings:

```
[linuxacademy@ip] set | grep SHELLOPTS
SHELLOPTS=braceexpand:emacs:hashall:histexpand:history:interactive-
comments:monitor
```

If we redirect something to the test file now, it will allow us to overwrite the existing contents without issue.

Path

One of the most commonly used environment variables is `PATH`. It's used by the system to find binary executable files. The `/usr/bin` directory is one of the locations found in `PATH`.

```
[linuxacademy@ip] ls -al /usr/bin
```

We can see that it contains a number of executable files. Because these are in our `PATH`, we can enter them on the command line without specifying the full location. The purpose of `PATH` is to let us run commands from any directory by letting the system know where to find them.

To show how useful `PATH` is, let's create an example shell script in our `/root` directory:

```
[linuxacademy@ip] mkdir tmpbin
[linuxacademy@ip] cd tmpbin
[linuxacademy@ip] vim test.sh
```

In the `test.sh` file, enter:

```
#!/bin/bash
echo “My test script”
```

Next, add execute permissions to the script:

```
[linuxacademy@ip] chmod 755 test.sh
```

If we go back to the root directory, we can't execute the script by simply entering `test.sh` because the system doesn't know where to find it.

```
[linuxacademy@ip] cd
[linuxacademy@ip] test.sh
-bash: test.sh: command not found
```

To execute the script in this way, we'll need to provide the filepath:

```
[linuxacademy@ip] tmpbin/test.sh
My test script
```

Even if we're in the `tmpbin` directory, we can't simply enter the filename.

```
[linuxacademy@ip] cd tmpbin
[linuxacademy@ip] test.sh
-bash: test.sh: command not found
```

We still need to provide it a relative path to execute it:

```
[linuxacademy@ip] ./test.sh
My test script
```

To change this behavior, we can add an entry in our `PATH` environment variable. First, let's see what our `PATH` contains:

```
[linuxacademy@ip] echo $PATH
```

All of these locations contain binary executable files that can be run without entering a full path. To add the location of our test script to the `PATH`:

```
[linuxacademy@ip] export PATH=$PATH:/root/tmpbin
```

When we check our `PATH` again, we'll see that `/root/tmpbin` has been added. Now that it's in our path, we can navigate to any directory on the system and run our script by calling only its name:

```
[linuxacademy@ip] cd
[linuxacademy@ip] test.sh
My test script
```

Create New Environment Variables

We can also create our own environment variables:

```
[linuxacademy@ip] export MYPATH=/home/user
```

```
[linuxacademy@ip] echo $MYPATH
/home/user
```

Similar to the way in which we modified the existing PATH, we can add new values to our own environment variables.

```
[linuxacademy@ip] export MYPATH=$MYPATH:/etc
[linuxacademy@ip] echo $MYPATH
/home/user:/etc
```

Environment variables don't have to be a path; we can use them to store other kinds of values as well.

```
[linuxacademy@ip] export MYSHORTCUT=128
[linuxacademy@ip] echo $MYSHORTCUT
128
```

History

Another important environment variable is the history, or HISTFILE. This variable controls how the command history is stored for a given user.

We'll want to navigate to our home directory, since that's where our history file lives.

```
[linuxacademy@ip] cd
[linuxacademy@ip] ls -al
total 24
drwx------  2 linuxacademy linuxacademy 4096 Mar  8 20:03 .
drwxr-xr-x. 4 root         root         4096 Mar  8 20:00 ..
-rw-------  1 linuxacademy linuxacademy   69 Mar  8 20:22 .bash_history
-rw-r--r--  1 linuxacademy linuxacademy   18 Jan 11 18:29 .bash_logout
-rw-r--r--  1 linuxacademy linuxacademy  176 Jan 11 18:29 .bash_profile
-rw-r--r--  1 linuxacademy linuxacademy  124 Jan 11 18:29 .bashrc
```

Next, we can take a look to see what it contains:

```
[linuxacademy@ip] cat .bash_history
```

We'll see a list of commands we've executed over time as this particular user. The location of the command history is controlled by the HISTFILE variable, and we can check to see that it matches our .bash_history location:

```
[linuxacademy@ip] echo $HISTFILE
/home/linuxacademy/.bash_history
```

We can also modify the behavior of history tracking using HISTCONTROL.

```
[linuxacademy@ip] echo $HISTCONTROL
ignoredups
```

The output shows a setting, ignoredups, that means history tracking ignores duplicate commands. We can also use a different setting, ignorespace, which will allow us to precede a command with a space to prevent it from being included in the file:

```
[linuxacademy@ip] export HISTCONTROL=ignorespace
```

To test this, let's run a test command:

```
[linuxacademy@ip] ls -al
```

If we check our history, this should be the last command run:

```
[linuxacademy@ip] cat .bash_history
...
ls -al
```

If we list the contents of a different directory, but enter a space before the command, it will not be recorded:

```
[linuxacademy@ip]  ls -al /etc
[linuxacademy@ip] cat .bash_history
...
ls -al
cat .bash_history
```

We may also view the size of our history file, stored in HISTFILESIZE:

```
[linuxacademy@ip] echo $HISTFILESIZE
1000
```

This means we're tracking the last 1000 commands entered. This can be modified using the same method as above.

Review

Environment variables are an incredibly useful tool when working on the command line. They allow you to store information to be used in scripting or simply as a shortcut when entering a command. After finishing this lab, you'll have a solid basis for understanding how administrators use environment to work more efficiently and make their jobs easier. It doesn't have to end there, however. Feel free to examine variables and settings that weren't covered here, and try working with them on your own.

Congratulations! You've completed the lab on environment variables!