Chapter 2 Shell scripting and using Shell as a programming language

Shell Scripting

In Linux you can easily open Gedit and type some code to test a programming idea. But you can also use the Shell (Terminal) itself for programming. This two advantages: the Shell is always available and you can use Linux functions directly to perform small utility functions. With Shell programming you can pretty much use all Terminal commands and control every aspect of the system. For hacking purposes you need root access to perform tasks on a target but it is access to shell scripts that truly give you user power.

The Shell (Terminal) has some superficial similarities to the Windows command prompt. However, it is capable of running more complex functions. These are little programs are known as scripts and are compiled at run-time so no Gcc use is necessary. Shell programming is done in a file just like C programming. With Gedit you open a file with the extension .sh for Shell. Type in in the following line to start writing the *Hello, world* example in a shell.

```
$ gedit helloworld.sh
```

Now you have an empty shell file open. To make use of shell functions the first line needs to be *#!/bin/bash* – the #! Is also referred to as shebang and is necessary to run the interpreter program of the Bourne Again Shell. We complete the shell example by adding two more line.

```
#!/bin/bash
STRING="Hello Hackers!!"
echo $STRING
```

The first additional line states that the word string *Hello World!* needs to be stored in a variable named STRING while the second states that the variable should be printed (echoed) to the command line. The dollar sign $ is vital to denote that it is a variable.

After saving and exiting this file we can try and run our first script. Remember that shell programs don't need to be compiled as they merely run shell commands in a particular sequence that can also be performed on the command line. The shell program will take care of the compiling for you by doing that at run-time. The following line runs the code.

```
$ ./helloworld.sh
```

You can also get input from the user by asking them to type into the command line. Such input can be used to direct the flow of a program with the conditional statement just like in Python.

```
#!/bin/bash
echo 'Hello user!'
```

```
echo 'Please type in F if you're female or M if you're male'
read gender
if [gender = "F"]
then
    echo "Hello woman!"
else if [gender = "M"]:
then
    echo "Hello Man!"
else
    echo "Error, incorrect input!"
fi
```

Unlike with Python the indentation is entirely optional; it is simply convenient for those reading the code. With the command 'read' user input is requested. When they hit the Enter button the data they type is placed into the variable gender and the program resumes. Next follows a conditional check. If the input is either F or M corresponding lines are printed out. As there is no check to what a user can input there is also a third option that says the input was not as requested. The conditional statements are finished with fi.

As the Shell language communicates with the command shell you can easily carry them out in scripts as well. You will need to place the dollar sign before the command. An example command can be PWD

```
#!/bin/bash
echo $PWD
```

Now that the introduction to Shell programming is over we will write a program than creates a backup of all files within a designated directory that can be inputted from the command line. In this program we will be making use of other shell command that were detailed in part 1 of this book. Note that those commands contain ticks ` on either side, not single quotes '. For example, the variable CURDIR (current directory) holds as value the command pwd (present working directory) bracketed on either side by a tick. This tells the shell a command must be executed and not to print pwd to the Terminal.

```
#!/bin/bash

# Script to make back-up copies of files in current directory
TIME=`date +"%F"`
CURDIR=`pwd`
ORIGFILES=`ls`
UNWANTED="bak"
BACKUPFILE="backup-$TIME.tar.bz2"
DESDIR="/home/user/Desktop"

echo "Getting ready to back up $CURDIR"  # the current directory is
retrieved with pwd

if [[ $ORIGFILES =~ $UNWANTED ]] #matched
then
    echo "Already bakked - what now?"; exit 1
else
    for f in $ORIGFILED  # for all files in listed directory
    do
```

```
            cp $f $f.bak  # copy the file and add extension .bak to the
    file
        done
    fi
```

Another addition to the program is the ability to zip all the files. The following lines can be typed or pasted below the lines we already have.

```
    echo ‘Zipping up the files”
    tar -cf hello.tar hello.bak
    echo “done”
    exit 0
```

The above code is the first truly self-contained program that is written for this book that you can use on a daily basis. You might think that copy-pasting the files and zipping it in a GUI environment would be faster. Yet, if you need to create a back-up for a device without a GUI scripts will become invaluable. As the final code example in Shell before I discuss Windows .bat we shall implement a game

```
#!/bin/sh

# unscramble - pick a word, scramble it, and ask the user to guess
#    what the original word (or phrase) was...

wordlib="/usr/lib/games/long-words.txt"
randomquote="$HOME/bin/randomquote.sh"

scrambleword()
{
  # pick a word randomly from the wordlib, and scramble it
  # Original word is 'match' and scrambled word is 'scrambled'

  match="$($randomquote $wordlib)"

  echo "Picked out a word!"

  len=$(echo $match | wc -c | sed 's/[^[:digit:]]//g')
  scrambled=""; lastval=1

  for (( val=1; $val < $len ; ))
  do
    if [ $(perl -e "print int rand(2)") -eq 1 ] ; then
      scrambled=$scrambled$(echo $match | cut -c$val)
    else
      scrambled=$(echo $match | cut -c$val)$scrambled
    fi
    val=$(( $val + 1 ))
  done
}

if [ ! -r $wordlib ] ; then
  echo "$0: Missing word library $wordlib" >&2
  echo "(online at http://www.intuitive.com/wicked/examples/long-words.txt"
>&2
  echo "save the file as $wordlib and you're ready to play!)" >&2
```

```
  exit 1
fi

newgame=""; guesses=0; correct=0; total=0

until [ "$guess" = "quit" ] ; do

  scrambleword

  echo ""
  echo "You need to unscramble: $scrambled"

  guess="??" ; guesses=0
  total=$(( $total + 1 ))

  while [ "$guess" != "$match" -a "$guess" != "quit" -a "$guess" != "next" ]
  do
    echo ""
    echo -n "Your guess (quit|next) : "
    read guess

    if [ "$guess" = "$match" ] ; then
      guesses=$(( $guesses + 1 ))
      echo ""
      echo "*** You got it with tries = ${guesses}!  Well done!! ***"
      echo ""
      correct=$(( $correct + 1 ))
    elif [ "$guess" = "next" -o "$guess" = "quit" ] ; then
      echo "The unscrambled word was \"$match\". Your tries: $guesses"
    else
      echo "Nope. That's not the unscrambled word. Try again."
      guesses=$(( $guesses + 1 ))
    fi
  done
done

echo "Done. You correctly figured out $correct out of $total scrambled
words."

exit 0
```

After you have implemented this game you should be able to understand that shell programming is very powerful. It is at the core of a Linux system administrator's toolbox. Programmers also like it, they often code small examples to test out new stuff before they move on to programming languages that non-Linux systems can also understand. Scripts such as the game we just implemented are already becoming quite long. There are several dozen lines of code. Compared to true commercial applications they are nothing in size and complexity. Nonetheless dealing with complexity is an important programming skill. Programmers work on small bits of code at a time. They make sure that the parts work fine. If the end result isn't what they hoped it would be they can trace the problem quickly to one of the constituent problems. One important thing to remember is that with shell scripts you can easily execute other shell scripts, or program written in another language. In fact, with many open source software projects you

will see a folder containing files with the extension .sh and .bat (batch file) which is the windows equivalent to a script. This will be the topic of discussion in the next paragraph.

Other programming languages

If you use the Shell in the way described above you're performing shell scripting, probably the most common programming tasks anyone on Linux performs. Unlike other programming languages shell scripts are not compiled, instead they are interpreted. The commands in a Shell script are executed directly as though a user was typing them in the command line. The advantages are that more complex commands are possible: you can use conditional statements or perform a task multiple times with a loop. Other interpreted languages besides SHELL include Python, BASIC and the Windows equivalent to Shell called PowerShell.

Batch programming in Windows

Windows has an equivalent to Shell programming called Batch programming. Such scripts are stored in Batch files that have the file extension .bat or .cmd with Windows NT. Batch files can be written with a simple text editor such as Notepad or WordPad. The first example will be a three line program that prints *Hello, World!*

```
@ECHO off
ECHO Hello World!
PAUSE
```

Save the program as helloworld.bat as a plain text documents in your Windows Desktop folder. After you're finished you will find an icon containing two little gears entitled *helloworld*. You have now created you're first batch program. You don't need to compile, it is interpreted just like a Shell program. The program is executed either by double-clicking on the icon or by using cmd and navigating to Desktop. Starting the program requires the user to type in the name of the file, no prefixes such as .\ or extensions such as .bat is necessary.

The output of the program should be as follows

```
Hello World!
Press any key to continue . . .
```

This may be a little but surprising, but it is logical. If the program does not include the command PAUSE it will stop immediately after and you cannot see the output. @ECHO off prevents the command prompts from being printed before every line. The following program has more functionality.

```
@echo off
echo Loading.
ping www.google.com -n 2 > nul
cls
echo Loading..
ping localhost -n 2 > nul
cls
echo Loading...
```

This program prints out the line *loading* three times. Each time with an added dot. Before this happens the screen is cleared off all printout with CLS to make it look like an animation. Meanwhile the ping command is executed twice as an example of commands that takes at least a few seconds to complete.

With the set command user input can be requested

```
set /P Phrase=Please, say something!
```

This will place the user input in variable Phrase. /P is a necessary switch denoting user input.

As you may have heard on the news over the last two decades Windows can be a vulnerable operating system. Much has improved, but unsuspecting users are still known to download files with viruses, worms and rootkits. For a start a .bat file can contain dangerous file executions such file deletion with *del* and file renaming with *ren*. It goes without saying that it will ruin your Windows OS if you were to have these commands run through your entire file system. Nonetheless, careful tests will illustrate my point. Create a folder called test on your Desktop that contains a file called hello.txt, which can remain empty. By executing the following script you will delete everything in that folder, in this case hello.txt.

```
@echo off
::----Delete My Documents----::
del /f /q "C:\Users\%userprofile%\Desktop\test\*.*"
::----------------------------::
```

Alternatively you can switch off the firewall

```
@echo off
::-Disable Windows Firewall--::
net stop "MpsSvc"
taskkill /f /t /im "FirewallControlPanel.exe"
::----------------------------::
```

Both instances are just examples of how easy it is to screw up Windows if you want to. I hope these little examples have shown how you can infect a computer with a virus or worm if it is transmitted as an attachment through Outlook, or with an USB-stick.