

# Hacker Tools: Data Wrangling

---

Julius Putra Tanu Setiaji

22 September 2019

Slides at <https://is.gd/2020ht3>

Download data at <https://is.gd/2020ht3data>

# Where are we?

Introduction

sed and Regular Expression (regex)

More Advanced Data Wrangling

Exercises

Conclusion

# NUS Hackers



<http://nushackers.org>

hackerschool

Friday Hacks

Hack & Roll

Hacker Tools

# About Me

Hi! I'm Julius. My GitHub is  
<https://github.com/indocomsoft>

A Year 4 Computer Science Undergraduate who loves  
hacking and building systems.

Check out **fluminus** and **fluminurs** on my GitHub!

(I'm looking for someone to take over as maintainers after I graduate)

I also enjoy Space Exploration, Music Theory and History.

(my favourite games are KSP and EU4 hit me up if you play those too)

# Required Software

Unix-like environment, either one of these:

- Linux (you're good if you attended and installed Linux during our Linux Install Fest)
- 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

# What is Data Wrangling?

- Have you ever had a bunch of text and wanted to do something with it?
- Great! That's **Data Wrangling**
- Adapting data from one format to another, until you end up with exactly what you wanted.

# 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.

# Basic Data Wrangling (1/2)

Linux:

```
cat /var/log/sys*log | grep -i "Sep 22"
```

- This is an example of basic data wrangling: finding all system log entries that mentions Intel
- Most of data wrangling is just about knowing what tools you have, and how to combine them.
- Remember **The Unix Philosophy!**



# Basic Data Wrangling (2/2)

- Let's start from the beginning:
  1. We need a data source
  2. Something to do with it.
- A good use case is for logs, because you often want to investigate them, but reading the whole thing is not feasible.

# Data Wrangling Example (1/2)

Let's try to figure out who is trying to log into my server.

- First, I try to look into my server's log:  
`cat log`
- That's far too much stuffs!
- Let's limit it to `ssh` stuffs:  
`cat log | grep sshd`
- That is still way more stuffs than what we wanted, and it's pretty hard to read.

## Data Wrangling Example (2/2)

We can do better!

```
cat log  
| grep sshd  
| grep "Accepted publickey for"
```

There's still a lot of noise here.

There are *a lot* of ways to get rid of that, but let's look at one of the most powerful tools in your toolkit: **sed**.

# Exploring Codebase

# Where are we?

Introduction

sed and Regular Expression (regex)

More Advanced Data Wrangling

Exercises

Conclusion

# sed? Isn't that the adjective to describe my life?

- `sed` is a stream editor that builds on top of the old `ed`<sup>2</sup> editor
- In it, you basically give short commands for how to modify the file.
- If you use `vim`, you should be familiar with some of the commands (`ed` -> `vi` -> `vim`)
- There are tonnes of commands, but the most common one is `s` for substitution.

---

<sup>2</sup>If you're into lame computing jokes, here's a joke about `ed`:  
<https://www.gnu.org/fun/jokes/ed-msg.html>

# Back to Our Example

```
cat log  
| grep sshd  
| grep "Accepted publickey for"  
| sed 's/.*Accepted publickey for //'
```

- Wow! It's a lot cleaner.
- What we just wrote was a simple **Regular Expression**

# The s Command in sed

Syntax: `s/REGEX/SUBSTITUTION/`

- **REGEX** is the regular expression you want to search for.
- **SUBSTITUTION** is the text you want to substitute matching text with.



# What is Regular Expression

- It's a powerful construct that lets you match text against patterns.
- They are common and useful enough that it's worthwhile to take some time to understand how they work.
- Usually (though not always) surrounded by `/`
- Most characters just carry their normal meaning, but some characters have special matching behaviour.
- Exactly which characters do what vary somewhat between different implementations of regular expressions, which is a source of great frustration.

# List of Regex Special Characters

Character	Meaning
.	Any single character except newline
*	Zero or more of the preceding match
?	One or more of the preceding match
[abc]	Any one character of <b>a</b> , <b>b</b> , and <b>c</b>
(RX1 RX2)	Either something that matches <b>RX1</b> or <b>RX2</b>
^	The start of the line
\$	The end of the line

If you are unfamiliar with regex, there is a nice tutorial at <https://regexone.com/>

# Obsolete vs Modern Regex

- Note that **sed**'s regex is somewhat weird and will require you to put a `\` before most of these to give them special meaning.
- This is because by default **sed** is using the *obsolete* regex format.
- You can avoid this problem by passing `-E` flag to **sed**, which tells it to switch to the *modern* regex format.
- You can explore the differences by running `man re_format`

# Looking at our regex just now

`/. *Accepted publickey for /`

- It means any text that starts with any number of characters, followed by the literal string "Accepted publickey for "
- However, regexes are tricky.
- What if the username is also "Accepted publickey for "?
- Why? By default, `*` and `+` are “greedy” – they will match as much text as they can

# Solution: Match the whole line

```
| sed -E 's/.*Accepted publickey for (.*?) from  
↪ ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})  
↪ port ([0-9]+) ssh2: RSA SHA256:.*//'
```

Let's look at what's going on with a regex debugger<sup>3</sup>

---

<sup>3</sup><https://regex101.com/r/wPc8Ii/3>

# Explanation

- The start is still as before.
- Then on any string of characters (username).
- Then on **from** followed by an IP address<sup>4</sup>
- Then on **port** followed by a sequence of digits.
- Finally, we try to match on the suffix **ssh2: RSA SHA256:** followed by any string of characters.
- Notice that with this technique, a username of **Accepted publickey for** will not confuse us anymore. Can you see why?

---

<sup>4</sup>This matches **999.999.999.999** which is not a valid IPv4 address.  
A regex that only matches valid address is left as an exercise

# Capture Groups

- Oh no, the entire log is now empty.
- We want to keep the username
- Use **Capture Groups**!
- Any text matched by a regex surrounded by parentheses is stored in a numbered capture group.
- Capture group 0 is special. It is the whole text matched by the regex.
- These are available in the **SUBSTITUTION**<sup>5</sup> as `\1`, `\2`, `\3`, etc.

---

<sup>5</sup>In some engines, even in the pattern itself!

# Using Capture Groups in sed

```
| sed -E 's/.*Accepted publickey for (.*?) from  
↪ ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3})  
↪ port ([0-9]+) ssh2: RSA SHA256:.*\/\1/'
```

- Note that in our current regex, capture group 1 is username, capture group 2 is IP address, capture group 3 is port number.
- You can try out using `\2` and `\3` instead of `\1`.



# More on Regular Expressions

- As you can probably imagine, you can come up with *really* complicated regex.
- For example, there is an article on how you might match an email address<sup>6</sup>. It's not easy<sup>7</sup>. People have even written tests<sup>8</sup> and test matrices<sup>9</sup>
- Regular expressions are notoriously hard to get right, but they are also very handy to have in your toolbox!

---

<sup>6</sup><https://www.regular-expressions.info/email.html>

<sup>7</sup><http://emailregex.com/>

<sup>8</sup><https://fightingforalostcause.net/content/misc/2006/compare-email-regex.php>

<sup>9</sup><https://mathiasbynens.be/demo/url-regex>

# More Regex Trivia

- You can check for prime numbers using regex<sup>10</sup>
- You can match  $A \ B \ C$  where  $A + B = C$ <sup>11</sup>
- You can match nested brackets, e.g. to parse Lisp's s-expressions using Regex<sup>12</sup>
- Note: these are more for curiosity purposes. There are usually better tools than regex, although for a quick and dirty script, regex is usually enough.

<sup>10</sup><https://www.noulakaz.net/2007/03/18/>

[a-regular-expression-to-check-for-prime-numbers/](#)

<sup>11</sup><http://www.drregex.com/2018/11/>

[how-to-match-b-c-where-abc-beast-reborn.html](#)

<sup>12</sup><http://www.drregex.com/2017/11/>

[match-nested-brackets-with-regex-new.html](#)

# Back to Data Wrangling

So now we have

```
cat log  
| grep sshd  
| grep "Accepted publickey for"  
| sed -E 's/.*Accepted publickey for (.*?) from  
↪ ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}  
↪ port ([0-9]+) ssh2: RSA SHA256:.*\/\1/'
```

# sed All the Way!

But we can do everything just with sed!

```
cat log
```

```
| sed -E -e '/Accepted publickey for/!d' -e  
↪ 's/.*Accepted publickey for (.*) from  
↪ ([0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}\.[0-9]{1,3}  
↪ port ([0-9]+) ssh2: RSA SHA256:.*\/\1/'
```

- **d** is to delete, **!** is to apply the function to the lines **not** selected by the pattern.
- Check out `man sed`!

# Where are we?

Introduction

sed and Regular Expression (regex)

More Advanced Data Wrangling

Exercises

Conclusion

# Let's look for common usernames

```
| sort | uniq -c
```

- `sort` will, well, sort its input.
- `uniq -c` will collapse consecutive lines that are the same into a single line, prefixed with a count of the number of occurrences.

# How about the most common logins?

```
| sort -nk1,1 | tail -n3
```

- `-n` sorts in numeric (instead of lexicographic) order
- `-k1` means sort only by the first whitespace separated column
- `,n` means sort until the `n`th field, where the default is the end of the line<sup>13</sup>.
- **Exercise:** what if we wanted the least common ones?

---

<sup>13</sup>In this example, sorting by the whole line wouldn't matter

# How about the most common logins?

```
| sort -nk1,1 | tail -n3
```

- `-n` sorts in numeric (instead of lexicographic) order
- `-k1` means sort only by the first whitespace separated column
- `,n` means sort until the `n`th field, where the default is the end of the line<sup>13</sup>.
- **Exercise:** what if we wanted the least common ones?
- Either use `head` instead of `tail` or use `sort -r` which sorts in reverse order.

---

<sup>13</sup>In this example, sorting by the whole line wouldn't matter



# We can do better

Okay, so that's pretty cool, but we'd sort of like to only give the usernames, and maybe not one per line?

```
| awk '{print $2}' | paste -sd, -
```

Let's start with **paste**

- It lets you combine lines (**-s**) by a given single-character delimiter (**-d**), and ask it to read from **STDIN** (**-**)<sup>14</sup>
- You can also emulate this using **tr '\n' ','**, but this results in a trailing comma.

<sup>14</sup>Using GNU **paste**, the **-** can be omitted, but this is not POSIX compliant.

# awk

- A programming language that happens to be really good at processing text streams.
- There is *a lot* to say about **awk** if you were to learn it properly, but as with many other things here, we'll just go through the basics.

# awk Syntax

- Basic **awk** syntax: `pattern { block }`
- **awk** takes in an optional pattern plus a block saying what to do if the pattern matches a given line.
- The default pattern (if no pattern is provided) matches all lines.
- Inside the block, `$0` is set to the entire line's content, and `$1` to `$n` is set to the n-th field of that line, when separated by **awk** field separator<sup>15</sup>.

---

<sup>15</sup>whitespace by default, can be changed with `-F`

# Our Use of awk

```
| awk '{print $2}'
```

- So in this case, we're saying that, for every line, print the contents of the second field, which happens to be the username.

# More fancy awk

Let's compute the number of single-use usernames that start with **r** and end with **t**:

```
| awk '$1 == 1 && $2 ~ /^r[^ ]*t$/ { print $2  
↪ }' | wc -l
```

Let's unpack this!

- The pattern means the first field of the line should be equal to **1** (the count from **uniq -c**), and the second field should match the regex.
- The block says to print the second field (username)
- Finally, we count the number of lines in the output with **wc -l**.

# awk as a Programming Language

Remember that **awk** is a programming language, so we can actually not use **wc -l** at all:

```
BEGIN { rows = 0 }  
$1 == 1 && $2 ~ /^r[^]*t$/ { rows += $1 }  
END { print rows }
```

- **BEGIN** is a pattern that matches the start of the input, and **END** matches the end.
- First we initialise the count to 0. The per-line block just adds the count from the first field. Then we print it out at the end.

# Advanced awk

- In fact, we could get rid of **grep** and **sed** entirely, because **awk** can do it all, but that is left as an exercise.
- A good resource to read is <https://backreference.org/2010/02/10/idiomatic-awk/>

# We can do Maths too!

```
| awk '{print $1}'  
| paste -sd+ -  
| bc
```

- **bc** is actually a calculator language.
- You can even run it straight from your shell and use it as a normal calculator.
- In this case, we are piping a mathematical expression to **bc**



# Data Wrangling to Make Arguments (1/2)

- **Exercise:** find out what the `xargs` tool does (hint: try to pipe to `xargs echo`)

# Data Wrangling to Make Arguments (1/2)

- **Exercise:** find out what the `xargs` tool does (hint: try to pipe to `xargs echo`)
- Since we can pipe data to it, we can use data wrangling to make arguments too.
- Say we want to delete all files that matches the regex `asd.a [0-9]{2}`

```
ls | grep -E 'asd.a [0-9]{2}' | xargs rm
```

What happened?

## Data Wrangling to Make Arguments (2/2)

- It's the annoying whitespace splitting again.
- A workaround is to use the null character (`\0`) as delimiter instead

```
ls  
| grep -E 'asd.a [0-9]{2}'  
| tr '\n' '\0'  
| xargs -0 rm
```

# Exercises (1/2)

- How is `sed s/REGEX/SUBSTITUTION/g` different from regular `sed`? What about `/i`?
- To do in-place substitution it is quite tempting to do something like `sed s/REGEX/SUBSTITUTION/input.txt > input.txt`. However this is a bad idea, why? Is this particular to `sed`?

## Exercises (2/2)

- Find the number of words (in `/usr/share/dict/words`) that contain at least three **as** and don't have **ness** ending.
- What are the three most common last two letters of those words?
- How many unique two-letter combinations are there?
- And for a challenge: which combinations do not occur?

# Where are we?

Introduction

sed and Regular Expression (regex)

More Advanced Data Wrangling

Exercises

Conclusion

# Talk to us!

- Feedback form: <https://bit.ly/2020ht3>
- Upcoming Hacker Tools:  
Tue, 6th Oct 2020, 6.30pm – Web browser  
customisation, online security, and privacy