w2_shell

Bash

Pattern matching

Try the following:

- `./arguments *` in the folder that contains the arguments program.

  ```
  Argument #0: [./arguments]
  Argument #1: [arguments]
  Argument #2: [arguments.c]
  ...
  Argument #8: [test.c]
  ```

- Make an empty subfolder with `mkdir empty`, switch to it with `cd empty` and then run `../arguments *`.

  ```
  Argument #0: [../arguments]
  Argument #1: [*]
  ```

- Go back to the folder and find three different ways to get the program to produce the above output.

```
./arguments \*
./arguments "*"
./arguments '*'
```

Files with spaces in their names

Create a file with a space in its name by typing `touch "silly file"`. Start typing `cat sill` and then press TAB to autocomplete:

```
cat silly\ file
```

Use this method to get the arguments program to print the following:

```
./arguments Hello\ world!
Argument #0: [./arguments]
Argument #1: [Hello world!]
```

Shell variables

In the shell, `VARIABLE=VALUE` sets a variable to a value and `$VARIABLE` retrieves its value. For example, to save typing a filename twice:

```
p=arguments
gcc -Wall $p.c -o $p
```

If you want to use a variable inside a word, you can use curly braces: `${a}b` means the value of the variable `a` followed by the letter b, whereas `$ab` would mean the value of the variable `ab`.

It is good practice to double-quote variables used like this, because if you tried for example to compile a program called `silly name.c` with a space in its name, then

```
program="silly name"
gcc -Wall $program.c -o $program
```

would expand to

```
gcc -Wall silly name.c -o silly name
gcc: error: silly: No such file or directory
gcc: error: name.c: No such file or directory
gcc: error: name: No such file or directory
gcc: fatal error: no input files
compilation terminated.
```

Correct would be:

```
program="silly name"
gcc -Wall "$program.c" -o "$program"
```

which expands to

```
gcc -Wall "silly name.c" -o "silly name"
```

Note that this does not work as expected either:

```
file=arguments gcc -Wall "$file.c" -o "$file"
```

The problem here is that the shell first reads the line and substitutes in the value of `$file` (unset variables expand to the empty string by default) before starting to execute the command, so you are reading the variable's value before writing it. Leaving off the quotes doesn't help: you need to set the variable on a separate line.

Create a user and a group

Create a new user with `sudo adduser NAME` . Check the user and group files with `tail /etc/passwd` and `tail /etc/group` to check that the new user has been created.

Time to change user: `su brian` and enter the password. Notice that the prompt has changed to `brian@debian12:/home/vagrant$`. So the user has changed, and because `/home/vagrant` is no longer the current user's home directory, it gets written out in full.

Next, create a user `nigel` (or some other name) add both your two new users, but not `vagrant`, to the group `users` (which already exists) using the command `sudo usermod -aG GROUPNAME USERNAME`.

Remove user from a group:

```
sudo gpasswd -d vagrant users
```

Explore file permissions

As user `brian` set up your home directory so that

- You can do everything (rwx).
- Members of the `users` group can list files and change to your home directory, but not add/remove files. You will need to change the group of your home directory to `users` for this, using the command `chgrp -R GROUPNAME DIRECTORY`.
- Everyone else cannot do anything with your home directory.

  ```
  chmod 750 ~
  ```

Create a file in your home directory, check, by using `su USERNAME` to log in as the different users, that:

- `nigel` can view Brian's home directory but not create files there;
- `nigel` can view but not edit Brian's readme file;
- `vagrant` cannot list files in or enter Brian's home directory at all. What happens when you try?

Also as `brian`, make a `private` subdirectory in your home folder that no-one but you can access. Create a file `secret.txt` in there with `nano private/secret.txt` as user `brian` from Brian's home directory. Check as Nigel that you can see the folder itself, but not cd into it nor list the file.

Using `ls -l` as Brian in both `~` and `~/private`, compare the entries for the files `~/readme.txt`, `~/private/secret.txt` and the folder `~/private`. Why do the groups of the two files differ? The rule is that you need permissions on the whole path from `/` to a file to be able to access it.

Setuid

As Brian, create a file `message-brian.c` in your home directory. Compile it and run `chmod u+s message-brian` and check the file again: you should now see `-rwsr-xr-x` for the file permissions. The `s` is the setuid bit. **Setuid (Set User ID)** is a special permission bit in Unix/Linux file systems that allows a file to be executed with the **privileges of its owner**, rather than the user who runs it. This is particularly useful for programs that need elevated permissions to perform specific tasks.

As Nigel go into Brian's home directory and run `./message-brian "Hi from Nigel!"`. Now run `ls -l` and notice that a `messages.txt` has appeared with owner and group `brian`. Although Nigel cannot create and edit files in Brian's home directory himself (he can't edit `messages.txt` for example, although he can read it), the program `message-brian` ran as Brian, which let it create the file.

```
brian@debian12:~$ gcc -Wall message.c -o message
brian@debian12:~$ ls -l
total 20
-rwxr-xr-x 1 brian brian 16176 Nov 30 18:26 message
-rw-r--r-- 1 brian brian   542 Nov 30 18:26 message.c
brian@debian12:~$ chmod u+s message
brian@debian12:~$ ls -l
total 20
-rwsr-xr-x 1 brian brian 16176 Nov 30 18:26 message
-rw-r--r-- 1 brian brian   542 Nov 30 18:26 message.c
brian@debian12:~$ su nigel
Password:
nigel@debian12:/home/brian$ ./message "Hi from n"
nigel@debian12:/home/brian$ ls -l
total 24
-rwsr-xr-x 1 brian brian 16176 Nov 30 18:26 message
-rw-r--r-- 1 brian brian   542 Nov 30 18:26 message.c
-rw-r--r-- 1 brian nigel    10 Nov 30 18:27 messages.txt
nigel@debian12:/home/brian$ cat messages.txt
Hi from n
```

Finding all files with the setuid bit set on a system:

```
sudo find / -perm /4000
```

You might get some errors relating to `/proc` files, which you can ignore: these are subprocesses that find uses to look at individual files.

Apart from `message-brian`, you'll find a few files by default: `sudo`, `mount`, `umount` and `su`. The first one you already know; look up what the next two do and think about why they are setuid. Specifically, what kinds of (un)mounting are non-root users allowed to do according to the manual pages?

Look up the `passwd` program in the manual pages. Why might that program need to be setuid?

```
The passwd command needs to have the setuid permission because it modifies sensitive system files, su
ch as /etc/passwd and /etc/shadow, which are typically only writable by the root user.

By setting the setuid bit, the passwd program runs with the privileges of its owner (usually root), r
egardless of which user executes it. This allows regular users to change their passwords securely wit
hout giving them unnecessary root access to the system.
```

Sudo

Switch back to `vagrant` and run the command `sudo cat /etc/sudoers`. Everything is commented out except `root ALL=(ALL) ALL` and the last line `#includedir /etc/sudoers.d` (this is not a comment!) which contains a single file `vagrant` with the line `vagrant ALL=(ALL) NOPASSWD: ALL` which is why vagrant can use sudo in the first place.

However, note the commented lines such as

```
# %wheel ALL=(ALL) NOPASSWD: ALL
# %sudo ALL=(ALL) ALL
```

If uncommented, the first one would let everyone in group wheel run commands using sudo (this is the default on some other Linux distributions), whereas the second one would allow everyone in the group `sudo` to do this, but would prompt for their own password beforehand.

Open a root shell with `sudo su` as user `vagrant`; this is so we don't get locked out if we break sudo. Edit the sudoers file with `visudo` as root, and add the following line:

```
%users ALL=(ALL) /sbin/reboot
```

You can now switch back to `brian` and do `sudo reboot`.

Shell Scripting

For portable POSIX shellscripts `#! /bin/sh/`
For less portable BASH scripts

`#! /usr/bin/env bash`

```
chmod +x script.sh
./script.sh
```

Programs return a 1 byte exit value. This gets stored into the variable ${?} after every command runs.

```
test $? -eq 0 && echo "Command succeeded\n"
[ $? -eq 0 ] && echo "Command succeeded\n"
```

variables:

```
p="Hello World"
echo "${p}"
export p
unset p
echo "${p}"
set -o nounset
echo "${p:? unset variable}"
```

`${0}` Name of the script

`${1}, ${2}, ${3}…` Arguments passed to your script

`${#}` The number of arguments passed to your script

`${@}` and `${*}` All the arguments

`$(COMMAND)` use to capture the output of command

`$((ARITHMETIC))` use to perform arithmetic operations

`if` statement

```
if CONDITION1; then

elif CONDITION2; then

else

fi
```

- File Checks:
  - `[ -f FILE ]`: True if FILE exists and is a regular file.
  - `[ -d DIR ]`: True if DIR exists and is a directory.
  - `[ -e FILE ]`: True if FILE exists (any type of file).
  - `[ -x FILE ]`: True if FILE exists and is executable.
- String Checks:
  - `[ "$STR" = "value" ]`: True if `STR` is equal to `"value"`.
  - `[ -z "$STR" ]`: True if `STR` is empty.
  - `[ -n "$STR" ]`: True if `STR` is not empty.
- Numeric Comparisons:
  - `[ "$A" -eq "$B" ]`: True if `A` is equal to `B`.
  - `[ "$A" -ne "$B" ]`: True if `A` is not equal to `B`.
  - `[ "$A" -gt "$B" ]`: True if `A` is greater than `B`.
  - `[ "$A" -lt "$B" ]`: True if `A` is less than `B`.
  - `[ "$A" -ge "$B" ]`: True if `A` is greater than or equal to `B`.
  - `[ "$A" -le "$B" ]`: True if `A` is less than or equal to `B`.

`for` statement

```
for VARIABLE in LIST; do

done
```

```
for n in 1 2 3 4 5; do
echo -n "${n} "
done

for n in $(seq 5); do
echo -n "${n} "
done

IFS=','
for n in $(seq -s, 5); do
echo -n "${n} "
done
```

`case` statement

```
case VARIABLE in
    PATTERN1)
        ;;
    PATTERN2)
        ;;
    *)
        ;;
esac
```

`basename` and `dirname`

```
basename [path] [suffix]
```

`path`: The full file path.

`suffix` (optional): A string to remove from the end of the filename (e.g., a file extension).

```
dirname [path]
```

Compile helper exercise

Building on what you saw in the videos and the previous exercises, write a shell script in a file called `b` (for build) that does the following:

- Your script should run under any Bourne-compatible shell (e.g. not just `bash`), and it should be written so that you can call it with `./b`.
- `./b compile NAME` should compile the file of the given name, so for example `./b compile hello` should run `gcc -Wall -std=c11 -g hello.c -o hello`.
- However, your script should accept both `./b compile hello` and `./b compile hello.c` as input, and do the same thing in both cases, namely compile `hello.c`. The output file for gcc in both cases should be called just `hello`.
- If the source file you provided as an argument does not exist (adding `.c` if necessary) then the script should print an error message and return a nonzero exit status - *not* invoke the C compiler.
- `./b run NAME` should run the program, assuming it exists in the current folder, so both `./b run hello` and `./b run hello.c` should run `./hello`. If it does not exist, again print an error message and exit with a nonzero status, don't try and run the program.
- `./b build NAME` should first compile the C source file, and then if the compile was successful it should run the program. If the compile failed, it should not try and run the program.
- If you call `./b` without any parameters, or `./b COMMAND` with a command other than compile or run or build, it should print some information on how to use it. If you call `./b compile` or another command with no filename at all, then the script should print an error message and exit with a nonzero exit status.

  ```
  #! /usr/bin/env bash

  if [ ${#} -eq 2 ];then
  ```

```
    NAME=$(basename ${2} .c)
    if [ ! -f "${NAME}.c" ];then
        echo "non-existing file"
        exit 1
    fi

    if [ "${1}" = "compile" ];then
        gcc -Wall -std=c99 "${NAME}.c" -o "${NAME}"
        exit 0
    fi

    if [ "${1}" = "run" ];then
        ./"${NAME}"
        exit 0
    fi

    if [ "${1}" = "build" ];then
        gcc -Wall -std=c99 "${NAME}.c" -o "${NAME}"
        ./"${NAME}"
        exit 0
    fi
fi

echo "run the bash with \"./b build FILENAME.\" "
```

Strict Mode

Use the following line near the top of your shell scripts: `set -euo pipefail`

- `set -e` makes the whole script exit if any command fails.
- `set -u` means referencing an undefined variable is an error.
- `set -o pipefail` changes how pipes work: normally, the return value of a pipe is that of the *last* command in the pipe. With the `pipefail` option, if any command in the pipeline fails (non-zero return) then the pipeline returns that command's exit code.

Pipes

- `cat [FILENAME [FILENAME...]]` writes the contents of one or more files to standard output. This is a good way of starting a pipe. If you leave off all the filenames, cat just reads its standard input and writes it to standard output.
- `head [-n N]` reads its standard input and writes only the first N lines (default is 10 if you leave the option off) to standard output. You can also put a minus before the argument e.g. `head -n -2` to *skip the last 2 lines* and write all the rest.
- `tail [-n N]` is like head except that it writes the last N lines (with a minus, it skips the first N ones).
- `sort` reads all its standard input into a memory buffer, then sorts the lines and writes them all to standard output. `r` reverse order.
- `uniq` reads standard input and writes to standard output, but skips repeated lines that immediately follow each other, for example if there are three lines A, A, B then it would only write A, B but if it gets A, B, A it would write all three. A common way to remove duplicate lines is `... | sort | uniq | ...`.
- `wc [-l]` stands for word count, but with `l` it counts lines instead. `w` count words.
- `command | tee [-a] [file...]` . copy the output of command to file as well as standard output. `a` append instead of overwriting.

  `0` standard input

  `1` standard output

  `2` standard error `out 2> file`

  `|` standard output → standard input

  `>` standard output → file (overwrite)

  `>>` standard output append to file

  `<` file → standard input

  `<<<` string as temporary file to standard input

  `2>&1` merge standard error into standard output `ls nonexistingfile > output 2>&1`

If PROGRAM needs a file to read from but pipe in (subshell):

```
PROGRAM <(SOMETHING)
cat <(echo "Hi")
```

subshell to argument:

```
COMMAND $(SOMETHING)
echo $(ls -l)
```

Word list exercises

dictionary file `/usr/share/dict/words`

- The number of words in the words file - there is one word per line.

  ```
  cat /usr/share/dict/words | wc -l
  ```

- The 6171st word in the file.

  ```
  head -n 6171 /usr/share/dict/words | tail -n 1
  ```

- The first five words that are among the last 100 words on the list.

  ```
  tail -n 100 /usr/share/dict/words | head -n 5
  ```

- The last ten words in the file, sorted in reverse order.

  ```
  tail -n 10 /usr/share/dict/words | sort -r
  ```

Redirection

Find a directory on your VM that contains some files, and is writeable by your current user.

- `ls -l` prints a detailed listing of the directory to standard output. Redirect this listing to instead store it in a file called `tmp`.

  ```
  ls -l > tmp
  ```

- If you inspect the file content (e.g., by `cat tmp`) and compare it to what you see when re-typing `ls -l` now, you should be able to find a difference. The `diff` utility takes in (at least) two filenames and produces output that compares the two files.

  ```
  diff tmp <(ls -l)
  ```

- Redirect it to a file `difflog`. By creating this file, we're changing what `ls` will report about this directory. Let's keep track of that change by running the same `diff` command again, but this time *appending* the new output to `difflog`.

  ```
  diff temp <(ls -l) >> difflog
  ```