Fundamental Skills - Bash Scripting

Category	Experience Level
Automation	Novice

Contents

Intro

We'll go over some simple examples of bash scripting in this lesson, a powerful Unix-based tool for automation.

All examples below with the `$` prefix indicate a command that is being typed into the command line.

Why Automate?

Scripting lets you:

- run any command available on Unix, and even run other scripts
- define custom reusable functions
- run multiple commands chained together with *loops* and *conditional statements*
- capture user input with flags or using the `read` function

Scripting in bash is good for several use cases:

- quickly repeating simple, repetitive tasks, such as re-running exploits to regain a foothold on a machine
- creating dynamic programs that can be used in multiple situations or on different targets
- running installation, deployment, and setup scripts that interact with your OS at a low-level
- creating quick shortcuts around your file system, e.g. to navigate to a common folder, create a server to host an enumeration script, or to start a program with certain settings
- parsing data from log files with text processing commands like `awk`, `sed`, and `grep`

Why *not* `bash`?

- more complex exploits, such as a buffer overflow or a long automated sequence, may be easier to write in python or ruby

Command Line Automation

Before we look at *scripts*, there are some other things we can do to make command-line work faster.

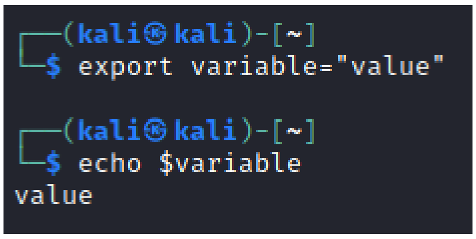
Bash Variables

You can set bash variables (aka *environment variables*) within your terminal, and then use them within commands.

To set a variable, use the `export` command:

```
export VARIABLE=value
```

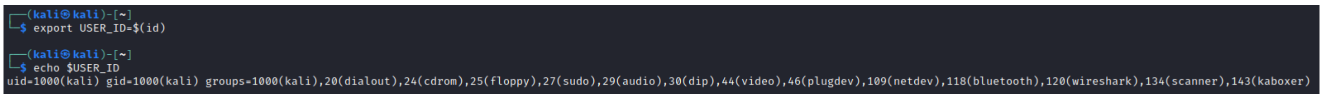
You can then print the value:



```
┌──(kali㉿kali)-[~]
└─$ export variable="value"

┌──(kali㉿kali)-[~]
└─$ echo $variable
value
```

You can even set the variable to the result of another command with an evaluation using `$()` - for example:



```
┌──(kali㉿kali)-[~]
└─$ export USER_ID=$(id)

┌──(kali㉿kali)-[~]
└─$ echo $USER_ID
uid=1000(kali) gid=1000(kali) groups=1000(kali),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),118(bluetooth),120(wireshark),134(scanner),143(kaboxer)
```

To set a persistent variable, define it in the `~/.bashrc` file in your home directory:

```
$ nano ~/.bashrc

...[edit the file]...

export VARIABLE_NAME=value
```

Aliases

Aliases are a simple way of adding a small script straight into your command line. Think of it as a custom terminal command.

If you are using a standard bash terminal, aliases are defined in your `~/.bashrc` file. If you're using something like `zsh`, you'll need to edit `~/.zshrc` or equivalent.

> Tip: See what terminal you're using by typing `echo $TERM`

To add an alias, add a line like the following anywhere in your `.bashrc` file:

```
alias command_name="command"
```

For example, you might group up some common aliases based on their purpose, using a comment (`#`) to indicate what they do:

```
# Edit common files
alias nanbash="nano ~/.bashrc"
alias nanhosts="sudo nano /etc/hosts"

# Start webserver in enum directory
alias enumserve="cd ~/Documents/enum; python3 -m http.server"
```

As you can see, you can run commands with root permissions (`sudo`), chain commands together with `;`, and even change your environment (such as with `cd`)

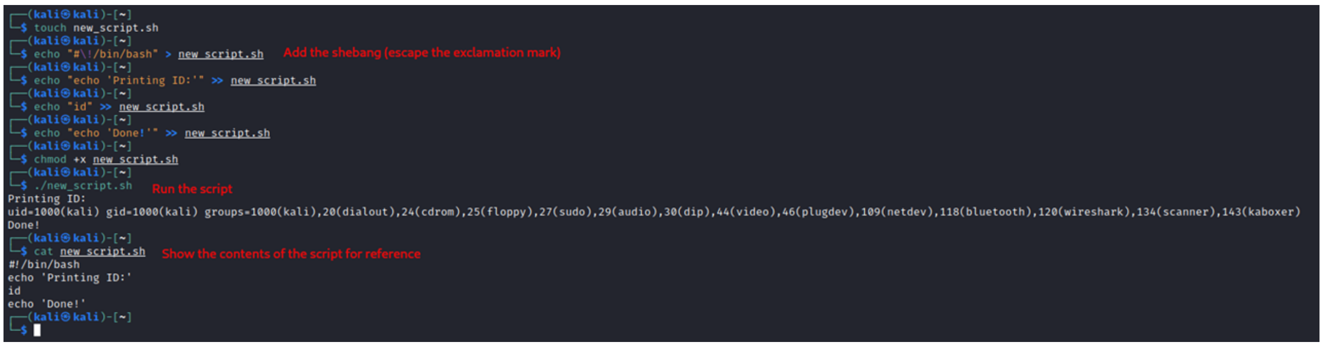
Creating a Bash Script

These are the main steps to make a new script:

- Create a file: script.sh
- Add a shebang to the top of your file
  - This tells Linux what program to use to execute the file
  - For bash, it's `#!/bin/bash`

- Write some commands in your script
  - You can use a text editor, like `vim` or `nano`
  - Or you can send a command to the file with `echo "commands here" > script.sh` (or append with `>>`)
- Make the file executable:
  - `chmod +x script.sh`
  - `chmod 711 script.sh`
- Run the script:
  - `./script.sh`

Here's an example:



```
┌──(kali㉿kali)-[~]
└─$ touch new_script.sh
┌──(kali㉿kali)-[~]
└─$ echo "#\!/bin/bash" > new_script.sh
┌──(kali㉿kali)-[~]
└─$ echo "echo 'Printing ID:'" >> new_script.sh
┌──(kali㉿kali)-[~]
└─$ echo "id" >> new_script.sh
┌──(kali㉿kali)-[~]
└─$ echo "echo 'Done!'" >> new_script.sh
┌──(kali㉿kali)-[~]
└─$ chmod +x new_script.sh
┌──(kali㉿kali)-[~]
└─$ ./new_script.sh
Printing ID:
uid=1000(kali) gid=1000(kali) groups=1000(kali),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plugdev),109(netdev),118(bluetooth),120(wireshark),134(scanner),143(kaboxer)
Done!
┌──(kali㉿kali)-[~]
└─$ cat new_script.sh
#!/bin/bash
echo 'Printing ID:'
id
echo 'Done!'
┌──(kali㉿kali)-[~]
└─$ █
```

Add the shebang (escape the exclamation mark)
Run the script
Show the contents of the script for reference

> Note: in the example above we typed echo "#!/bin/bash" with a backslash before the exclamation mark - this is to 'escape' the exclamation mark, which is a special character in bash

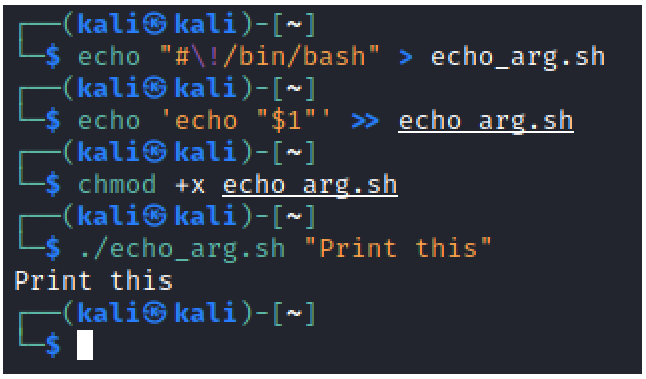
User Input

You can take user input as *positional arguments*. These are supplied after the name of the script (e.g. `./script.sh argument1`) and accessed within the script with `$x` (where `x` is the x-th argument).

Here's a (very) simple example script that prints (with `echo`) the contents of the first argument:

```bash
#!/bin/bash
echo "$1"
```

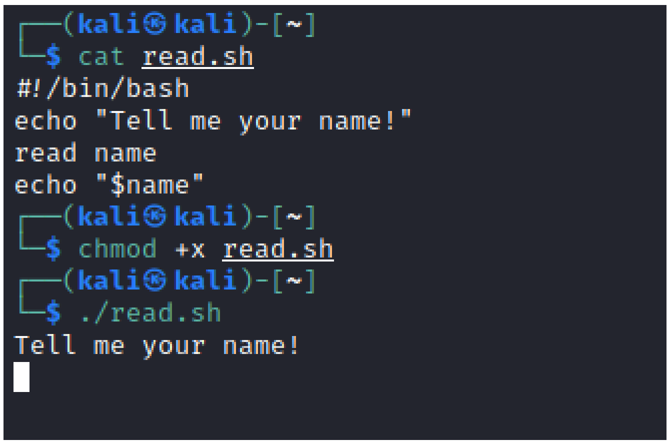
Here's an example:


```
┌──(kali㉿kali)-[~]
└─$ echo "#\!/bin/bash" > echo_arg.sh
┌──(kali㉿kali)-[~]
└─$ echo 'echo "$1"' >> echo_arg.sh
┌──(kali㉿kali)-[~]
└─$ chmod +x echo_arg.sh
┌──(kali㉿kali)-[~]
└─$ ./echo_arg.sh "Print this"
Print this
┌──(kali㉿kali)-[~]
└─$ 
```

You can also use `read` to take user input *during* the execution of the script:

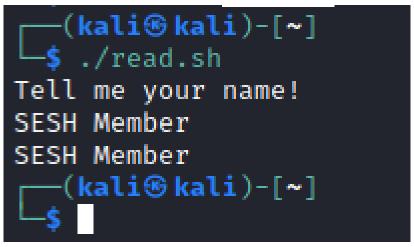
```
#!/bin/bash
echo "Tell me your name!"
read name
echo "$name"
```

Here the script waits for input:


```
┌──(kali㉿kali)-[~]
└─$ cat read.sh
#!/bin/bash
echo "Tell me your name!"
read name
echo "$name"
┌──(kali㉿kali)-[~]
└─$ chmod +x read.sh
┌──(kali㉿kali)-[~]
└─$ ./read.sh
Tell me your name!

```

After input is received, it's saved to the `$name` variable and printed:


```
┌──(kali㉿kali)-[~]
└─$ ./read.sh
Tell me your name!
SESH Member
SESH Member
┌──(kali㉿kali)-[~]
└─$ 
```

You can also use named flags as arguments - this process is a bit more complicated, and there are a few different ways of doing it.

One way is to use a `for` loop (more on these later) to loop over all the parameters provided, and check which ones are present. Here's an example:

```
for arg in "$@"
do
```

```
    case $arg in
       -s|--standalone)
       STANDALONE="Standalone flag present"
       shift ;;
       -i|--input)
       INPUT="$2"
       shift
       shift ;;
       -h|--help)
       print_usage
       exit 1 ;;
       *)
       OTHER_ARGUMENTS+=("$1")
       shift ;;
    esac
done

echo "$STANDALONE"
echo "Flag with input present - input: $INPUT"
echo "Other parameters: ${OTHER_ARGUMENTS[*]}"
```

`shift` is used to move on and process the next character on the command line. It is also possible to use `getopts` - an example can be found here.

Conditions

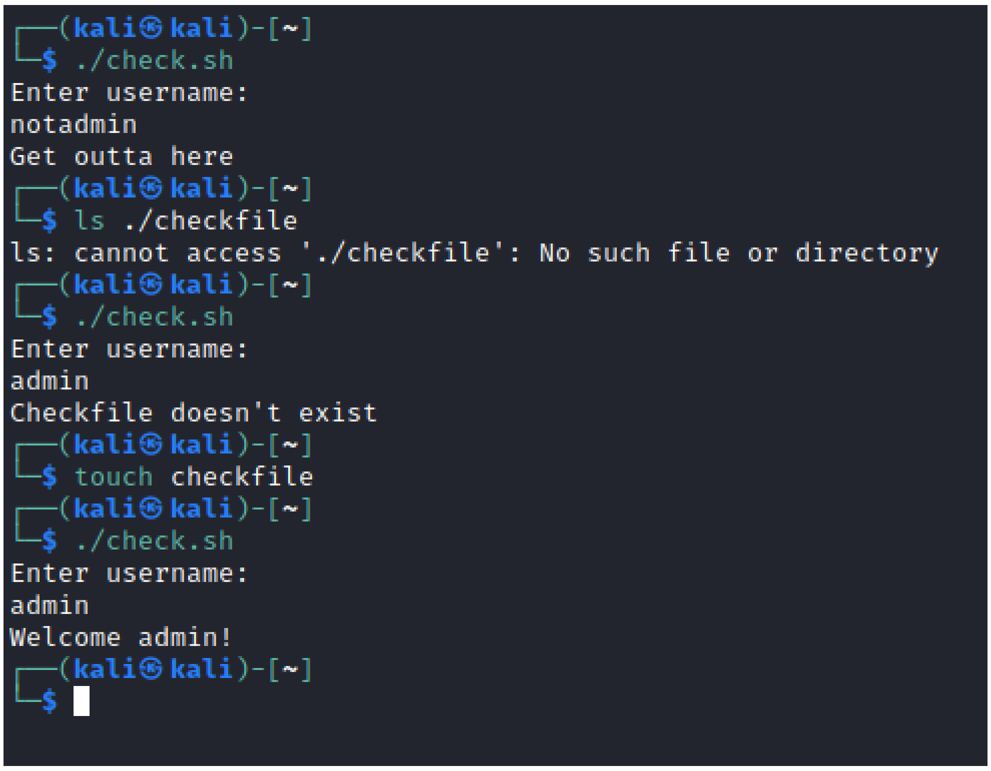
Bash supports simple logical statements, such as `if`, `elif`, and `else`, as well as numerous logical operators (such as `=`, `!=`, `-gt` and `-lt` for greater than/less than, and `-e` to check whether files exist).

Here's an example of a simple script that checks if a provided username is correct, then checks whether a file exists:

```
#!/bin/bash
echo "Enter username:"
read name
if [ $name != "admin" ]
then
    echo "Get outta here"
elif [ -e "./checkfile" ]
then
    echo "Welcome admin!"
else
```

```
    echo "Checkfile doesn't exist"
fi
```

Here's an example:



```
┌──(kali㉿kali)-[~]
└─$ ./check.sh
Enter username:
notadmin
Get outta here
┌──(kali㉿kali)-[~]
└─$ ls ./checkfile
ls: cannot access './checkfile': No such file or directory
┌──(kali㉿kali)-[~]
└─$ ./check.sh
Enter username:
admin
Checkfile doesn't exist
┌──(kali㉿kali)-[~]
└─$ touch checkfile
┌──(kali㉿kali)-[~]
└─$ ./check.sh
Enter username:
admin
Welcome admin!
┌──(kali㉿kali)-[~]
└─$ 
```

Loops

You can use `for` and `while` loops in bash:

- `for` loops repeat the code inside the loop as per a certain number of items (e.g. for every line in a file)
- `while` loops repeat the code inside the loop as long as a condition is true

A simple `for` loop could be written like so:

```
for ip in $(seq 1 10)
do
    ping -c 1 10.11.1.$ip
done
```

This iterates over each number in the sequence 1 to 10, assigns the number to the variable `$ip`, and runs `ping` using that variable.

A while loop can be written like so:

```
count=1
while [ $count -lt 11 ]
do
    echo $count
    count=$[$count + 1]
done
```

This simply counts up to 10, increasing the count by 1 each time.

Custom Functions

You can define functions in bash, which are repeatable code blocks that can take arguments.

Here's a simple definition of a function:

```
get_logs () {
    echo "$(tail /var/log/apache2/access.log)"
}
```

This will print the most recent 10 lines in the Apache access log (Apache is a common webserver).

You can then call this function multiple times in the script to monitor the logs, with ten second breaks in between:

```
get_logs () {
    echo "$(tail /var/log/apache2/access.log)"
}

get_logs
sleep 10
get_logs
sleep 10
get_logs
```

You can also give arguments to a function:

```
echo_name () {
    echo "$1"
}
```

```
echo_name "First Name"
echo_name "Second Name"
```

Cheatsheet

Define an Alias

```
alias name="command"
```

Reload your bash environment:

```
$ source ~/.bashrc
```

For Loops

Standard syntax:

```
for item in $list
do
  [bash command with $item]
done
```

One line:

```
$ for item in $list; do [bash command with $item]; done]
```

Worksheet

1. Create a script to change directory to `/tmp`, clone the shefesh.com git repository, and open `index.html` with the `firefox` command
2. Start a simple Python server in the cloned directory. Then write a script that uses `curl` to request the `index.html` file, find all the `<p>` elements with `grep`, and output their contents with the tags stripped