

# Hacker Tools: Shell & Scripting

---

Jing Yen UWU

7 February 2023

Slides at <https://hckr.cc/ht-shell-slides>

(Materials developed by Julius)

# Where are we?

Introduction

Shell

Scripting

Conclusion

# NUS Hackers



<http://nushackers.org>

**hackerschool**

Friday **Hacks**

**Hack** & Roll

**Hacker** Tools

# About Me

Hi! I'm Jing Yen. My GitHub is  
`https://github.com/JingYenLoh`

I'm a Year 4 Computer Science Undergraduate who loves  
hacking and building systems.

I don't watch anime.

# What you will learn today

How to hack on a Unix-like environment:

- How to use the shell
- How to create scripts for automation

# Required Software

Unix-like environment, either one of these:

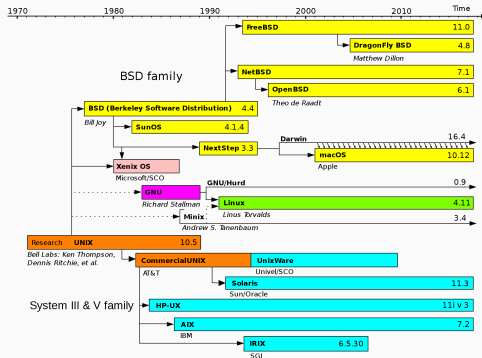
- Linux (you're good if you attended and installed Linux during our Linux Install Fest last week)
- macOS<sup>1</sup>
- BSD
- Other Unix-like OS'es (Minix, Solaris, AIX, HP-UX, etc.)
- WSL (Windows Subsystem for Linux) should also be alright, but no guarantee

---

<sup>1</sup>Open Terminal, and run `xcode-select --install` first

# Unix? Can I eat that?

- A family of multitasking, multiuser OS'es.
- First developed in the 1970's.
- Popularised the use of interactive command line.



# The Unix Philosophy

1. Write programs that do one thing and do it well.
2. Write programs to work together.
3. Write programs to handle text streams, because that is a universal interface.



# Where are we?

Introduction

Shell

Scripting

Conclusion

# Introduction to Shell

- An efficient, textual interface to your computer.
- Provides an interactive programming language (“scripting”).
- Many shells to choose from:
  - Standard ones: `sh` or `bash`
  - Shells that match languages: `csh`
  - “Better” shells: `fish`, `zsh`
- For this workshop, the focus is on the ubiquitous `sh` and `bash`.<sup>2</sup>

---

<sup>2</sup>Feel free to explore other shells. On macOS, many people prefer `fish` or `zsh`

# The Shell Prompt

- What greets you when you open a terminal.

```
0 16:21:57 julius@r-165-105-25-172:~/GitHub/hackerschool-hackertools  
501 (master) $
```

- Lets your run programmes and commands.

# Common Commands

- `man` to get the **man**ual pages of a command
- `cd` to **ch**ange **d**irectory
- `ls` to **l**ist files and directories
- `mkdir` to **ma**ke **d**irectory
- `rm` to **re**move files and directories
- `cp` to **co**py file
- `mv` to **mo**ve file
- `pwd` to **pr**int **w**orking **d**irectory

# Command Editing Shortcuts

`bash` has shortcuts based on `emacs` keybindings:

- `Ctrl` + `a`: beginning of line
- `Ctrl` + `e`: end of line
- `Alt` + `b`: move back one word
- `Alt` + `f`: move forward one word
- `Ctrl` + `k`: delete from cursor to the end of line
- `Ctrl` + `_`: undo

And some special ones:

- `Ctrl` + `u`: delete from cursor to the start of line
- `Ctrl` + `w`: delete from cursor to start of word

You can find more in documentation for `readline`

# Command Control Shortcuts

- **Ctrl** + **c**: terminates the command
- **Ctrl** + **z**: suspends the command (**fg** to continue)
- **Ctrl** + **l**: clears the screen
- **Ctrl** + **s**: stops the output to the screen
- **Ctrl** + **q**: allows output to the screen

# Fun (?) Commands

- `:() { : | : & };;` - what is fork bomb
- `rm -rf /` - what is destroy everything

# Where are we?

Introduction

Shell

Scripting

Introduction

Shell Syntax

Composability

Job and Process Control

Exercises



# Script (1/2)

You can write programs directly at the prompt, or write into a file (writing scripts)

```
1 #!/bin/sh
```

```
2 echo something
```

- Open an editor (for beginner, nano is recommended), save the script as `example-script`
- On your shell, run `chmod +x example-script`
- You can run your script as `./example-script`

# Script (2/2)

```
1 #!/bin/sh
2 echo something
```

Magic?

- *#!/bin/sh* is also known as the **shebang**, specifies the interpreter<sup>3</sup>
- **echo** is a command that prints its arguments to the standard output.

---

<sup>3</sup>You can use other interpreters too, e.g. *#!/usr/bin/env python* for a python script.

# Flags (1/3)

- Most command line utilities take parameters using **flags**.
- They come in short form (`-h`) and long form (`--help`)
- Usually, running `COMMAND -h` or `man COMMAND` will give you a list of the flags the program takes.
- Short flags can be combined: `rm -r -f` is equivalent to `rm -rf` or `rm -fr`

# Flags (2/3)

- A double dash `--` is used in to signify the end of command options, after which only positional parameters are accepted.
  - For example, to create a file called `-v`, Use `touch -- -v` instead of `touch -v`
  - For example, to grep a file called `-v`, `grep pattern -- -v` will work while `grep pattern -v` will not.

# Flags (3/3)

Some common flags are a de facto standard:

- `-a` commonly refers to all files (i.e. also including those that start with a period<sup>4</sup>)
- `-f` usually refers to forcing something, e.g. `rm -f`
- `-h` displays the help for most commands
- `-v` usually enables a verbose output
- `-V` usually prints the version of the command

---

<sup>4</sup>In Unix, by convention files whose names begin with a period is hidden. The origin is an accident, find out more [here](#)

# Unix Directory Structure

Unix has a different directory structure from Windows.

There is no concept of drives.

Everything is files and directories. The root directory is /

We use forward slash / instead of backward slash \

Specifically for Linux, there is FHS<sup>5</sup>

---

<sup>5</sup>[https:](https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard)

[//en.wikipedia.org/wiki/Filesystem\\_Hierarchy\\_Standard](https://en.wikipedia.org/wiki/Filesystem_Hierarchy_Standard)

# Important Unix Directories

- `/bin`, `/sbin`, `/usr/bin`, `/usr/local/bin`, `/opt` = executables
- On Linux: `/home` = user home directories
- On macOS: `/Users` = user home directories
- `/var/log` = log files
- `/tmp` = temporary files
- `/dev/urandom` = random number generator

# Where are we?

Introduction

Shell

Scripting

Introduction

Shell Syntax

Composability

Job and Process Control

Exercises



# Running a command

```
echo Hello
```

■ `COMMAND ARG1 ARG2 ARG3`

# Variables (1/3)

```
echo location  
name=Julius  
echo $name
```

- Used to store text
- `name=value` to set variable
- `$name` to access variable

## Variables (2/3)

There are also a number of special variables:

- `$?`: get exit code of the previous command
- `$1` to `$9`: arguments to a script
- `$0`: name of the script itself
- `$#`: number of arguments
- `$$`: process ID of current shell

# Variables (3/3)

Create a script variable-example containing the code below, then try running it with various arguments.

```
1  #!/bin/sh
2  echo $0
3  echo $1
4  echo $2
5  echo $#
```

# Loop (1/4)

Loop is used to run a command a bunch of times.

For example:

```
for i in $(seq 1 5); do echo hello; done
```

# Loop (2/4)

```
for i in $(seq 1 5); do echo hello; done
```

Let's unpack this!

```
for x in list; do BODY; done
```

- ; terminates a command – equivalent to newline
- Split `list`, assign each to `x`, and run `BODY`
- Split by “whitespace” – we will get into it later
- Compared to C, no curly braces, instead `do` and `done`

# Loop (3/4)

```
for i in $(seq 1 5); do echo hello; done
```

Let's unpack this!

```
$(seq 1 5)
```

- Run the program `seq` with arguments 1 and 5
- Substitute the `$(...)` block with the output of the program
- Equivalent to

```
for i in 1 2 3 4 5; do echo hello; done
```

# Loop (4/4)

```
for i in $(seq 1 5); do echo hello; done
```

Let's unpack this!

```
echo hello
```

- Everything in a shell script is a command
- Here, it means run the `echo` command, with argument `hello`.
- All commands are searched in `$PATH` (colon-separated)
- Find out where a command is located by running `which COMMAND`, e.g. `which ls`



# Conditionals (1/2)

```
if test -d /bin; then echo true; else echo false; fi;
```

Let's unpack this!

```
if CONDITION; then BODY; fi
```

- `CONDITION` is a command.
- If its exit code is 0 (success), then `BODY` is run.
- Optionally, you can also hook in an `else` or `elif`

## Conditionals (2/2)

```
if test -d /bin; then echo true; else echo false; fi;
```

Let's unpack this!

```
test -d /bin
```

- `test` is a program that provides various checks and comparison which exits with exit code 0 if the condition is true<sup>6</sup>.
- Alternate syntax: `[ condition ]`, e.g. `[ -d /bin ]`

---

<sup>6</sup>Remember, you can check exit code using `$?`

# Everything Together

Let's create a command like `ls` that only prints directories:

```
1  #!/bin/sh
2  for f in $(ls)
3  do
4      if test -d $f
5      then
6          echo dir $f
7      fi
8  done
```

# Bug!

Hold on! What if the directory is called "My Documents"?

- `for f in $(ls)` expands to  
`for f in My Documents`
- Will first perform the test on My, then on Documents
- Not what we wanted!

# Argument Splitting

- Bash splits arguments by whitespace (tab, newline, space)
- Same problem somewhere else: `test -d $f`
- If `$f` contains whitespace, `test` will error!
- Need to use quote to handle spaces in arguments `for f  
in "My Documents"`
- How do we fix our script?
- What do you think `for f in "$(ls)"` does?

# Globbering (1/2)

- `bash` knows how to look for files using patterns:
  - `*`: any string of characters
  - `?`: any single character
  - `{a,b,c}`: any of these characters
- Thus, `for f in *` means all files in this directory
- When globbing, each matching file becomes its own argument
- However, still need to make sure to quote, e.g.  
`test -d "$f"`

# Globbering (2/2)

You can make advanced patterns

- `for f in a*:`

# Globbering (2/2)

You can make advanced patterns

- `for f in a*:` all files starting with a in the current directory
- `for f in foo/*.txt:`



# Globbering (2/2)

You can make advanced patterns

- `for f in a*:` all files starting with a in the current directory
- `for f in foo/*.txt:` all .txt files in foo
- `for f in foo/*/p??.txt:`

# Globbing (2/2)

You can make advanced patterns

- `for f in a*:` all files starting with `a` in the current directory
- `for f in foo/*.txt:` all `.txt` files in `foo`
- `for f in foo/*/p???.txt:` all three-letter text files, starting with `p`, in subdirectories of `foo`

# Other whitespace issues

- `if [ $foo = "bar" ]; then`: What's the issue?

# Other whitespace issues

- `if [ $foo = "bar" ]; then`: What's the issue?
- What if `$foo` is empty? arguments to `[` are `=` and `bar`
- Possible workaround: `[ x$foo = "xbar" ]`, but very hacky

# Other whitespace issues

- `if [ $foo = "bar" ]; then`: What's the issue?
- What if `$foo` is empty? arguments to `[` are `=` and `bar`
- Possible workaround: `[ x$foo = "xbar" ]`, but very hacky
- Instead, use `[[ CONDITION ]]`: bash built-in comparator that has special parsing
- Good news: it also allows `&&` instead of `-a`, `||` instead of `-o`, etc.

# shellcheck

- The mentioned problems are the most common bugs in shell scripts.
- A good tool to check for these kinds of possible bugs in your shell script: `https://www.shellcheck.net/`

# Where are we?

Introduction

Shell

Scripting

Introduction

Shell Syntax

Composability

Job and Process Control

Exercises

# Composability

- Shell is powerful, in part because of **Composability**
- You can chain multiple programs together, rather than one program that does everything
- Remember **The Unix Philosophy**:
  1. Write programs that do one thing and do it well.
  2. Write programs to work together.
  3. Write programs to handle text streams, because that is a universal interface.



# Pipe (1/2)

```
dmesg | tail
```

Let's unpack this!

```
a | b
```

- Means run both a and b, but send all the output of a as input to b, and then print the output of b

## Pipe (2/2)

You can chain this even longer!

```
cat /var/log/sys*log | grep "Sep 10" | tail
```

- `cat /var/log/sys*log` prints the system log
- This output is fed into `grep Sep 10`, which looks for all entries from today.
- This output is then further fed into `tail`, which prints only the last 10 lines.

# Streams

- All programs launched have 3 streams:
  - STDIN: the program reads input from here
  - STDOUT: the program prints to here
  - STDERR: a second output that the program can choose to use.
- By default, STDIN is your keyboard, STDOUT and STDERR are both your terminal

# Stream Redirection (1/2)

- However, this can be changed!
- `a | b`: makes STDOUT of `a` the STDIN of `b`.
- `a > foo`: STDOUT of `a` goes to the file `foo`
- `a 2> foo`: STDERR of `a` goes to the file `foo`
- `a < foo`: STDIN of `a` is read from the file `foo`
- `a <<< some text`: STDIN of `a` is read from what comes after `<<<`
- You can also pipe to `tee` (look up in `man` what `tee` does)

# Stream Redirection (2/2)

**So why is this useful?**

# Stream Redirection (2/2)

**So why is this useful?**

It lets you manipulate output of a program!

# Stream Redirection (2/2)

## So why is this useful?

It lets you manipulate output of a program!

- `ls | grep foo`: all files that contain the word `foo`
- `ps | grep foo`: all processes that contain the word `foo`
- On Linux: `journalctl | grep -i intel | tail -n 5`:  
last 5 system log messages with the word `intel`  
(case-insensitive)
- Note that this forms the basis for **data-wrangling**, which will be covered later.

# Grouping Commands

`(a; b) | tac`

- Run a, then b, and send all their output to tac<sup>7</sup>
- For example: `(echo qwe; echo asd; echo zxc) | tac`

---

<sup>7</sup>tac print in reverse



# Process Substitution

b <(a)

- Run a, generate a temporary file name for its output stream, and pass that filename to b
- To demonstrate: `echo <(echo a) <(echo b)`
- On Linux: `diff <(journalctl -b -1 | head -n20) <(journalctl -b -2 | head -n20)`
- This shows the difference between the first 20 lines of the last boot log and the one before that.

# Where are we?

Introduction

Shell

Scripting

Introduction

Shell Syntax

Composability

Job and Process Control

Exercises

# Job (1/2)

Used to run longer-term things in the background.

- Use the `&` suffix
  - It will give back your prompt immediately.
  - For example: `(for i in $(seq 1 100); do echo hi; sleep 1; done) &`
  - Note that the running program still has your terminal as STDOUT. Instead, can redirect STDOUT to file.
  - Handy especially to run 2 programs at the same time like a server and client: `server & client`
  - For example: `nc -l 1234 & nc localhost 1234 <<< test`

# Job (2/2)

- `jobs`: see all jobs
- `fg %JOBS`: bring the job corresponding to the id to the foreground (with no argument, bring the latest job to foreground)
- You can also background the current program: `^Z`<sup>8</sup>, then run `bg`
  - `^Z` stops the current process and makes it a job.
  - `bg` runs the last job in the background.
- `$!` is the PID of the last background process.

---

<sup>8</sup> `Ctrl` is usually denoted as `^`, thus `Ctrl+z` is denoted as `^Z`

# Process Control (1/2)

- `ps`: lists running processes
  - `ps -A`: lists processes from all users
  - Check out the man page for other arguments.
- `pgrep`: find processes by searching (like `ps -A | grep`)
  - `pgrep -f`: find processes with arguments
- `kill`: send a *signal* to a process by ID (`pkill` to search and run `kill`)
  - Signal tells a process to do something
  - `SIGKILL` (`-9` or `-KILL`): tell it to exit *right now* (equivalent to `^\\`)
  - `SIGTERM` (`-15` or `-TERM`): tell it to exit gracefully (equivalent to `^C`)

## Process Control (2/2)

- `kill`: send a *signal* to a process by ID (`pkill` to search and run `kill`)
  - Signal tells a process to do something
  - Most common<sup>9</sup>:
    - SIGKILL (-9 or -KILL): tell it to exit *right now* (equivalent to `^\`)
    - SIGTERM (-15 or -TERM): tell it to exit gracefully (equivalent to `^C`)

---

<sup>9</sup>Prefer SIGTERM over SIGKILL:

<https://turnoff.us/geek/dont-sigkill/>

# More Resources

- If you are completely new to the shell, you might want to read a comprehensive guide, such as BashGuide<sup>10</sup>.
- For a more in-depth introduction, The Linux Command Line<sup>11</sup> is a good resource.

---

<sup>10</sup><http://mywiki.woledge.org/BashGuide>

<sup>11</sup><http://linuxcommand.org/tlcl.php>

# Where are we?

Introduction

Shell

Scripting

Introduction

Shell Syntax

Composability

Job and Process Control

Exercises



# xargs

- Sometimes piping doesn't quite work because the command being piped into does not expect the newline separated format.
- For example, `file` command tells you properties of the file.
- Try running `ls | file` and `ls | xargs file`
- What is `xargs` doing?

# Other Exercises

- Try running `touch {a,b}{a,b}`, then `ls`. What appeared?
- Sometimes you want to keep STDIN and still output to a file. Try running `echo HELLO | tee hello.txt`
- Run `echo HELLO > hello.txt`, then `echo WORLD >> hello.txt`. What are the contents of `hello.txt`? How is `>` different from `>>`?

# Where are we?

Introduction

Shell

Scripting

Conclusion

# Talk to us!

- **Feedback form:**

<https://hckr.cc/ht-shell-feedback>

- **Upcoming Hacker Tools:**

Data Wrangling, , 6.30pm, Same Location!

<https://hckr.cc/ht-data-signups>

- **Happening this Thursday!:**

Applications of OpenCV: <https://hckr.cc/hs2223-w5>

- **Upcoming Friday Hacks:**

HackRoll 23: More Hacks: <https://hckr.cc/links>