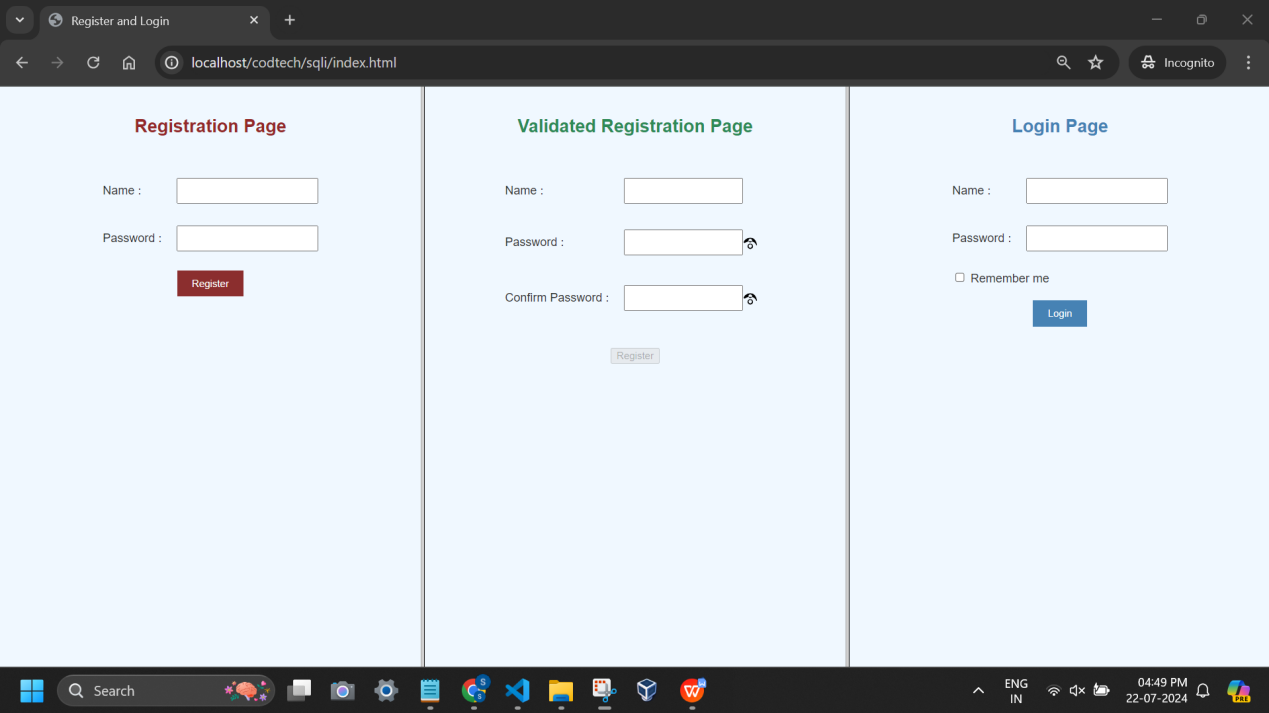
Ref: <https://owasp.org/www-project-top-ten/> 

In a website where deprecated features like framesets are used, it's not safe. Note: The links in the actual code and the screenshots may vary a bit, as through development, the directory structure was modified for simplicity.

This website is similar to DVWA but with more vulnerability types and realistic approaches, and without the “low, medium, high” levels of security. Solutions for mitigation of the vulnerabilities have been described here too.

Landing page: 

1. **SQL Injection (SQLi) : <http://localhost/codtech/signup-signin/index.html>**

Index.html has register.html (executes insert.php) on its left and login.html (executes check.php). To run the code, put it in the htdocs folder of xampp and use this url as per my dir structure:

<http://localhost/codtech/signup-signin/index.html>

(make sure that in both the php files i.e. the insert and check ones, the hashing method used is same. Otherwise the pswds wont ever match even if theyre same). After registering, Login with those creds. A table only with the entered login creds will be displayed in the table. If in check.php:

$uname=mysqli\_real\_escape\_string($conn,trim($\_POST['username']));

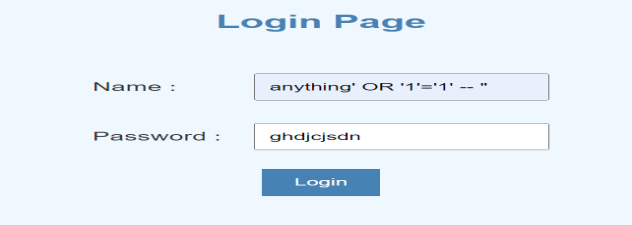
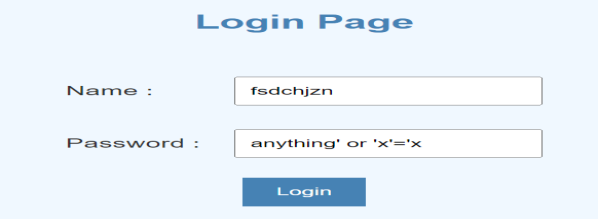
$pass=mysqli\_real\_escape\_string($conn,trim($\_POST['password']));

is used, sqli method will replace " and ' with \" and \' (Preceed with a forward slash). This prevents code injection

$uname=$\_POST['username'];

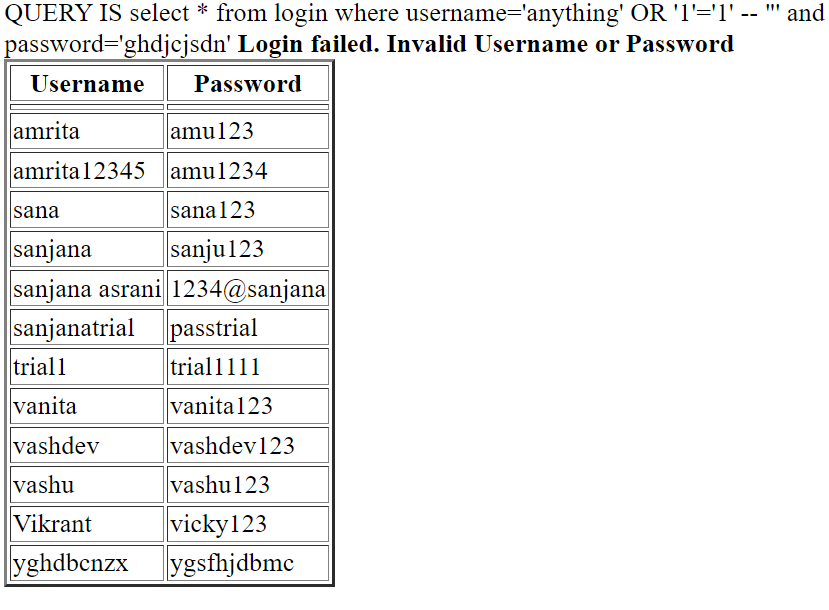
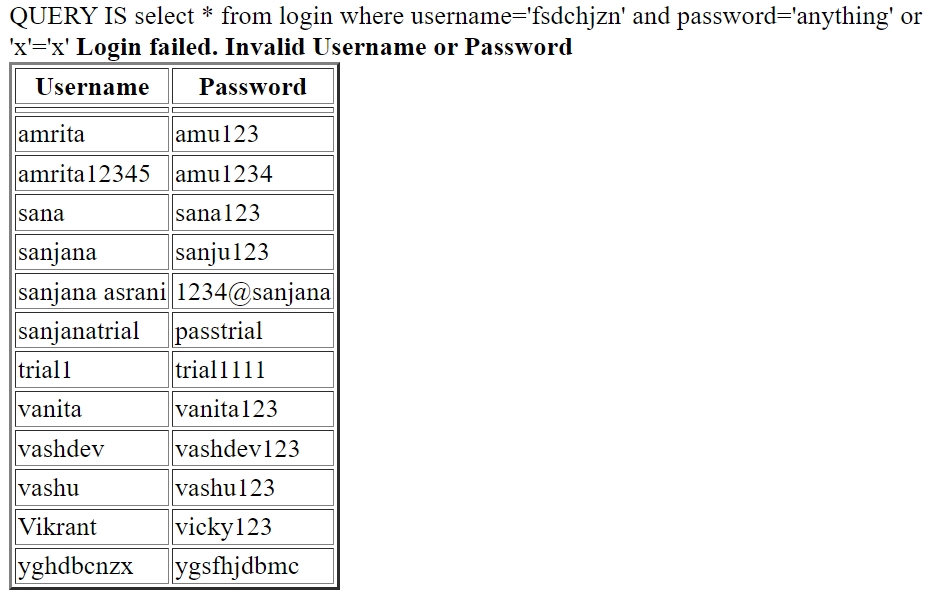
$pass=$\_POST['password'];

If above code is used, this can happen:

-- is an sql comment. The query post it is ignored. The input ' OR '1'='1 modifies the SQL query to always be true, effectively logging in without a valid username/password.

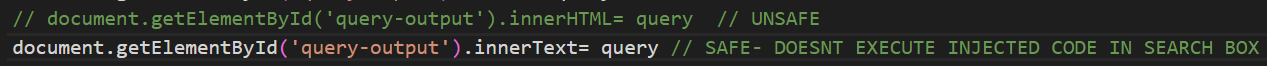
* first ac in db will be logged in for us if the name is filled as ' OR 1=1
* log me using this specific email address only sanjanavasrani' and 1=1;

**NOTE: the sqli in pswd field will not work if it being hashed before comparison with original pswd (in the database). CONCLUSION:** Do not use user input directly in SQL queries. Always sanitize them.

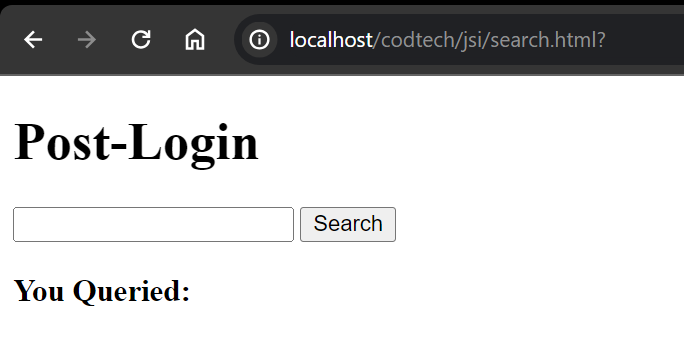
1. **Javascript injection: http://localhost/codtech/post-login/dashboard.php**

**If the input fields render and treat the input values as they are, and if any html or js is inputted in it, it will be rendered as code (executed) as html or script. Hence, the innerHtml method is used. This is needed when dynamic html is to be executed. Else, innerText method is used**

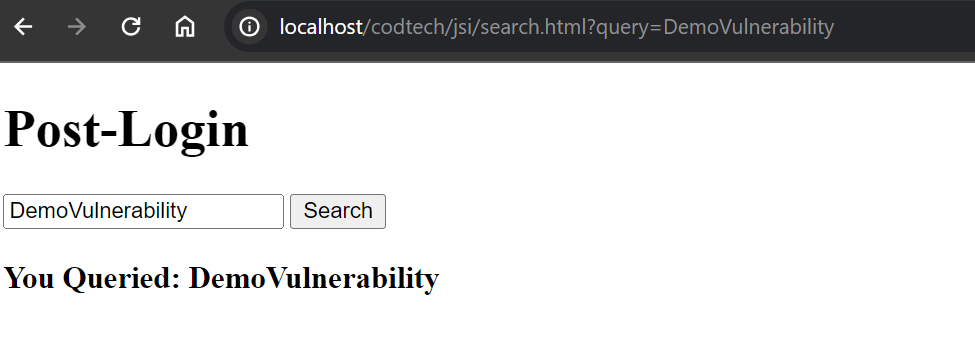


A successful login is done: 

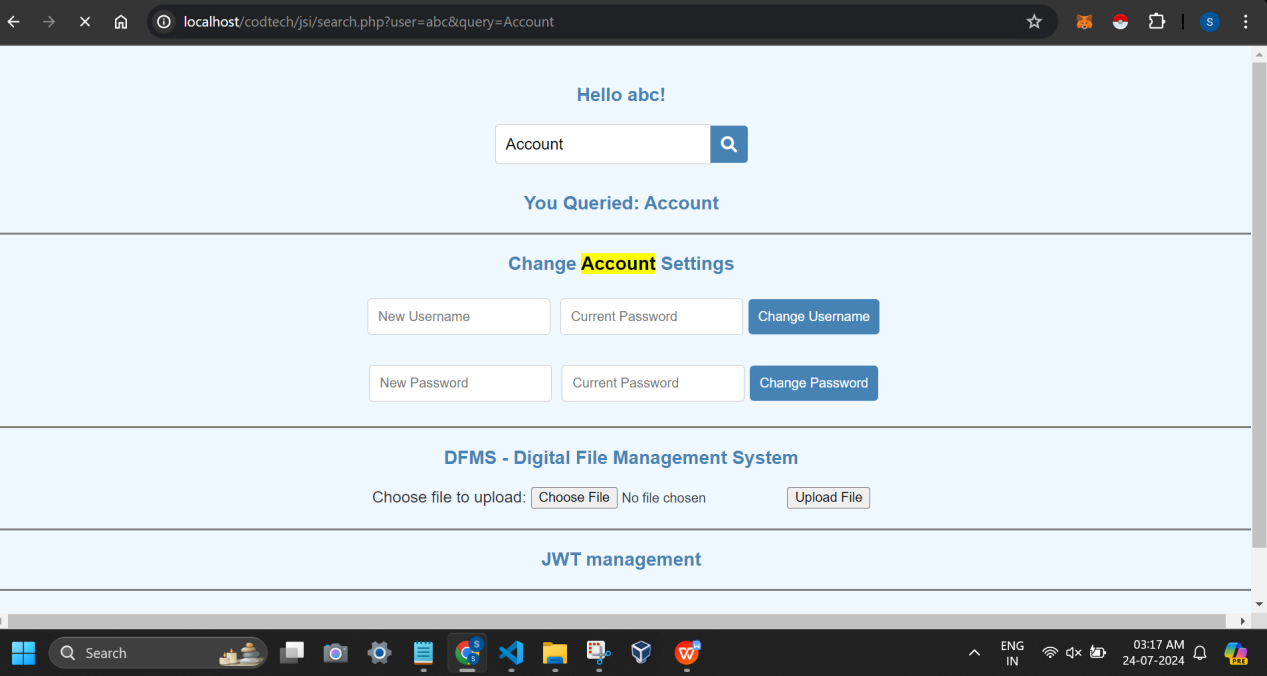
the url doesn’t get the parameters by itself eg. ?Username=sanjana&pswd=sanju123 (SINCE ITS LOGIN PROCESS, IT’S A POST METHOD, DATA IS SENT IN REQUEST BODY NOT IN URL)

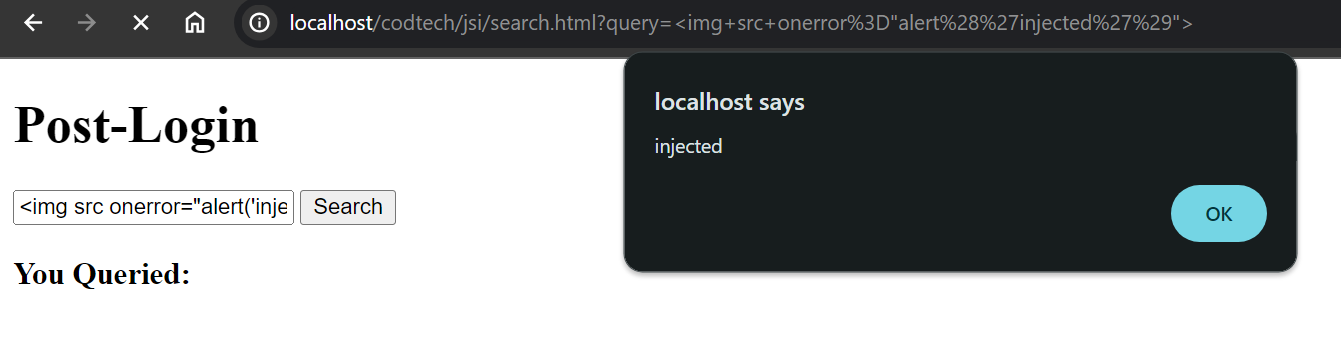
Now this page gets opened: 

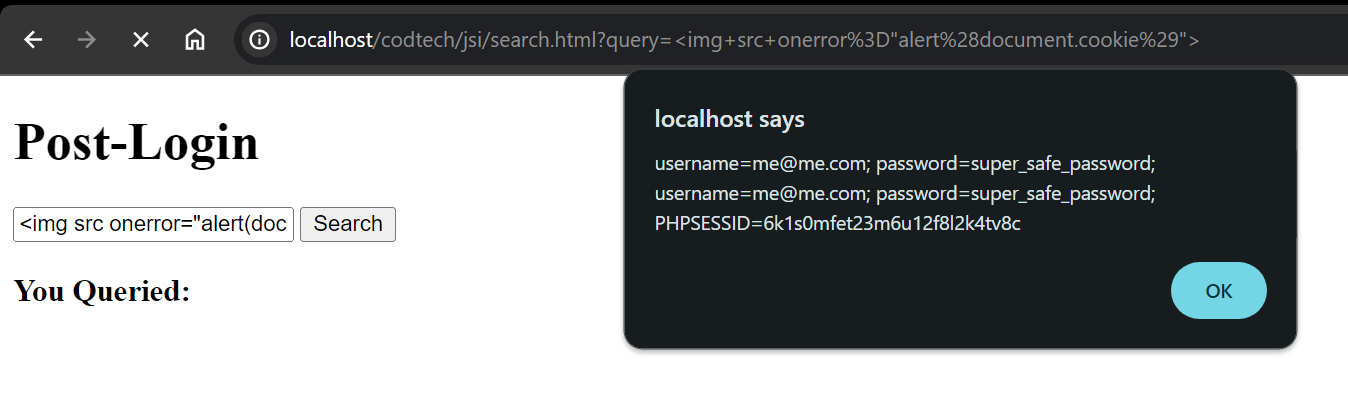
the parameters in url do get added after login as it now becomes a GET REQUEST since this is a search query service request from server, data is sent over url unlike in POST.



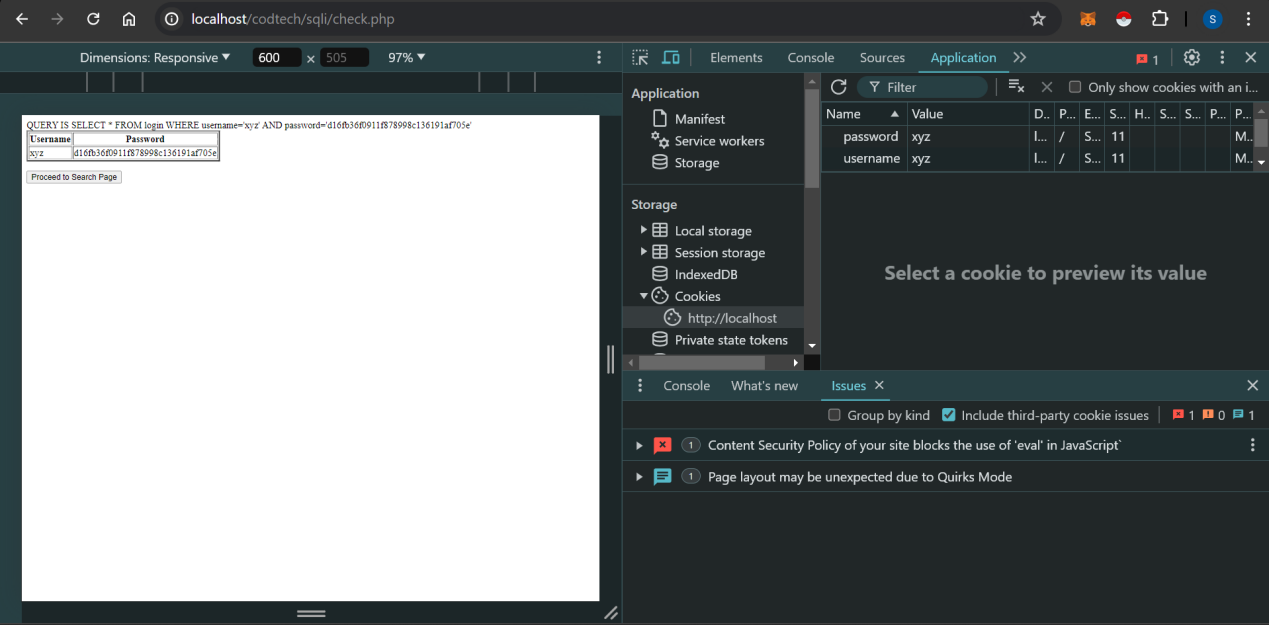
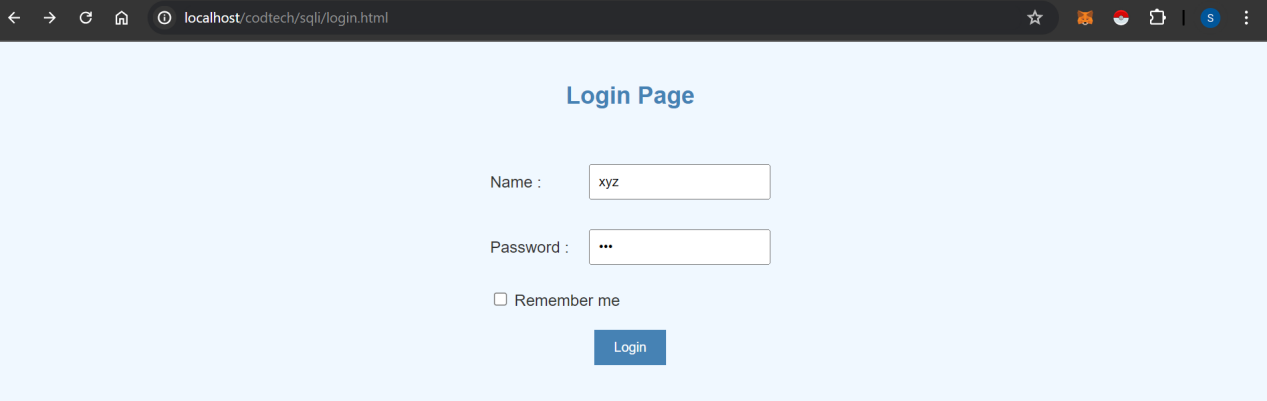
So now, when innerHtml is used, if a js code is put in the search box, the url is injected with the script code as well. So if the url is shared and someone opens them, the injected code will run (the query as it has been executed doesn’t get rendered on the html page, thus empty)



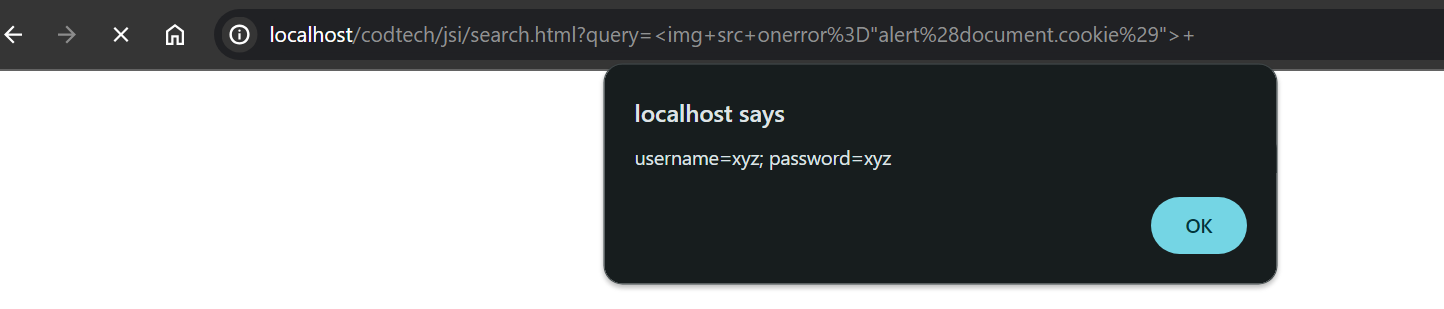
<script>alert('injected');</script> | <img src onerror="alert('injected')"> 

this img tag has an empty source so its bound to throw an error. when the error occurs, we do an alert. <img src onerror = "alert(document.cookie)"> 

As any javascript code can be injected here, a code to email the saved cookies can be run by the attacker. (A session id= username + pswd). If innerText was used, injected code is rendered as string and thus not executed. Thus no alerts here 

In this example, the cookies were hardcoded for demonstration purpose. But for real life scenario, If remember me is clicked on the login page, the username and pswd are stored as cookies: thus allowing a auto-login (the fields get auto filled for login next time) 

(make sure the pswd is hidden and replaced by special characters like \* here, or else it makes it vulnerable to shoulder surfing).

these cookies can be retrieved using js Thus we can store the cookies after hashing To prevent this vulnerability, we can take the following steps:

### ****Use Secure Cookies:****

Ensure that cookies are marked as HttpOnly, Secure, and SameSite.

Demonstration of a safe login Auth:

This is a pswd less login app

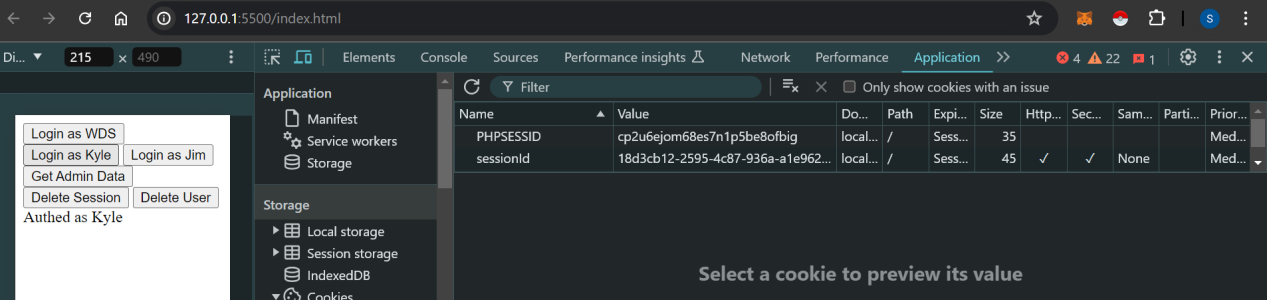
Ref: <https://youtu.be/mL8EuL7jSbg?feature=shared>

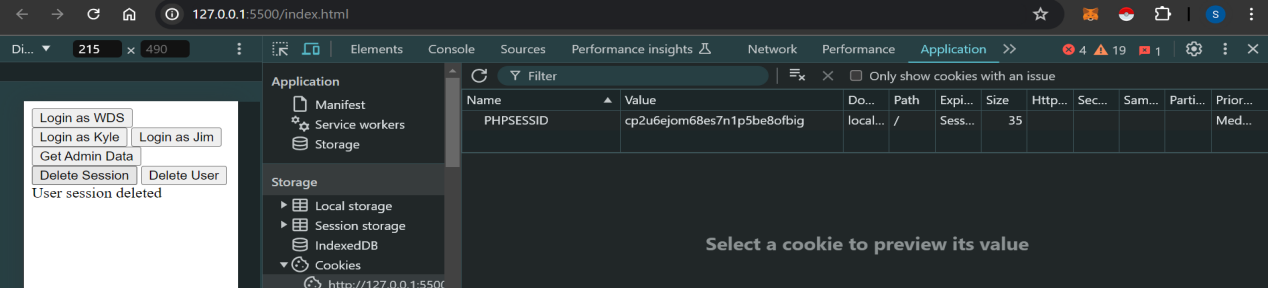
Only wds is admin and can access admin stuff

If kyle accesses it, It gets “authorized”

Jim doesn’t even exist in the map (acting as db)

A new session id is created for every new login (go to inspect -> applications -> cookies)



Deleting a session id: 

Run the prj using these commands: npm install -y

npm i --save-dev nodemon

Install the dependencies: npm install express cors cookie-parser

node server.js

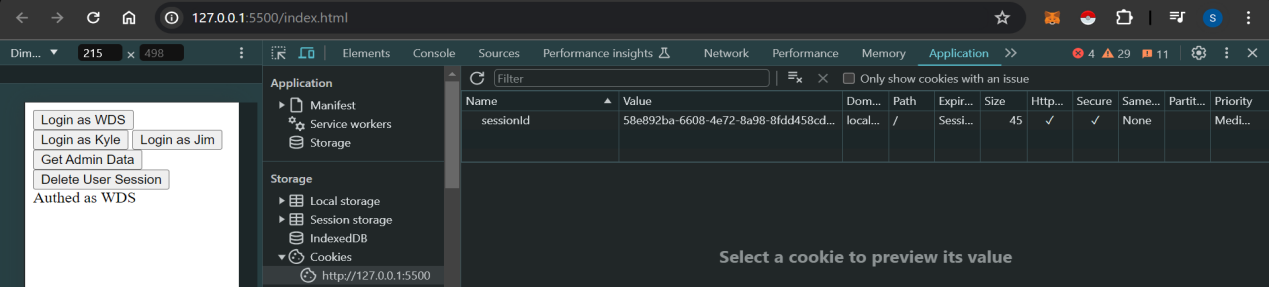
Click on index.html -> open with live server

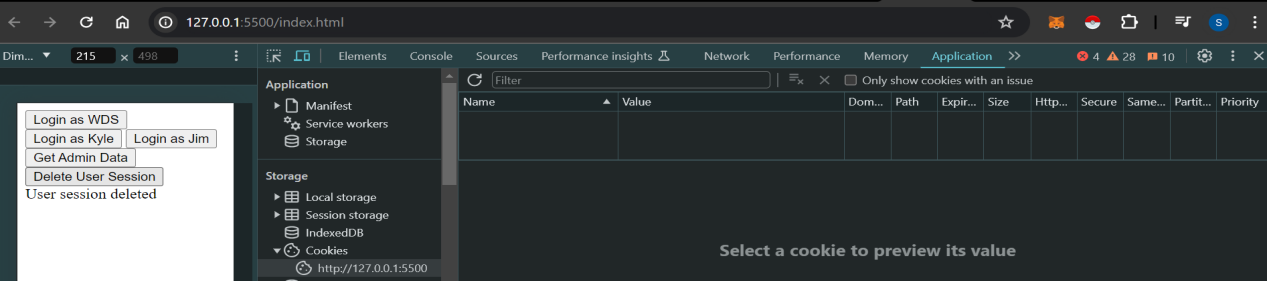
Even if cors is used, http only reqs are used…

Still is vulnerable

npm run runcmd

Deleting a session id:

Before: logged in as admin wds 



Deleting a user let say kyle will result in

“authed as kyle” before and “unauthorized” after.

### ****Hash Passwords:**** Never store plain text passwords. Instead, hash passwords on the server before saving them. Even cookies can be hashed and stored for a much safer option.

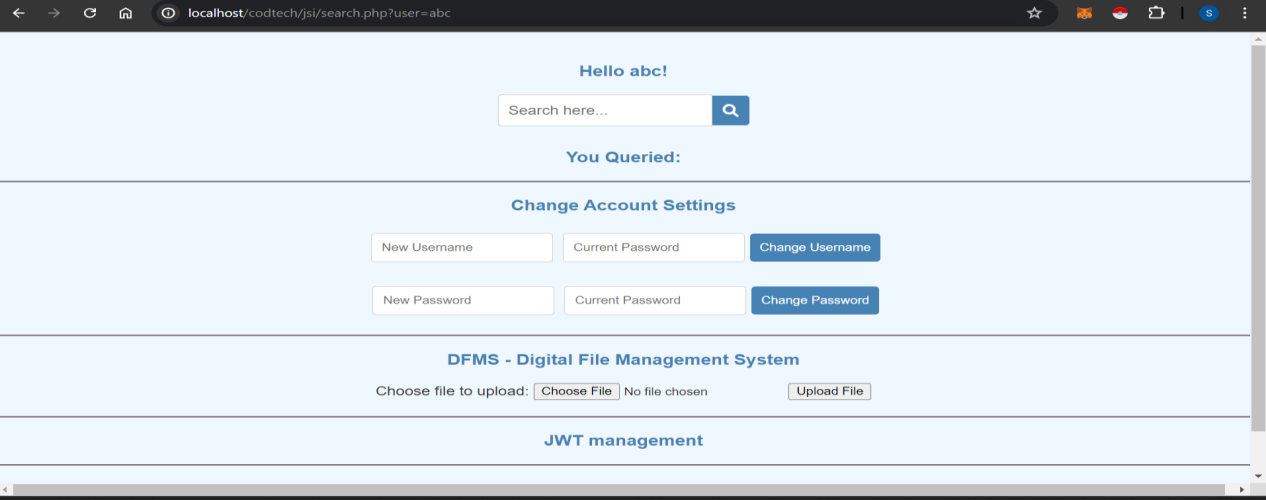
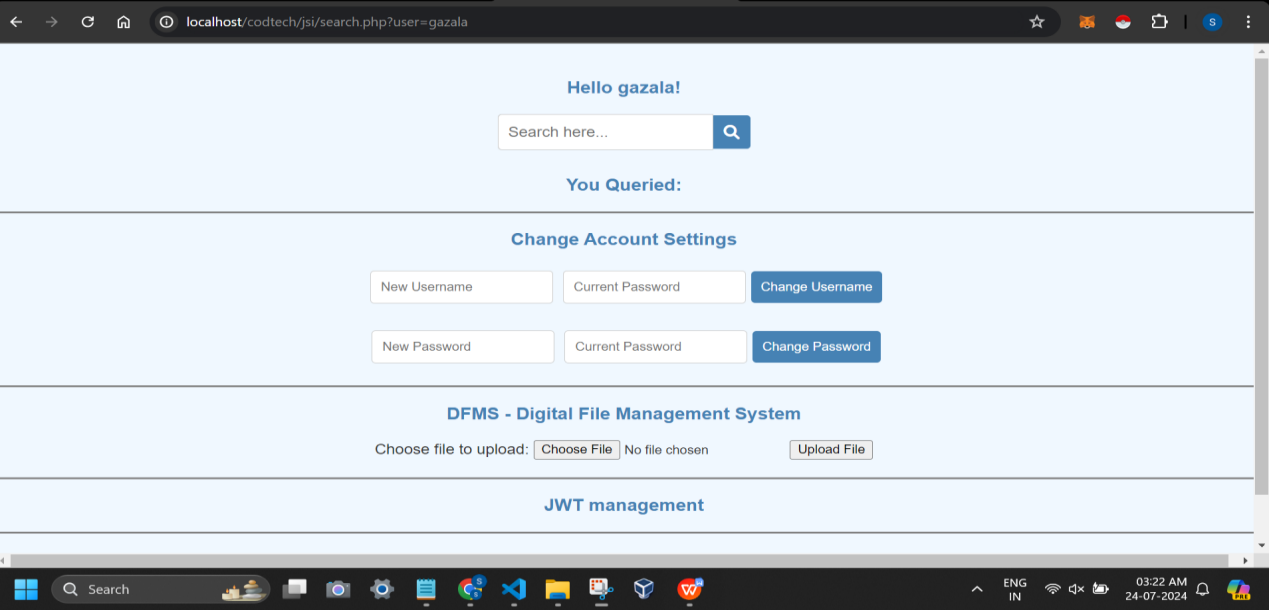
### ****Use Sessions:**** Use sessions to manage user authentication instead of cookies. Store session identifiers in secure, HttpOnly cookies. Sessions die after a period of time. if an application uses predictable session IDs or lacks proper access controls, a regular user can manipulate URLs to gain access to restricted areas like the admin dashboard.

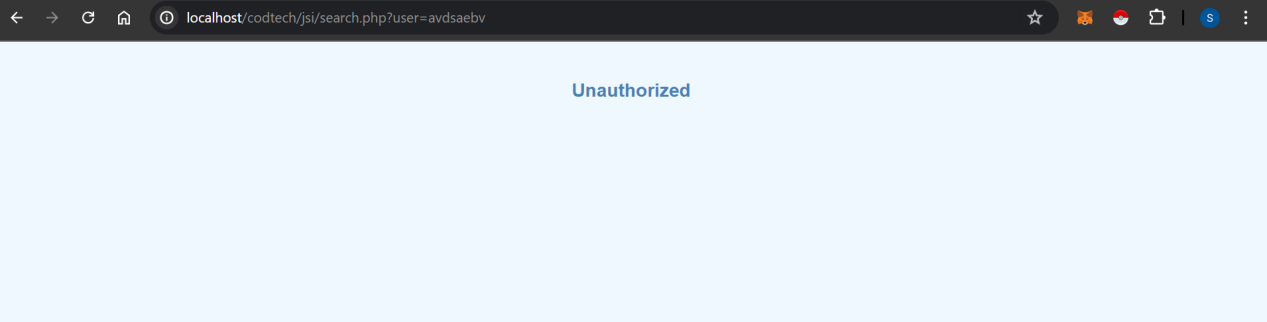
### **Implement Strong Authentication Mechanisms:** Use multi-factor authentication (MFA) to enhance security.

### Short-lived Tokens: Use short-lived tokens (e.g., JWTs) for authentication and refresh them regularly.

### Properly escape and sanitize user inputs, use Content Security Policy (**CSP**), and validate and sanitize data both on client and server sides.

1. Insecure Direct Object References IDOR

Only registered and logged in users should be able to access the search.php page as the only way of redirection is through the btn for search.php on check.php. As seen, after login, the username is added to the url as a parameterIf not proper authentication is done, without having to enter a pswd, one can guess usernames and access other accounts:Now one has access, he can change the pswd and lock the original user out permanently, if here as well, a reconfirmation for pswd is not done. While, unregistered users should never be able to access the features:

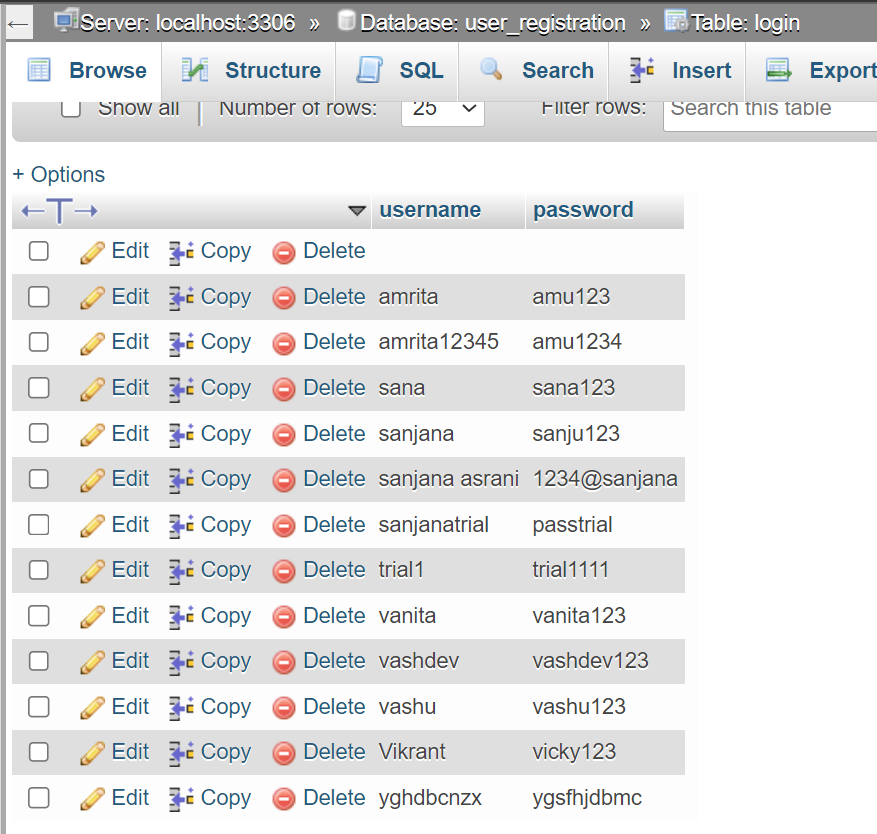


Also, anyone can access this link: <http://localhost/codtech/post-login/upload.php>

because it doesn’t verify the users login… it assumes that if user has access to the url, whose only source is through dashboard which requires a login, this must be okay too.

1. **Insecure Password Storage or Lack of Strong encryption**

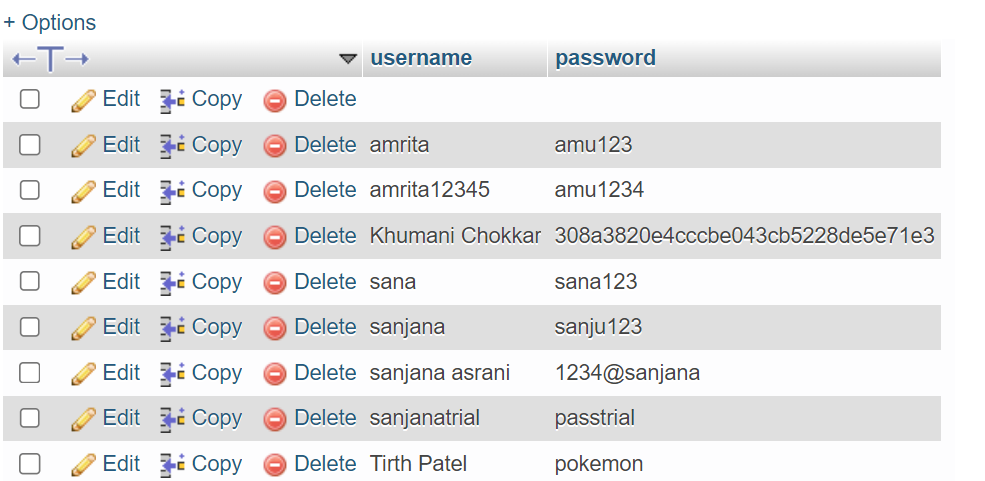
A programmer can attempt to remedy the password management problem by obscuring the password with an encoding function, such as base 64 encoding, but this effort does not adequately protect the password. To defend the application from someone who has access to the configuration, here the database, hashing is required. Passwords are being stored as TEXT in database:

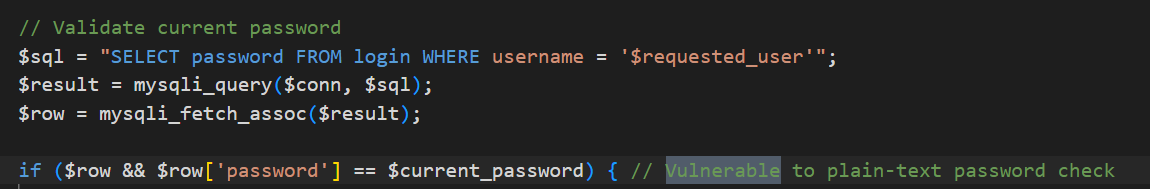


Use a hashing technique for encryption like md5 or sha256: in insert.php, Add a line of code to hash the pswd and then store it in db:

$pass = mysqli\_real\_escape\_string($conn, trim($\_POST['password']));

$pass=md5($pass); //or password\_hash($pass, PASSWORD\_DEFAULT); or sha1($pass);

In New records, hash of entered pswd is compared with the pswd field in db, if hash matches, login is successful:

If pswds were saved as plaintext, this change pswd comparison becomes vulnerable No hashes are being compared.

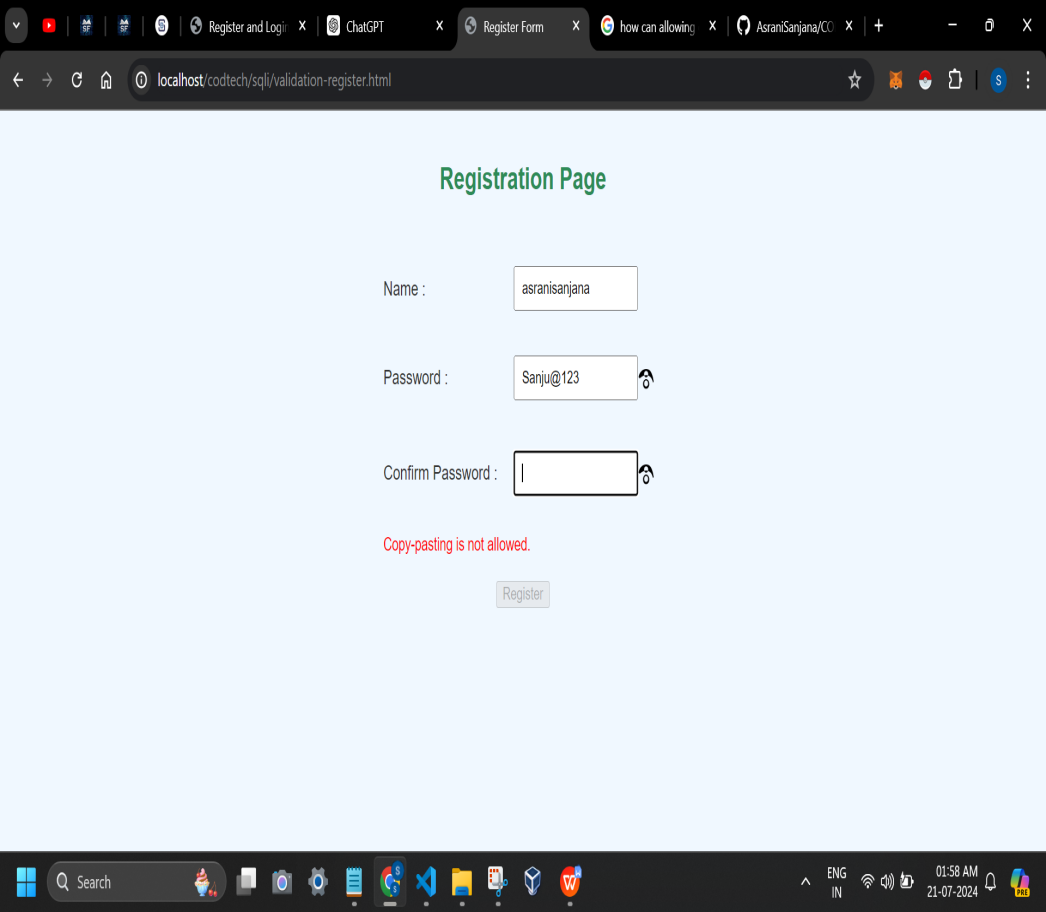
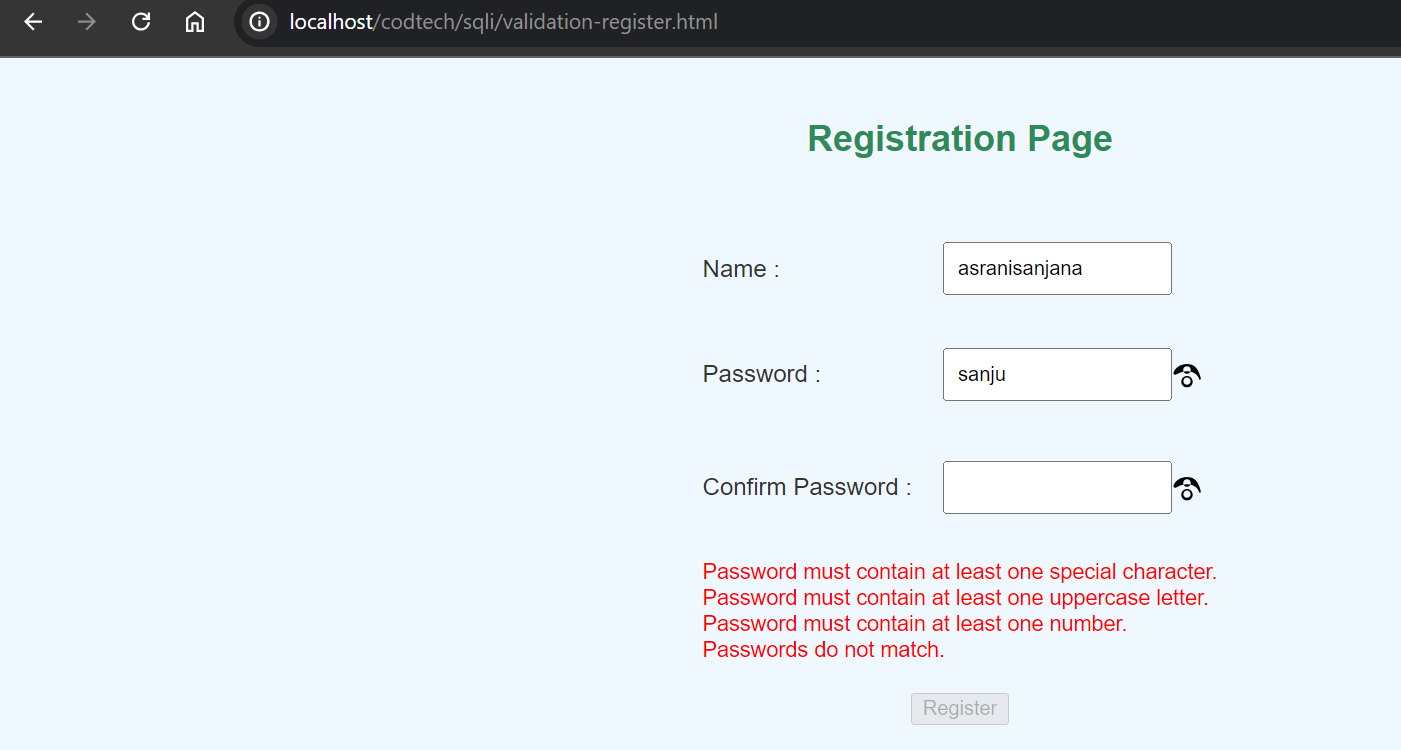
In this way even if someone has access to the database, and can see the pswds, only the hashes of pswds are visible. And as these hashes cant be reversed, login using these hashed pswds is not possible.

**CONCLUSION:** Never store passwords in plain text. Always use hashing functions like password\_hash.

1. **Weak Credentialing Policies**

As for registeration, if it can be seen that there are no validation checks based on strength of a pswd Eg. Pswd must contain special characters, then it allows weak pswds which then lead to a vulnerable access to the system.

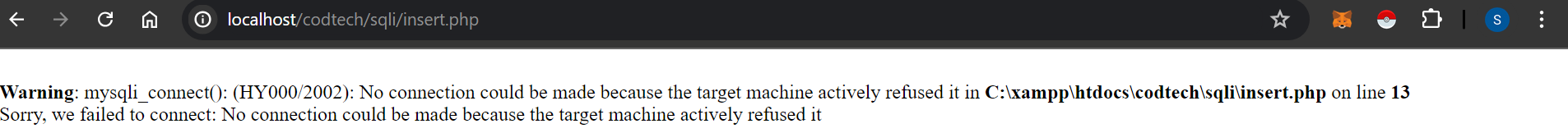
Creating a validation-register.html file to demonstrate correct policy techniques:

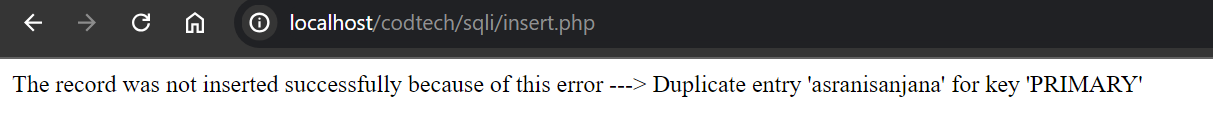


Furthermore, pswd strength checking by scoring its performance during brute forcing, dictionary attacks can be done. Ref: <https://github.com/AsraniSanjana/CODTECH-Task1-password-strength-checker>

Also, if they copy and paste the same password into the Confirm new password field, they might not realize that the password they had in mind was not the password they actually created for their account. As a result, they might end up locked out of their account the next time they try to log in.

Displaying detailed error messages to users can be considered a vulnerability. It's a form of **information disclosure** which can be exploited by attackers. This vulnerability falls under the category of **improper error handling** and **information leakage**.

Detailed error messages can reveal information about the system, such as the database type, server software, or programming framework. This information can help attackers craft more targeted attacks. 

Detailed authentication errors (e.g., "Username not found" or "Incorrect password" or “username already taken”) can help attackers determine valid usernames, making brute force attacks more efficient. One more example of information exposure is when api keys, creds (eg. Sender email for a email service) or db conn pswds are not kept in a .env file but direclty in the code. Hence, Store sensitive information in a .env file or use environment variables to keep them out of the source code and ensure that the .env file is not included in version control. Best Practices to Mitigate This Vulnerability

1. **Use Generic Error Messages:** such as "An error occurred. Please try again."

2. **Log Detailed Errors Internally:** Keep detailed error logs on the server side where only authorized personnel can access them. Ensure that logs include sufficient information for troubleshooting without exposing sensitive data.

1. Implement **rate limiting** to prevent **brute force attacks** and monitor logs for unusual activity patterns, such as repeated failed login attempts.

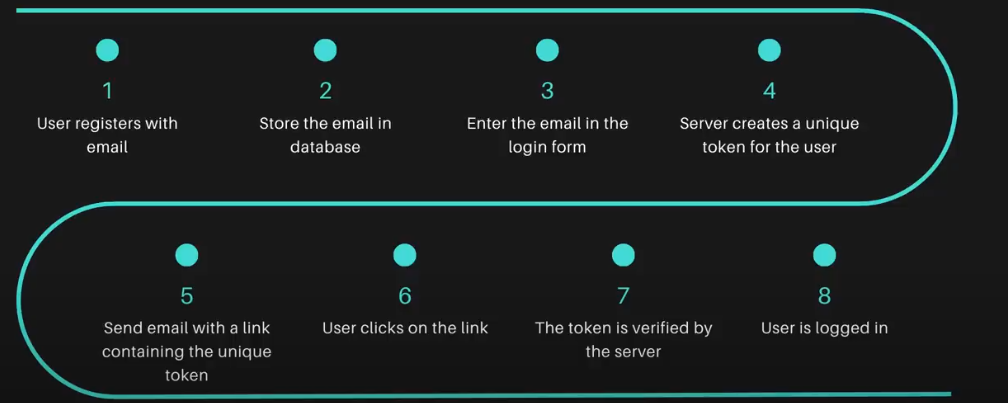
Using Brute forcing, attacker can gain access to website especially if the creds were not strong. Eg. With dictionary attacks, rainbow attacks or even use of social engineering to get demographic information or by shoulder surfing, one can guess the pswd.

1. **Inadequate authentication**

**2fa or mfa… email link activation using a token or sms code otp should be used.**

**(the token should not be weak or predictable)**

**Passwordless auth using email magic link:**



* Pswd stored in cookies accessed through jsi or The “Remember Me” functionality should never store a user’s password on the device. if someone enters that in url and if sessions are not there, anyone with the link even unregistered people can access the resource (solution: **session managemen**t in php, redirection to login page)

### ****Insecure Session Management:****

* **Session Fixation:** If session management is insecure, an attacker could fixate a session ID and use it to impersonate an admin.
* **Session Hijacking:** If session tokens are not securely managed (e.g., through weak or predictable tokens), attackers might steal an admin’s session token and perform actions like deleting users.
* Backdoor: undocumented way to bypass existing cybersecurity measures and gain access to the computer system or device. Software and hardware developers sometimes install backdoors into their own products to retain access for troubleshooting purposes
* social engineering (phishing)
* pswd guessing: brute force dictionary or rainbow table attacks
* deceptive mails
* 2fa or mfa

1. File upload

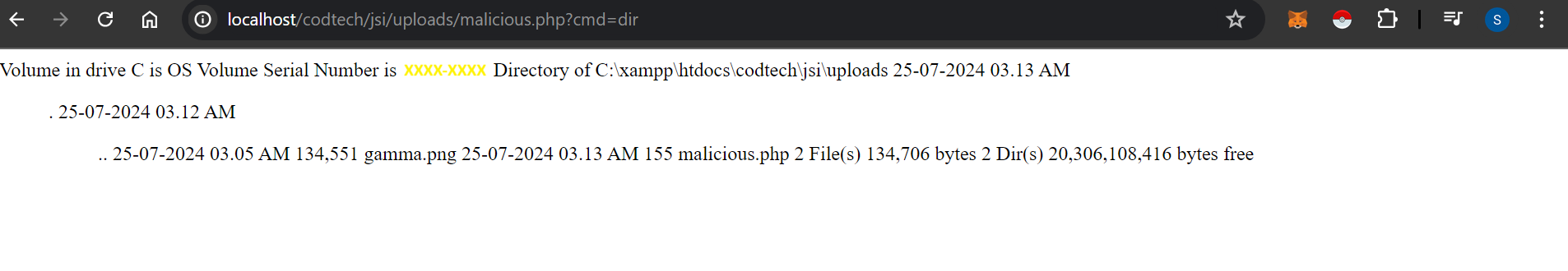
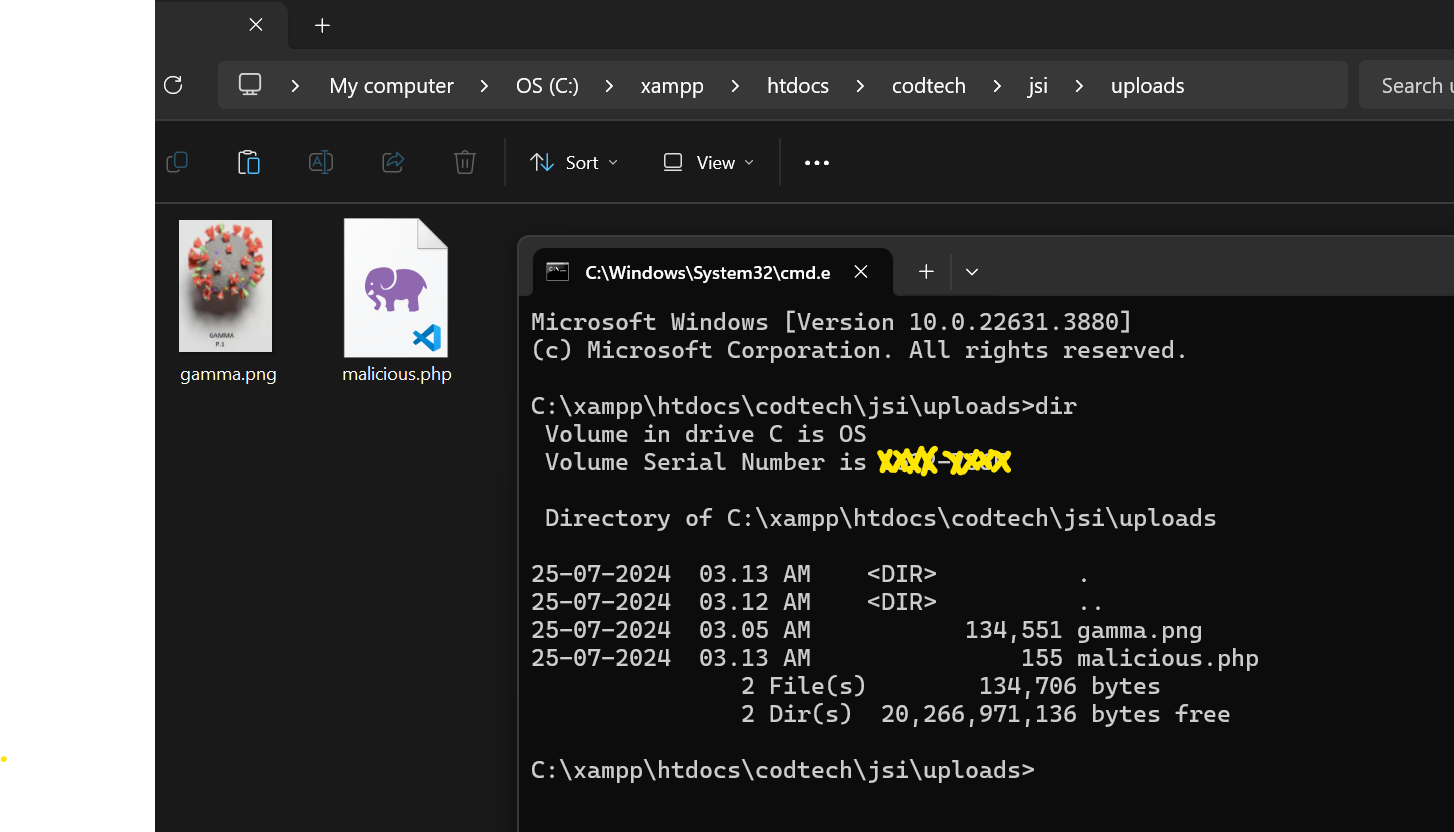
We create a new folder, uploads.. where all the uploaded files reside. Uploading a malicious.php file which code to open the shell or the cmd prompt makes it vulnerable.

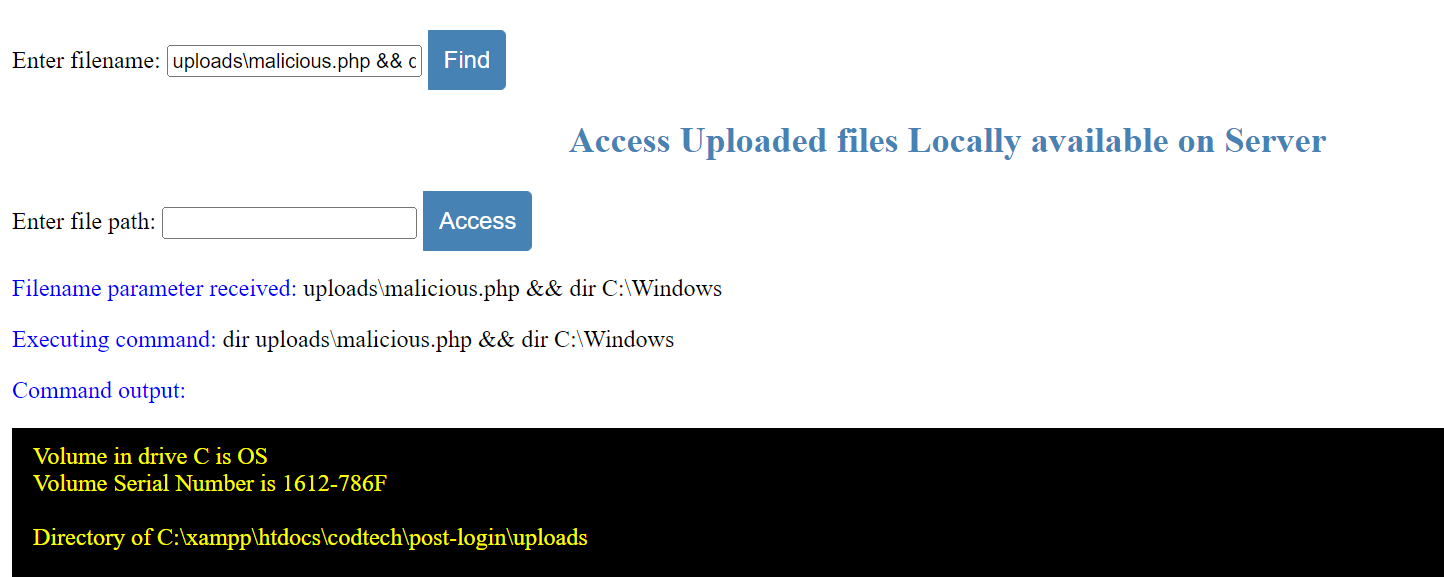
Now since a proper ACCESS CONTROL is not done and insecure file storage is present with read write permissions and priveleges to anyone and everyone, this file can be accessed on the public url and the cmd prompt can be used to exploit the server system where the file resides.

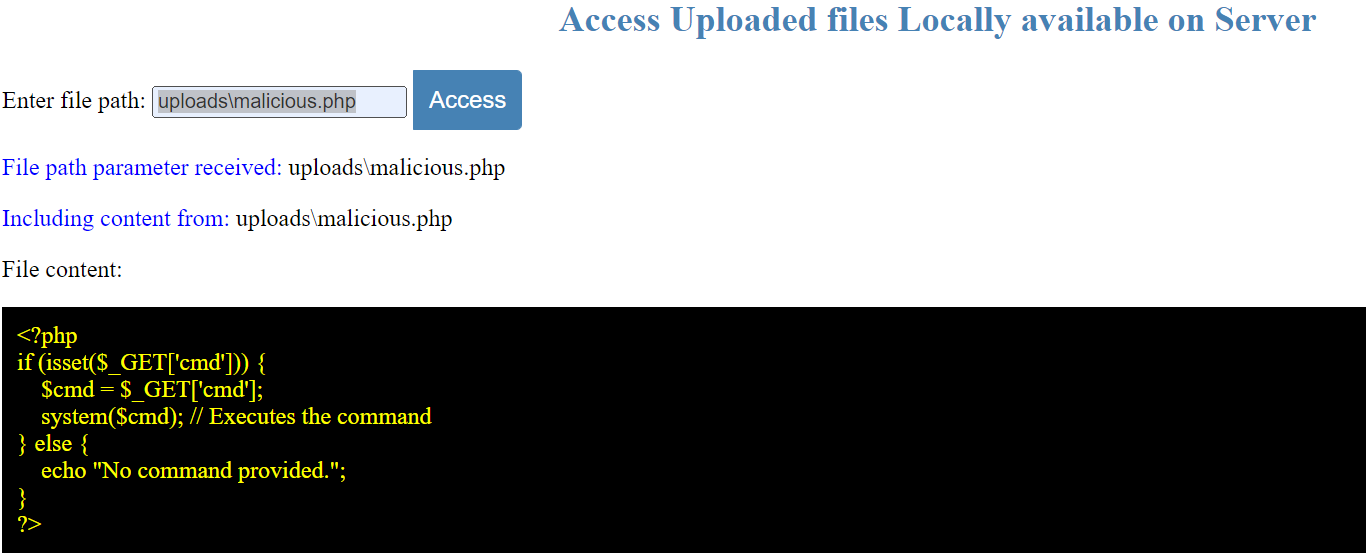
· **File Upload Vulnerability:**· Could lead to the exposure of sensitive data or unauthorized access if the uploaded file contains sensitive information. Risks include file system access, unauthorized viewing or modification of files.

· **Remote Code Execution (RCE):**· Can lead to full server compromise, where an attacker can execute any commands or scripts on the server.

Here my file’s path followed by the cmd to be executed is [http://localhost/codtech/post-login/uploads/malicious.php?cmd=dir](http://localhost/codtech/jsi/uploads/malicious.php?cmd=dir)

Accessing the CMD prompt via a web server allows attackers to execute arbitrary commands, potentially compromising the system. Commands like `del \*`, `format C:`, or `cat /etc/passwd` could delete files, format disks, or access sensitive data, leading to data loss or unauthorized access. The above cmd is opened from the uploads dir where all files are uploaded on server: 

For the next options, One can ip: uploads\task.jpg && dir C:\Windows 

If attackers gain access to the command prompt on a vulnerable server, they could run various commands to gather information about the system and potentially exploit it. if an attacker can upload and execute a VBScript on a vulnerable server, they could potentially gain control over the server or perform malicious actions.

### Prevention Measures

### ****Sanitize Inputs:**** Validate and sanitize all user inputs and uploaded files.

* **File Permissions:** Restrict file upload directories to non-executable.
* **Server Hardening:** Use firewalls, intrusion detection systems, and secure server configurations.

1. **XSS**

If an XSS vulnerability exists on your site, an attacker can inject JavaScript that reads cookies and sends them to a malicious server. If cookies are not flagged as HttpOnly and Secure, they can be intercepted over insecure connections or accessed via JavaScript.This can lead to session hijacking.

### Key Differences between xss & jsi:

· **Reflected XSS** can be exploited by manipulating the URL parameters and sending a crafted URL to the victim. When the victim clicks on the URL, the malicious script gets executed in their browser. This can be used for phishing attacks, session hijacking, and stealing cookies. An attacker sends a URL like http://example.com/xss.php?search\_query=<script>alert('Reflected XSS');</script> to the victim. Opening the url runs the script without any interaction of user. This Often involves cross-site interactions and is particularly concerned with how malicious scripts can affect other users.

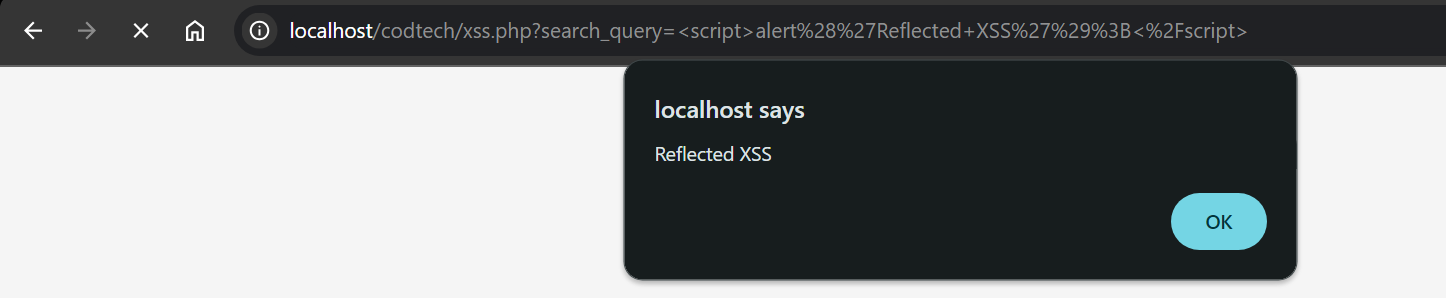
* + · **JavaScript Injection (JSI)** typically requires user interaction to enter the script directly into an input field on the web page, which limits its ability to exploit users through external means like URLs. JSI requires the attacker to input the malicious script directly into the website's input fields, limiting the attack's scope to users who directly interact with the website rather than those who might be tricked by a malicious link. This can be within a single site or application context and may not always involve cross-site concerns but focuses on the ability to inject and execute JavaScript code.
* Types of XSS: Stored XSS, Reflected XSS and DOM-based XSS

1. **reflected or non persistent xss:**

This is reflected in the url and effect of it is immediate.



If echo "<p>Search Results for: " . $query . "</p>"; is used:



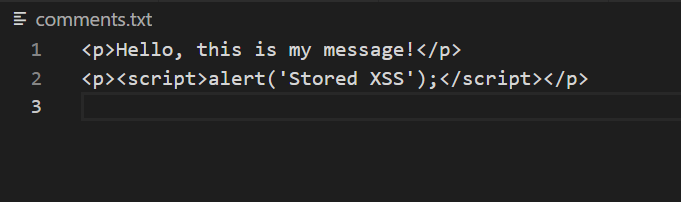
If echo "<p>Search Results for: " . htmlspecialchars($query, ENT\_QUOTES, 'UTF-8') . "</p>"; is used:

(script tag wont execute, no alert on clicking submit):



1. **stored XSS or persistent or second-order XSS**

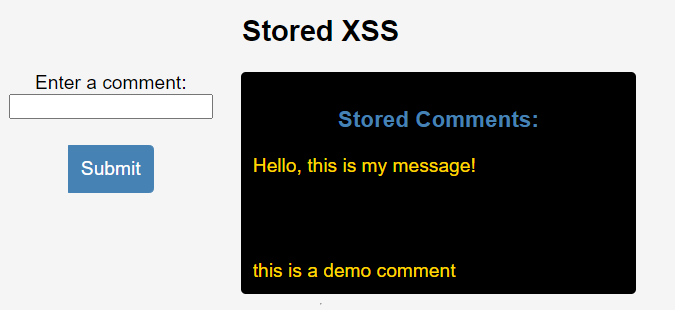
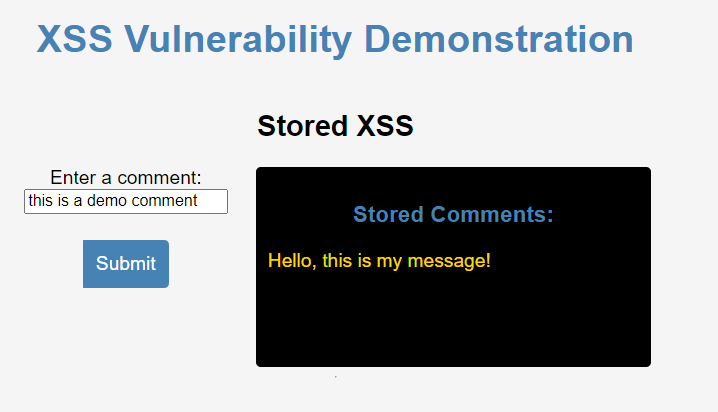
All inputs are stored in the comments.txt



<p>Status: All is well.</p>

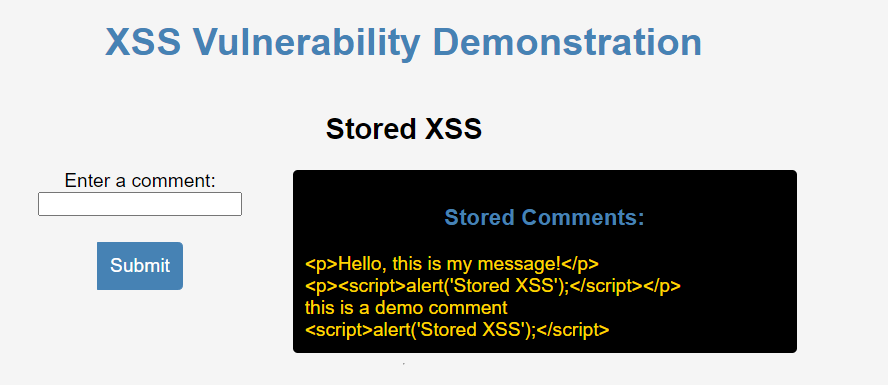
<p>Status: <script>/\* Bad stuff here... \*/</script></p>

The bad stuff can have js code that runs. While displaying these stored comments, it is to be noted that the tags do not get printed as text. Instead theyre rendered. Thus for a script tag, it is actually executed.

If echo nl2br(file\_get\_contents('comments.txt')); is used: SCRIPT IS EXECUTED. 

The empty lines are because the malicious comments with script tag have been executed instead of being rendered as text. These comments will run everytime the page is reloaded, as theyre stored in the comments.txt file.

If *echo nl2br(htmlspecialchars(file\_get\_contents('comments.txt'), ENT\_QUOTES, 'UTF-8'));* is used:

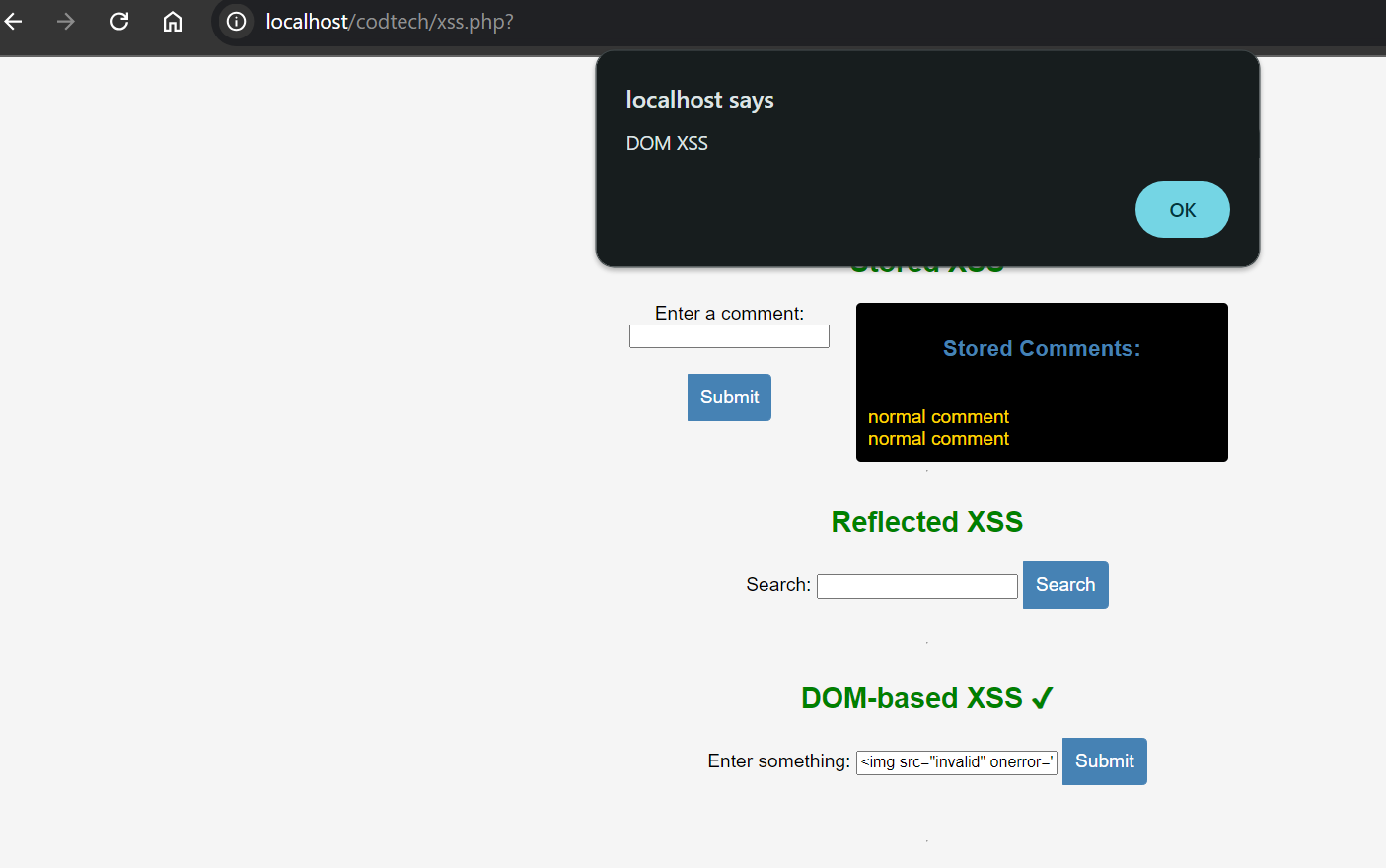
No alerts popped as script tag not executed: 

When using HTML character entities, the comments are stored and rendered as they are. I.e. the script and p tags will be visible too, not just the text

Even &lt;script&gt;alert('Reflected XSS')&lt;/script&gt; will be rendered as <script>…

1. **DOM-based XSS.** ...arises when an application contains some client-side JavaScript that processes data from an untrusted source in an unsafe way, usually by writing the data back to the DOM.

If document.getElementById('dom\_result').innerHTML = userInput; Is used



(Some modern browsers or development environments may automatically sanitize script tags to prevent XSS, so try the below ip instead.)

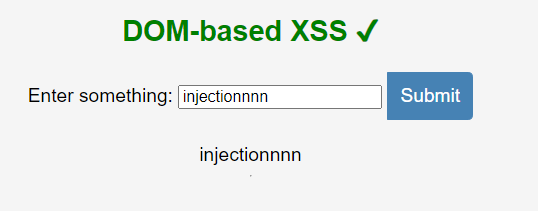
Ip: <img src="invalid" onerror="alert('Reflected XSS')">

If

userInput = document.createTextNode(userInput); //this line sanitizes the input

document.getElementById('dom\_result').appendChild(userInput); //adds it into dom

is used:

This prevents dom xss by not executing the script and treating it as text.

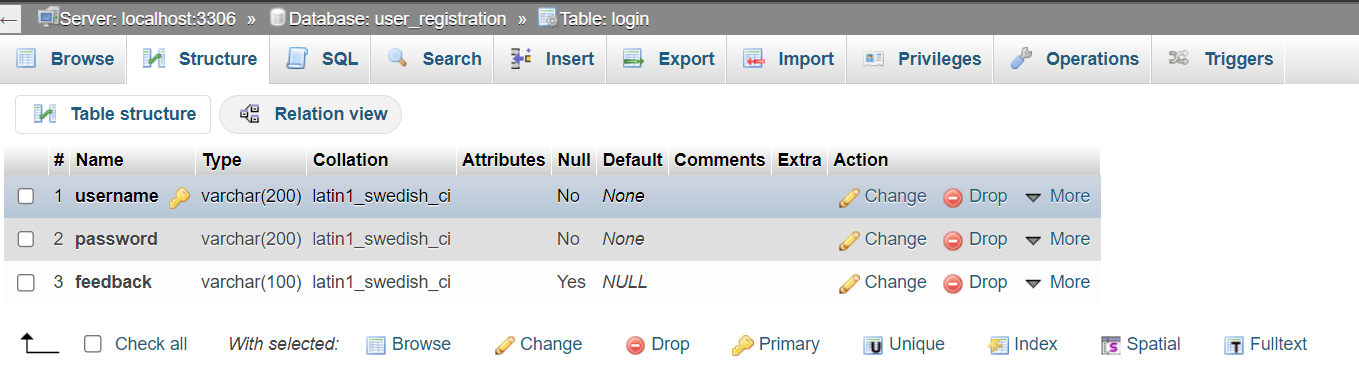
Mitigation: Sanitize user input before injecting it into the DOM. Avoid directly setting HTML content based on user input. **CONCLUSION:**Always escape output when displaying user input.

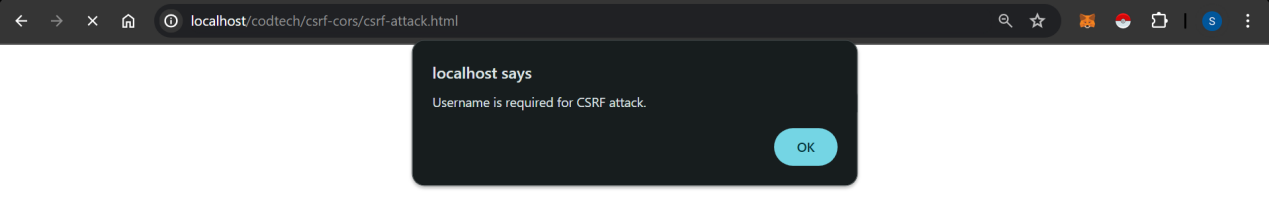
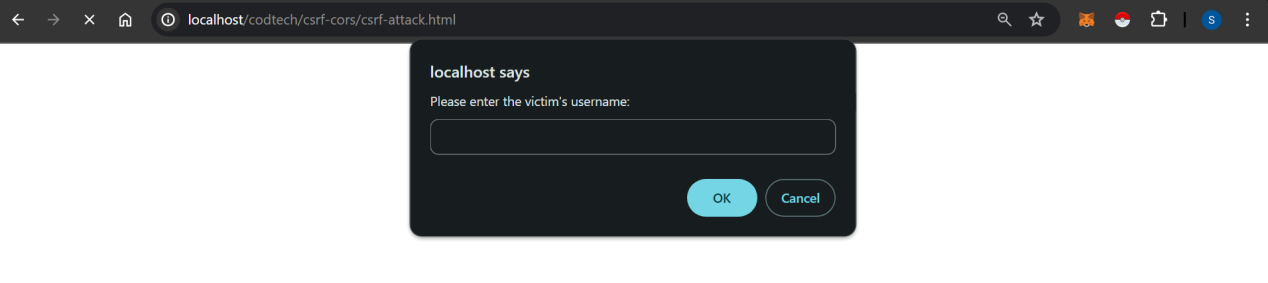
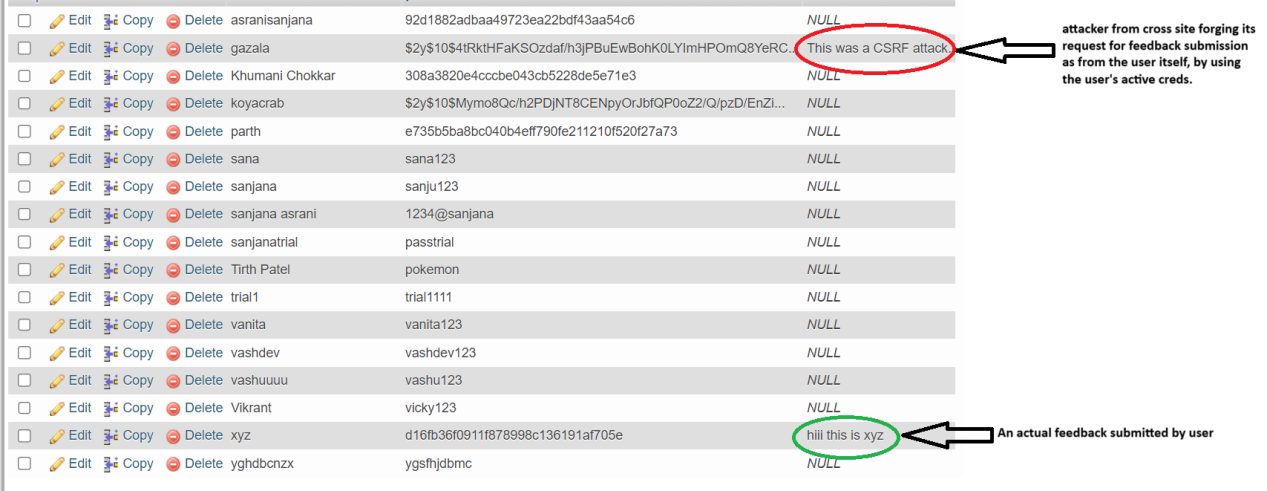
1. **Cross-Site Request Forgery (CSRF)**

**If a session cookie lacks proper CSRF protection, an attacker can craft a request that will be executed in the context of an authenticated user, leading to unauthorized actions. If the attacker manages the user to click on a link (usually done by **clickjacking** or **phishing**), the malicious website can use the existing active session of the user on for eg. A bank website and carry out transactions without having to put in account details.**

**Implementation for csrf:**

**Create a new field in the same db user\_registration with the colmn name as “feedback”**



When cookies are not stored, a username is required (as a parameter in request body to send feedback), which can be manually provided here for demonstration purpose. After opening [http://localhost/codtech/csrf/csrf-attack.html](http://localhost/codtech/csrf-cors/csrf-attack.htm), If cookies were stored, directly feedback is submitted for that username: Result:

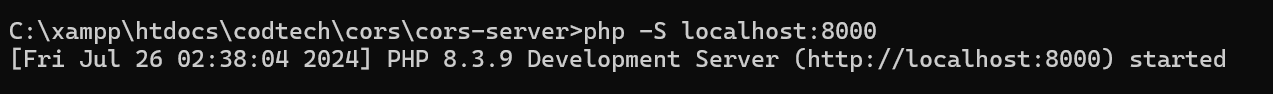
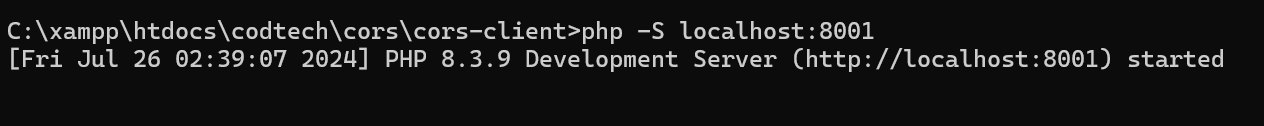
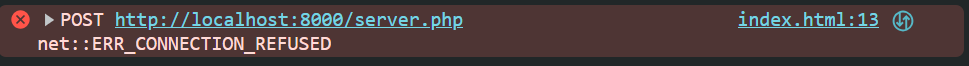
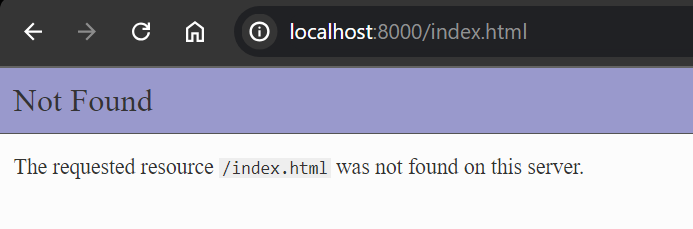
Why This is Dangerous: Unauthorized Actions: The attacker can perform actions on behalf of the victim without their knowledge or consent; Data Manipulation: The attacker can modify or delete data; Privilege Escalation: If the victim has administrative privileges, the attacker can potentially gain control over the entire system.

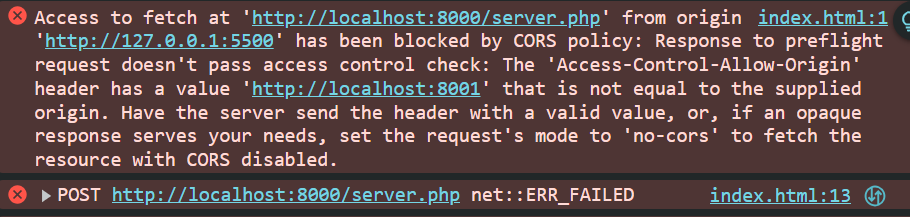
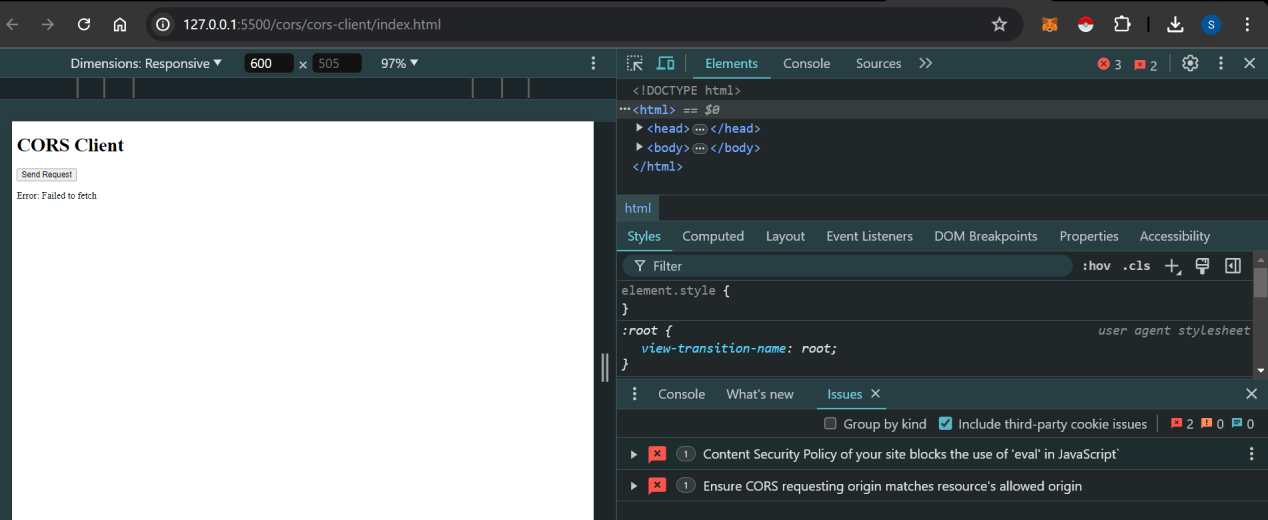
so as now by manipulating the url, not only can the attacker brute force their way into changing the creds and lockig the user out (dos) but also, are able to submit the feedback form from their side. a hidden field to store the username has been added, which again is sensitive info and can be exploited. An attacker could use tools like browser developer tools or intercepting proxies (e.g., Burp Suite) to modify the hidden fields in the feedback form. For instance: Tampering: The attacker could change the username value in the hidden field to another valid username before submitting the form. This could allow them to submit feedback as if they were another user. The attacker could save the form data and reuse it to submit feedback or other forms of data with the same parameters, potentially manipulating the system.

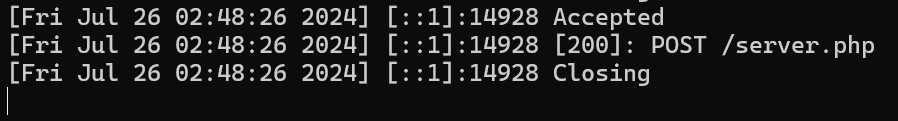
**CONCLUSION:** Implement CSRF tokens in forms to protect against unauthorized actions and allowing hidden form submission from other pages.

1. Implementation for cors:

**Cors doesn’t increase security. It is just a whitelist policy that says “oh, allow these websites too” in other words, to impose relaxations on the same origin (samesite) policy . Prev default cookie settings of browsers were set to samesite: NONE i.e. no one is restricted from sending requests. Thus vulnerable and needs csrf protection. New default is LAX i.e. even if request comes indirectly from a malicious site eg. Wanting to steal cookies, the cookies are only passed on if it’s a navigation action, hence doesn’t return cookies to the malicious site. Thus keeping the user safe. This **LAX** policy prevents a csrf attack even without a csrf token.** Make sure u have php in ur pc and its added to path in sys variables to be able to run the below cmds to change ports of localhost for both client and server ***To simulate different origins***

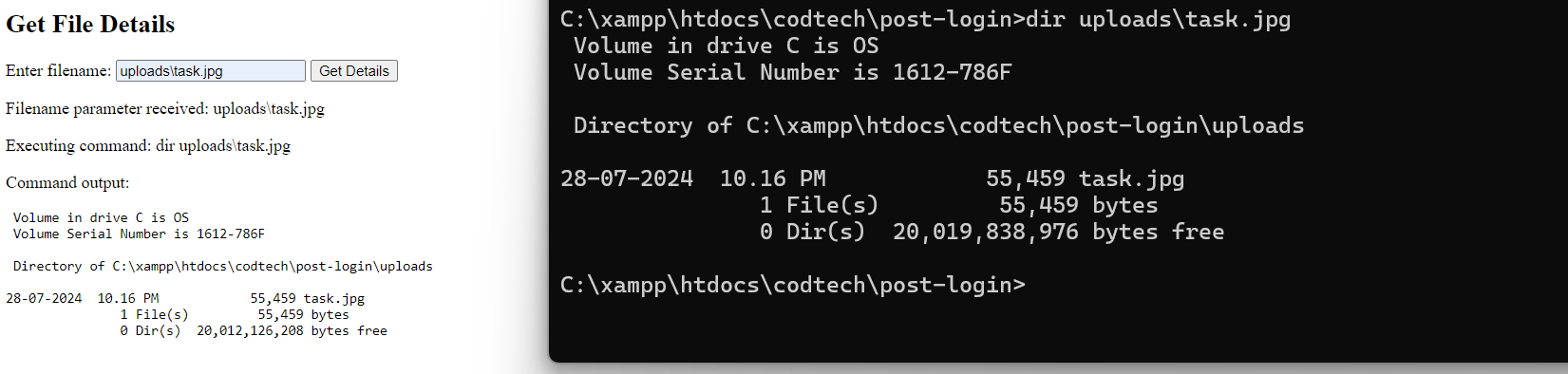
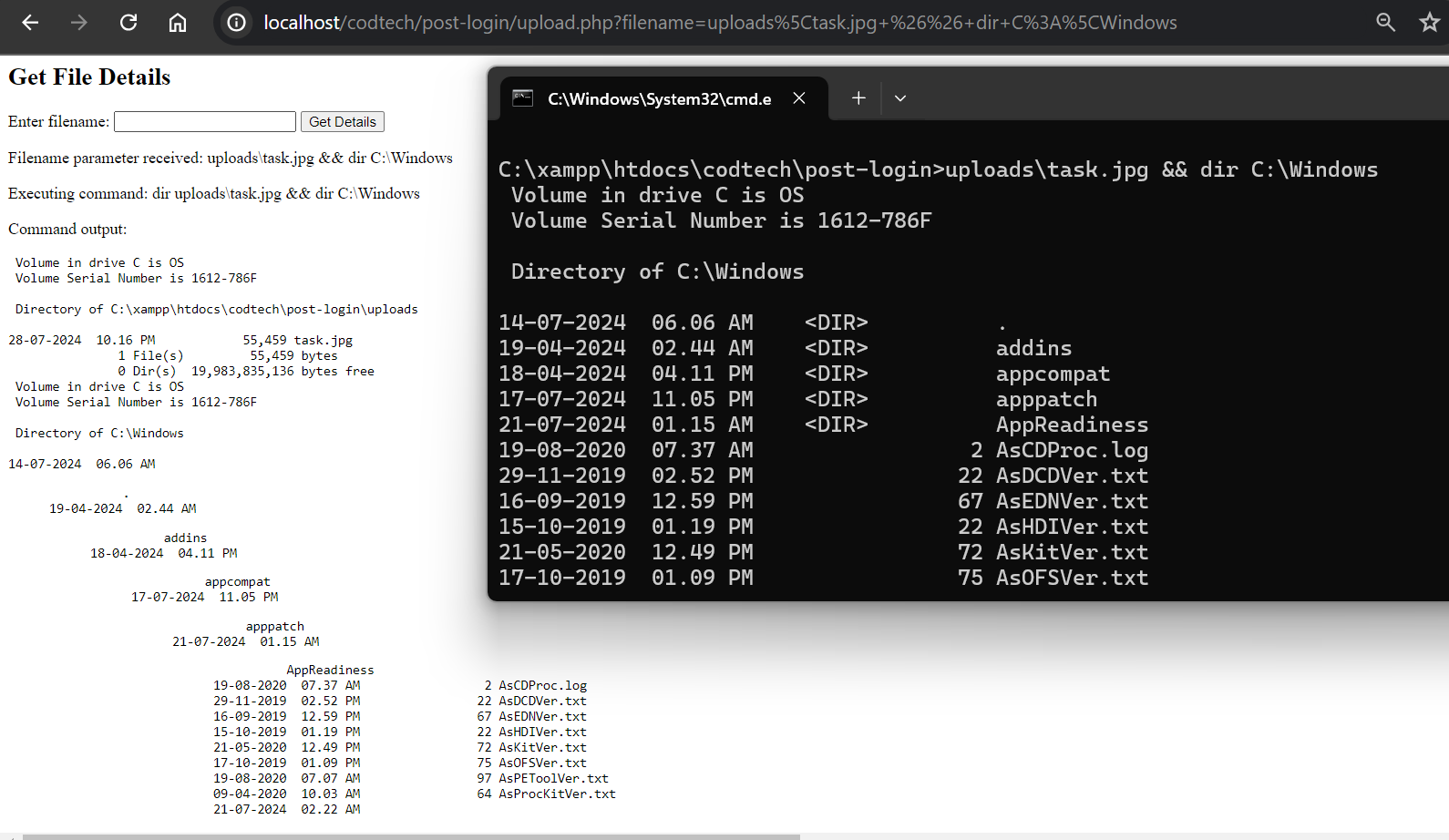
Server on port 8000: *php -S localhost:8000*  Client on port 8001: *php -S localhost:8001… <http://localhost:8001/index.html>* Make sure the server is up and running on port 8000 as shown above or else below error will be thrown: After this, open the index.html client file with live server. Since index.html is a client file, trying to open it from other port is not allowed: 

If I try to send a cors request through port 5500: (open with live server) while in server.php, code is: header("Access-Control-Allow-Origin: [http://localhost:8001");](http://localhost:8001\");) If I open the same link I.e. open with live server: <http://127.0.0.1:5500/cors/cors-client/index.html>. But now the server.php has this code: header("Access-Control-Allow-Origin: \*");

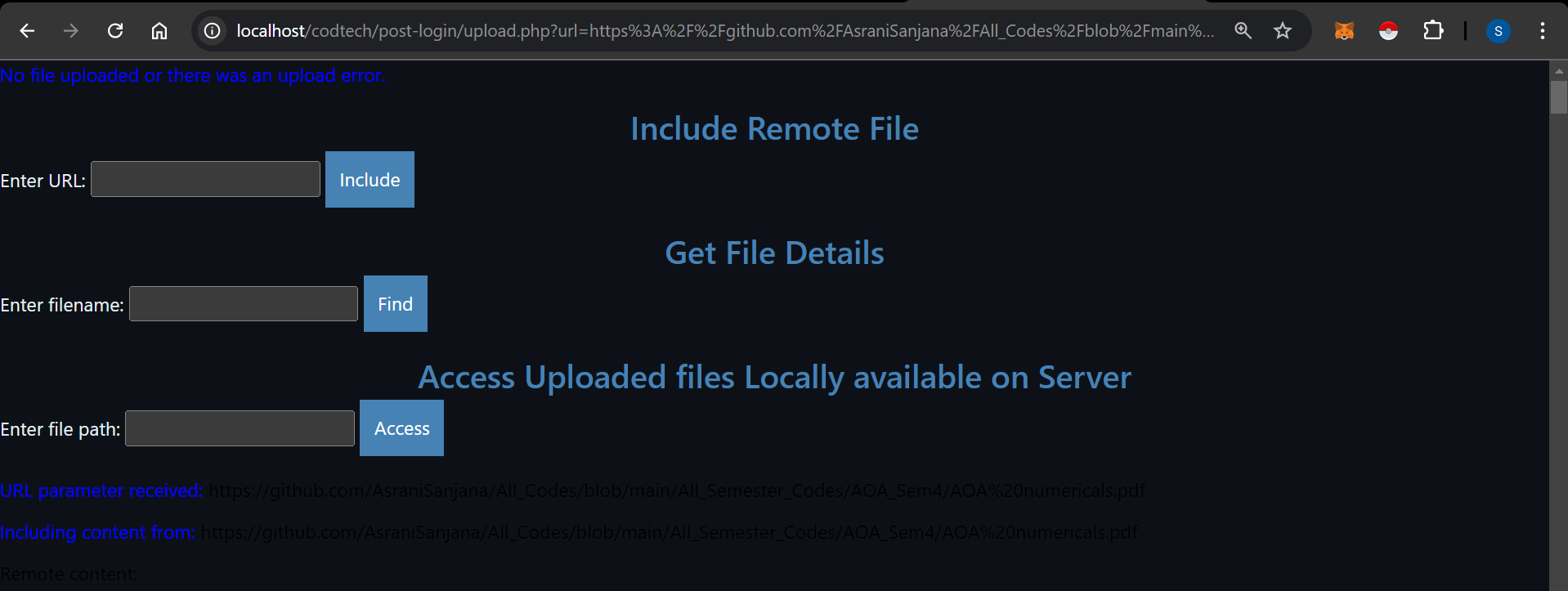
No cors error: Logs in server cmd prompt: 

· If sensitive actions like user deletion are transmitted over an unencrypted connection (HTTP instead of HTTPS), they could be intercepted. **Risk:** Attackers could potentially intercept and modify requests to delete users or perform other unauthorized actions.

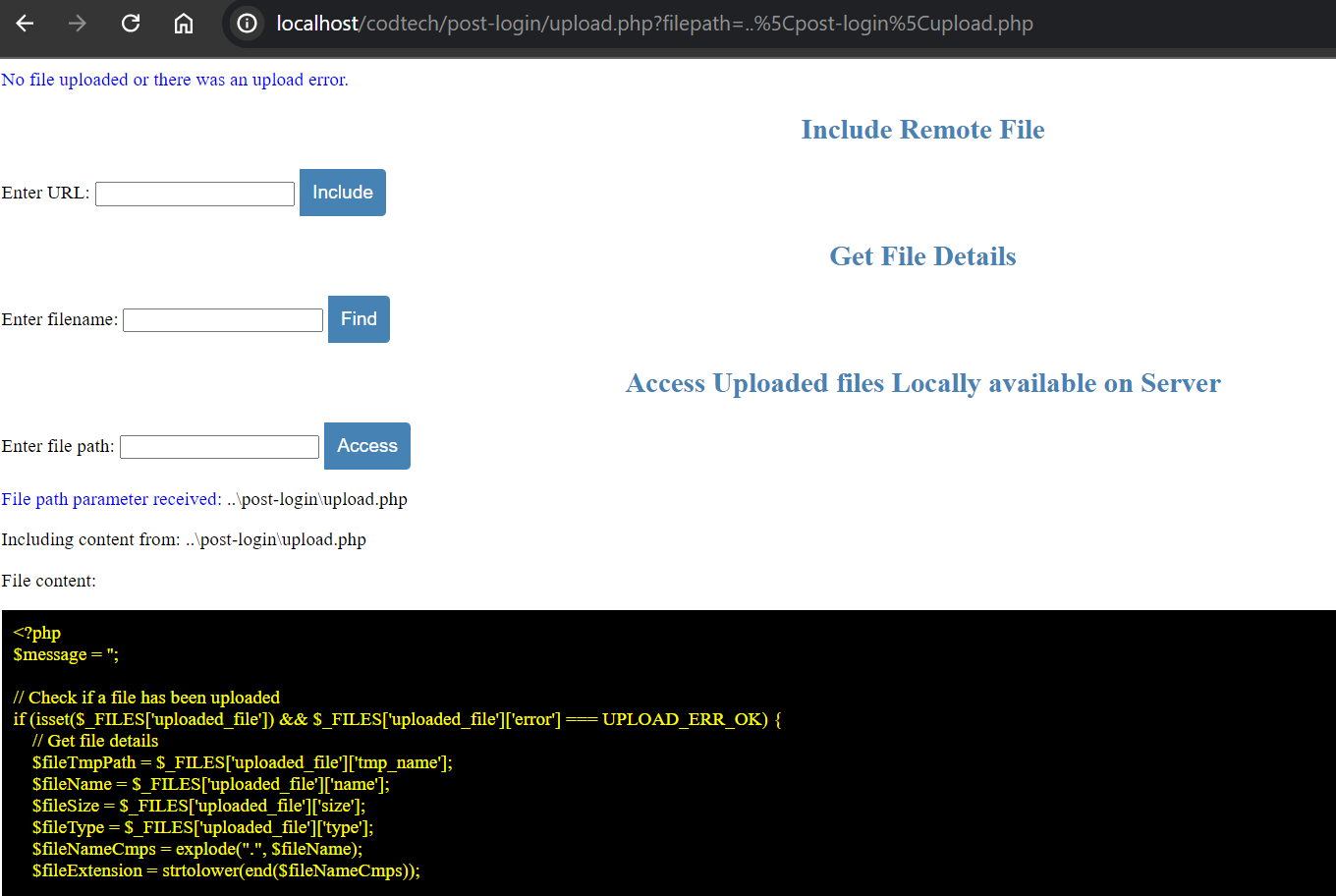
1. Cmd injection

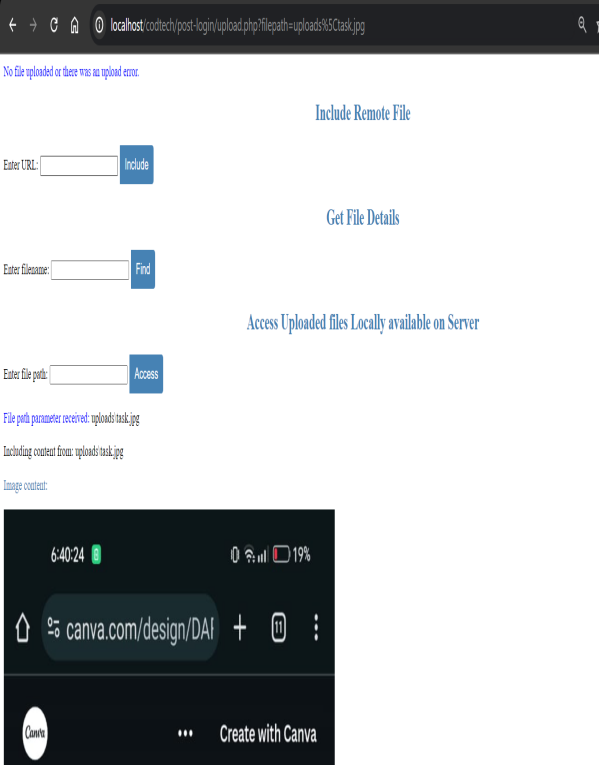
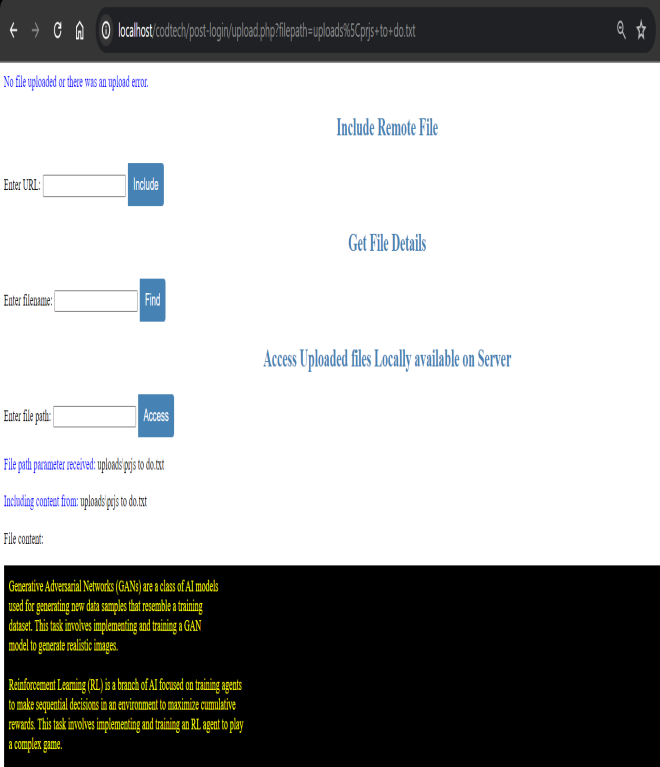
Now injecting a cmd: uploads/task.jpg && dir C:\Windows (the actual cmd prompts output is displayed for comparison)

1. Remote file inclusion

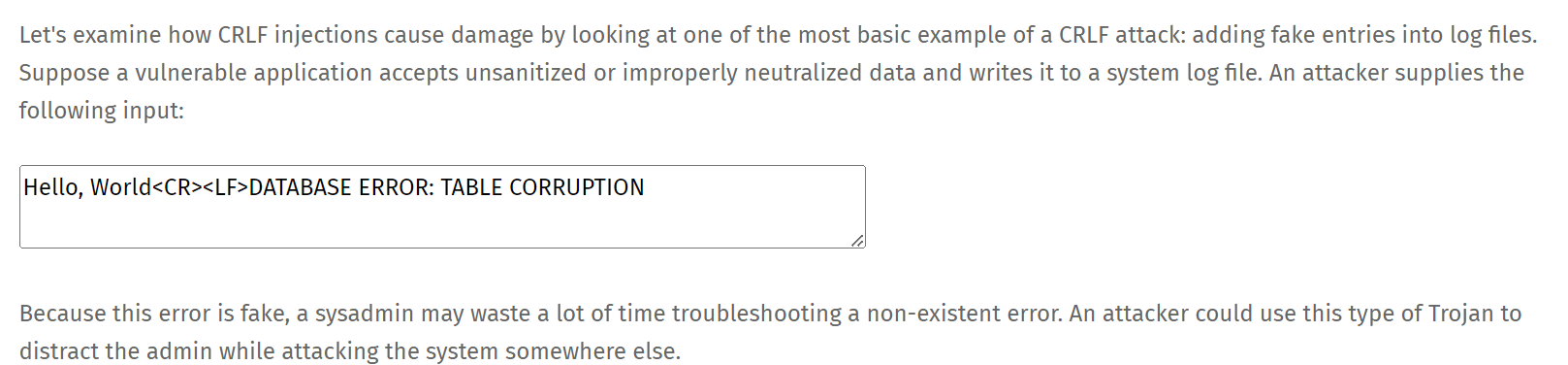
If content from a url page is to be used and a malicious link of an executable code file is inserted i.e. By fetching and evaluating the remote PHP code, you can see how an attacker might include and execute malicious code on your server. Similar to how malicious.php was uploaded. we can put a url of an image by copying the link address. 

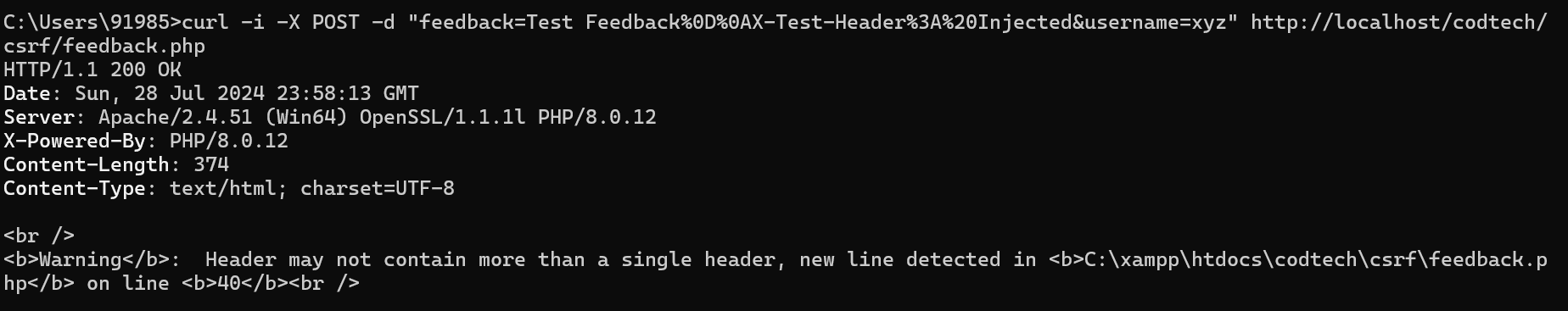
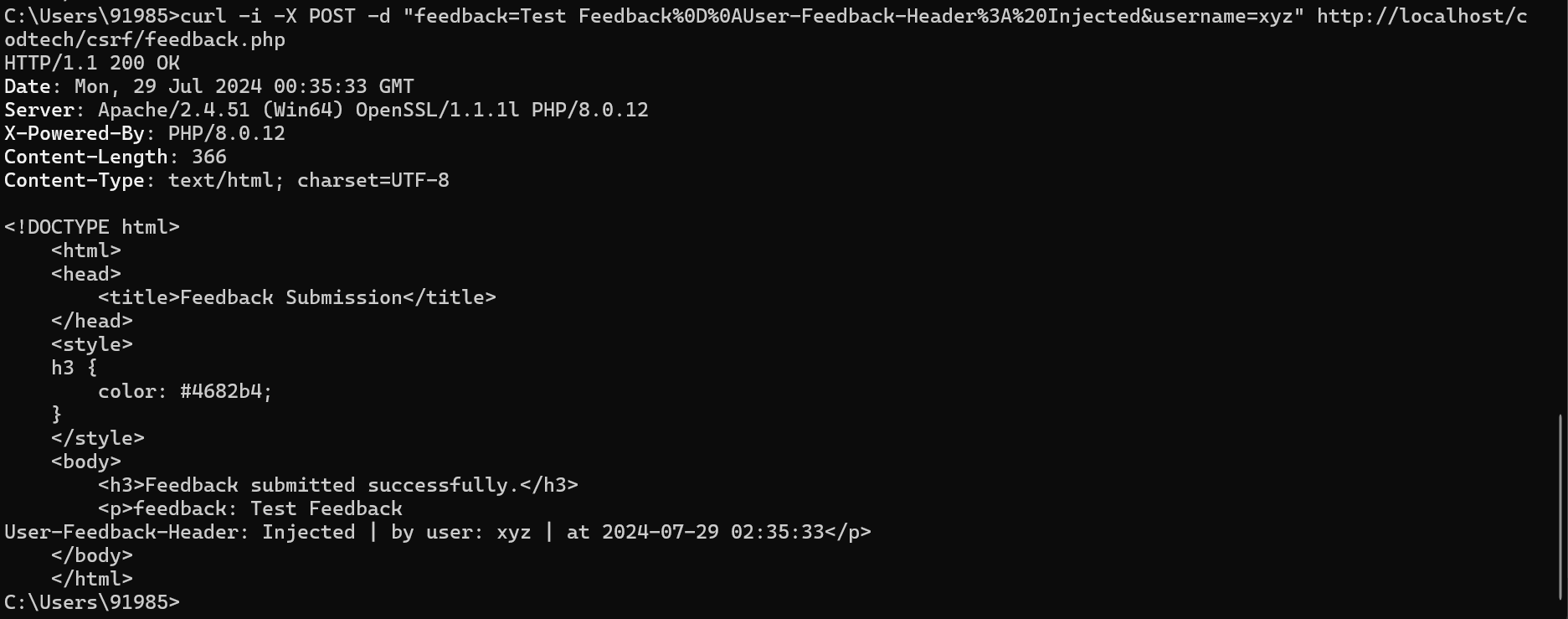
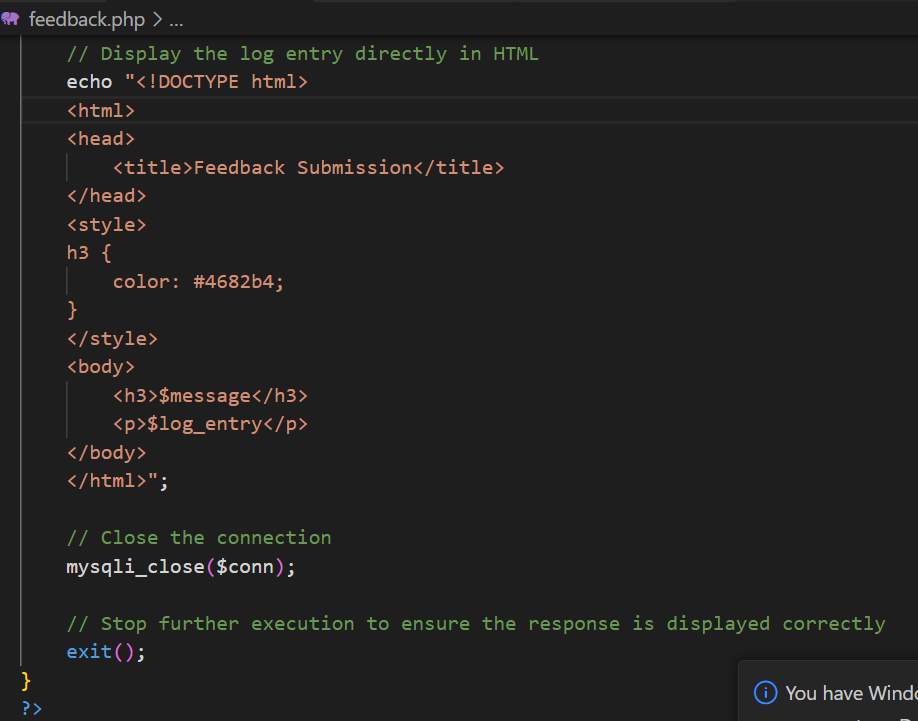
1. Local file inclusion LFI & Path traversal

Even though intent it to give access for uploaded files. One can misuse this by navigating through the file structure like this: The file structure can been guessed by the url of the page in both cases of absolute and relative urls. Hence, non-descriptive URLs, Restricted Access to Sensitive Files, ACLs (Access control lists) are important to mitigate this vulnerability.

Since the files uploaded are locally present on server and can be accessed by users, a user can try to access any file on the system by path traversal

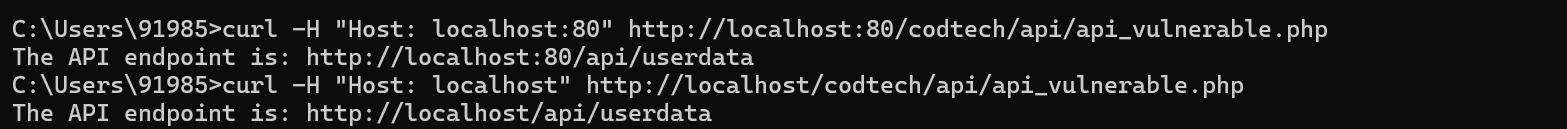
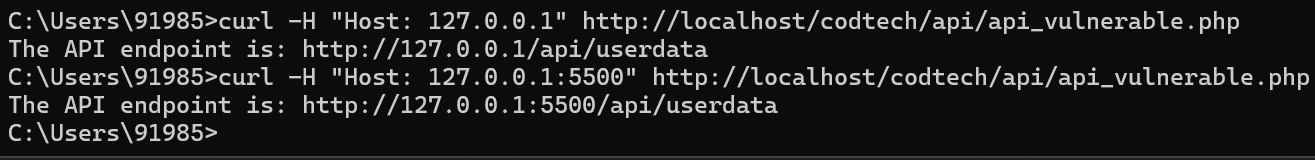
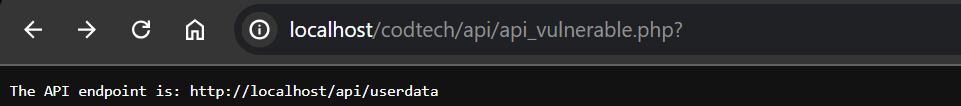
1. Crlf

A payload with CRLF characters (%0D%0A) can be crafted. This sequence represents a carriage return and line feed, used to break HTTP headers and inject new headers.

Cmd> curl -i -X POST -d "feedback=Test Feedback%0D%0AUser-Feedback-Header%3A%20Injected&username=xyz" <http://localhost/codtech/csrf/feedback.php> When its tried to direclty send a fedback from cmd line and with a motive to add custom headers in request The %0D%0A can be seen as being translated to <br/> which is encoded as a newline character (LF). Due to security reasons of header() in php, another way to demo this vuln is: The above output is corresponding to this code snippet: 

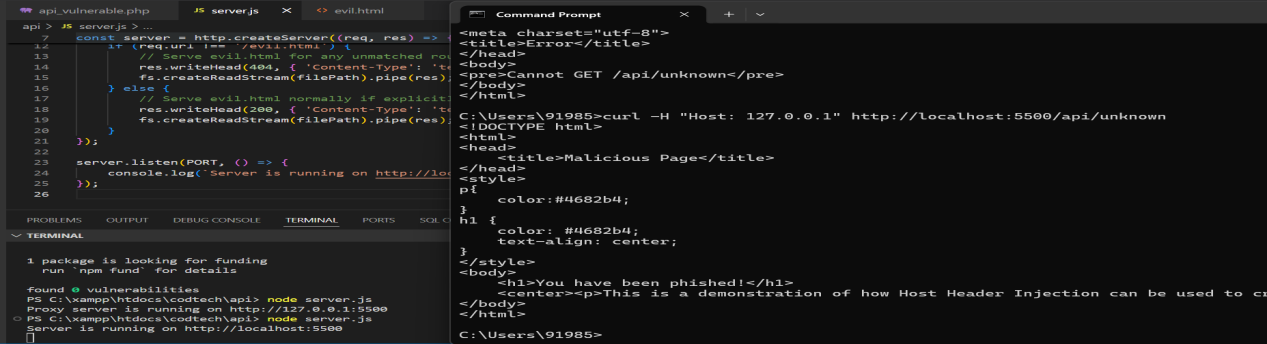
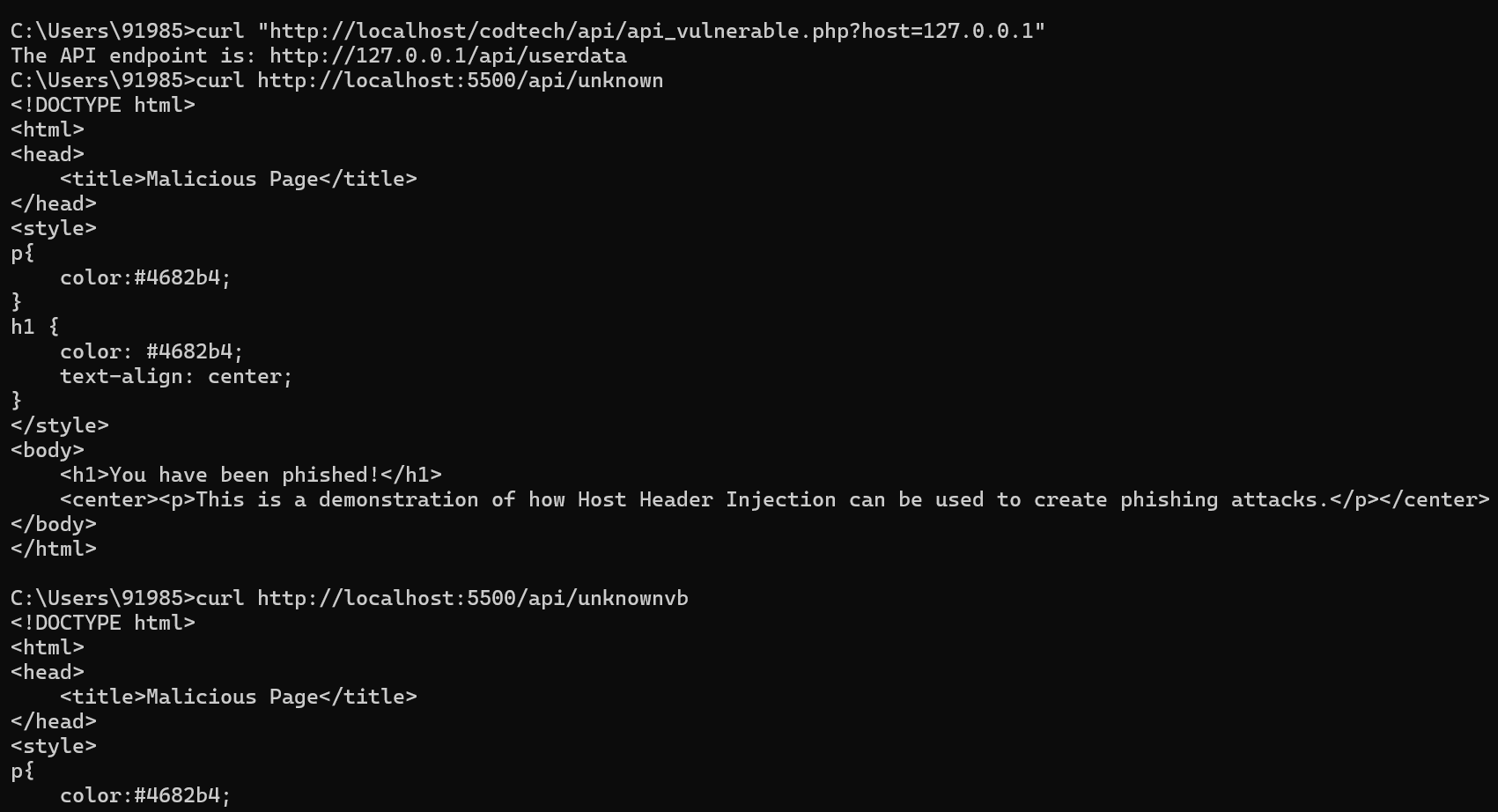
Logging it as well and displaying it on the page as well Hello World\r\nX-Test:Injection trial. When these characters are sent to db: 

1. Host header injection:

Using curl for demo: curl <http://localhost/codtech/api/api_vulnerable.php?> curl -H "Host: 127.0.0.1" <http://localhost/codtech/api/api_vulnerable.php?>  If a web extension is used to change the host header eg. Modify Header Value Simulated Redirect: By manipulating the Host header, the script will construct URLs that point to your local malicious page (evil.html) instead of the api that it was supposed to point to. As soon as the api\_vulnerable.php is visited, a dynamic api link is formed. Now if the host headers can be manipultaed, instead of the api link, another malicious links could be inserted. Here im just using a url paramter to pass a host name. Like this: · **Request Sent**: You send a request with localhost to api\_vulnerable.php -> You change the host parameter: the url parameter to 127.0.0.2:5500 -> You send a request with Host: 127.0.0.1:5500 to api\_vulnerable.php.

· **Redirection**:· api\_vulnerable.php constructs the endpoint : http://127.0.0.1:5500/api/userdata and redirects to it.

· **Server Response**:· Since http://127.0.0.1:5500/api/userdata is not found, the server serves evil.html, which might be set up as the default response for unspecified paths through htaccess

Just like I have configured evil.html to be the default page when the exact path on localhost is not found using server.js 

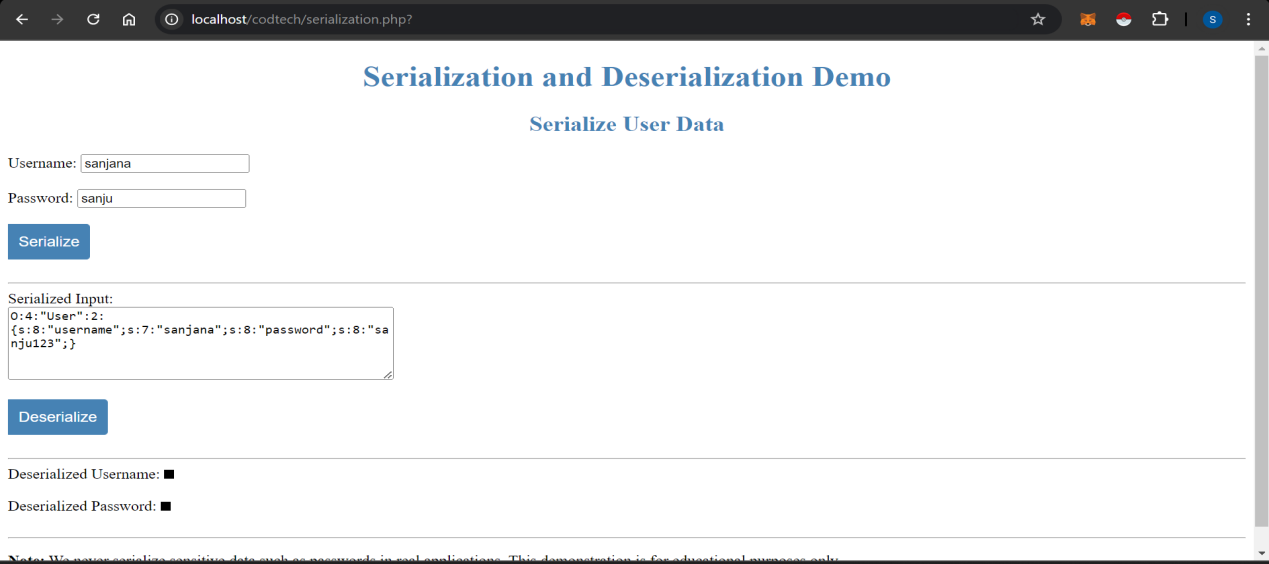
1. serialization

Oops classes and objs used in php . Use of prepared stmts is used to prevent sqli here.

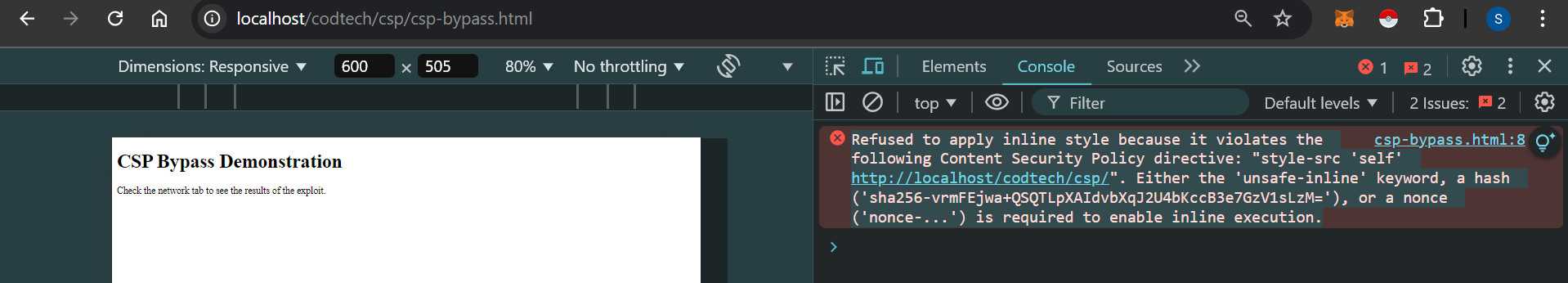
Serialized data often includes more than just credentials. It might contain session information, permissions, and other sensitive data. An attacker could manipulate this information to bypass security controls. Furthermore Serialized data is often opaque and not easily readable, making it difficult to detect tampering or malicious payloads. Unlike simple credential compromises, these vulnerabilities can allow attackers to execute arbitrary code, escalate privileges, and manipulate sensitive data beyond what typical user access controls permit. Thus Serialization and deserialization vulnerabilities can be significantly more dangerous than just having access to a login portal with credentials.

· **Malicious Payload:**· The MaliciousPayload class defines a \_\_wakeup method that is called upon deserialization. This method executes a query to retrieve and display plaintext passwords from the database.

· **Deserialization Attack:**· An attacker could serialize an instance of MaliciousPayload and inject it into the application. When the application deserializes this object, the \_\_wakeup method runs, exposing the plaintext passwords.

The attacker realizes that by manipulating serialized objects, they can craft a payload to trick the application into revealing sensitive information eg. log plaintext passwords or return sensitive data directly to the attacker. If somehow attacker gains a hashed pswd, he can misuse the serialization process given he knows the business logic and application flow. When the application deserializes the crafted object, it executes the malicious payload. This payload instructs the application to expose the plaintext password associated with the hash the attacker already has. If an attacker tries to deserialize an insecure string, he could get the creds and get into the system

1. Csp bypass

Ref: <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy/script-src>  If *<meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self' http://localhost/codtech/csp/evil.html; style-src 'self' http://localhost/codtech/csp/;">*  is used in csp-bypass.html. A very basic version of above ref is done to demonstrate exploitation of unsafe-inline and unsafe-eval. The dynamic changing of the meta policy is being done by **Meta injections** (again a csp bypass vuln).

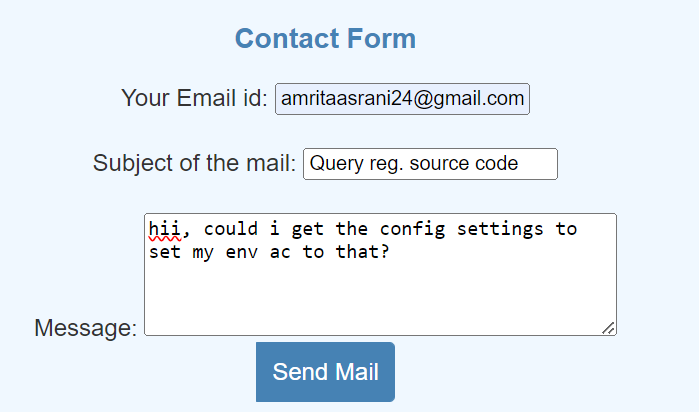
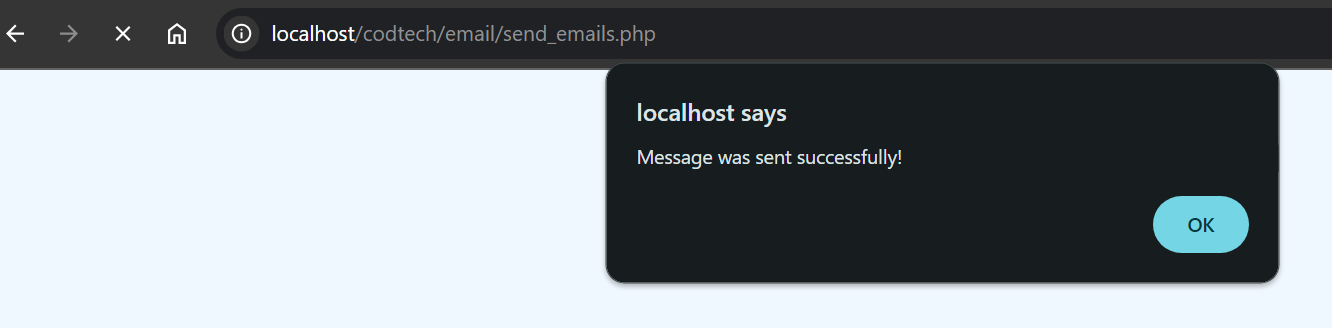
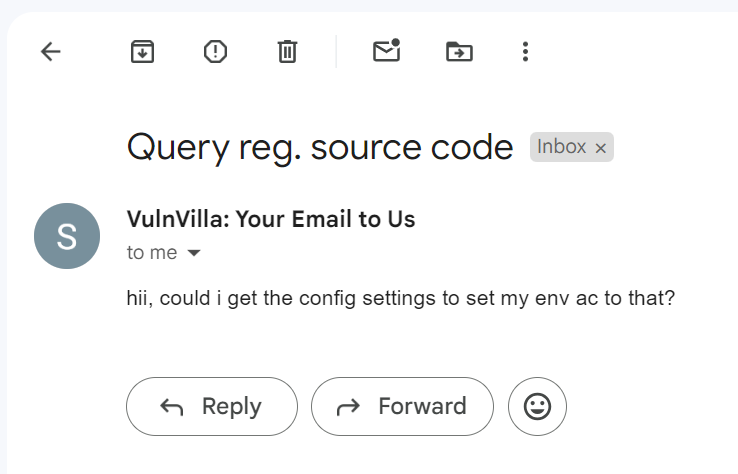
1. Email header injection

additional headers could be included, sending the email to unintended addresses, cause spam, modify body to add malware/links

1. **Email Field Input**: Eg. attacker@example.com\r\nBCC:[another@example.com\nSubject:Hacked](mailto:another@example.com)

**[2. Message Field Input](mailto:another@example.com)**[: <script>alert('XSS');</script>](mailto:another@example.com)

[If the email content is displayed in an environment that renders HTML, this could execute unwanted scripts.](mailto:another@example.com) If your email handling script is not robust, attackers might be able to send a large volume of emails to overwhelm your server or email system.mitigation: · Implement rate limiting and spam detection mechanisms. Use Secure CAPTCHAs or other forms of verification to limit automated submissions. When the user clicks on send email, it is sent to the user itself for demo purposes here,

Generally these emails back to sender would include an automated reply. This is just for demonstration purposes.

1. Data Parsing Vuln Demo
2. XML INJECTION

Normal ip: <user><name>John Doe</name></user> 

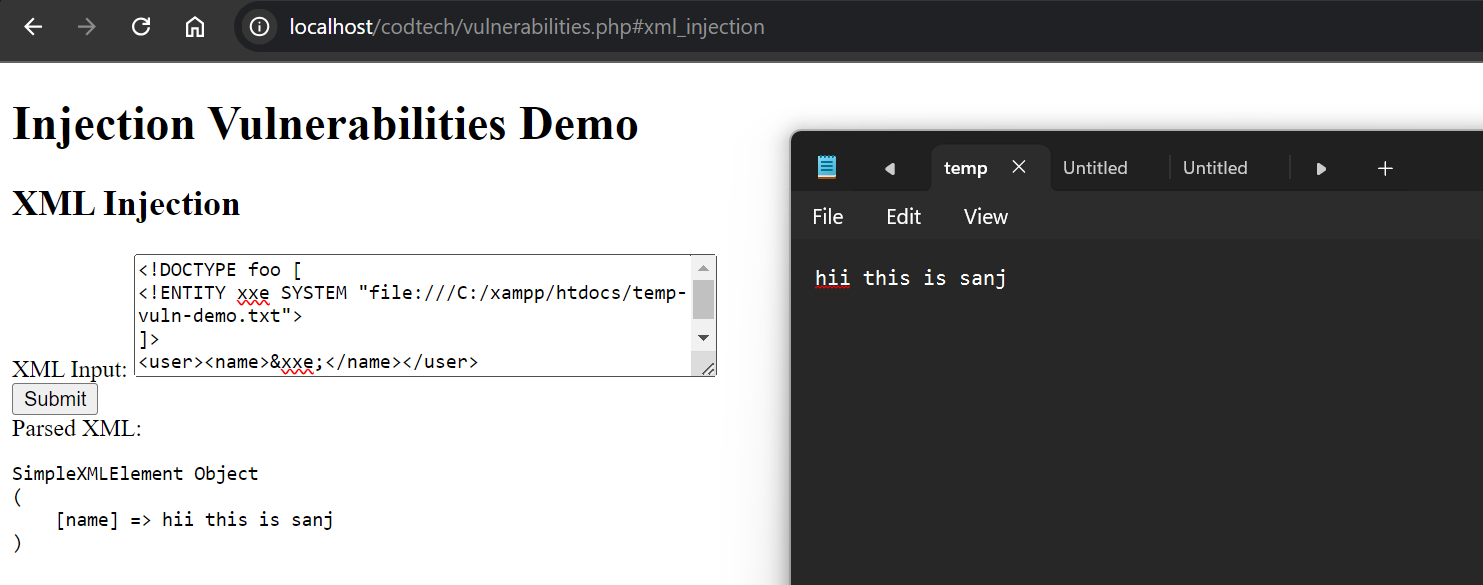
Malicious input: This can potentially allow to read files from the server if external entities are allowed.

<!DOCTYPE foo [

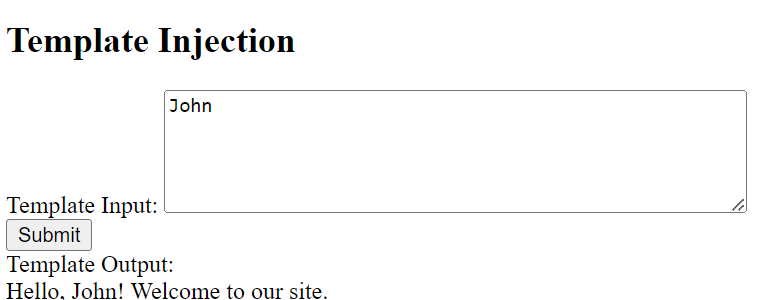
<!ENTITY xxe SYSTEM "file:///C:/xampp/htdocs/temp-vuln-demo.txt">

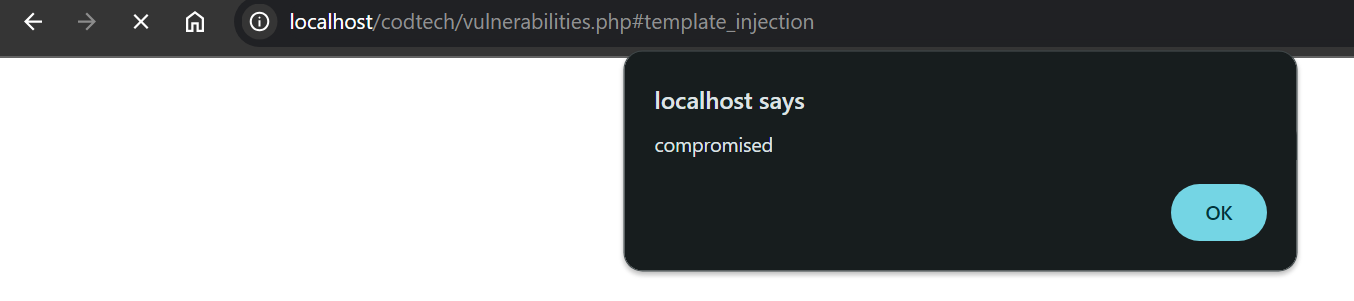
]>

<user><name>&xxe;</name></user>



1. Template injection:

Normal input: 

Malicious input: <script>alert("compromised");</script> **Key Differences between xpath and xquery:**

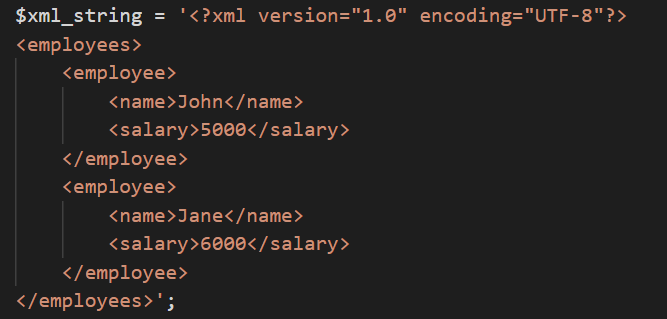
**XPath** is mainly used for querying and filtering XML data.

· **Purpose:** Used to navigate through elements and attributes in XML documents.

· **Types of Queries:**

* **Path Expressions:** Select nodes or a set of nodes on the basis of attributes and relationships.
  + Example: /employees/employee/name selects all <name> elements under <employee> elements.
* **Predicates:** Filter nodes.
  + Example: /employees/employee[name='John'] selects <employee> nodes where the <name> child node's value is 'John'.
* **Functions:** Perform operations on nodes.
  + Example: count(/employees/employee) counts the number of <employee> nodes.

1. Xpath injection:



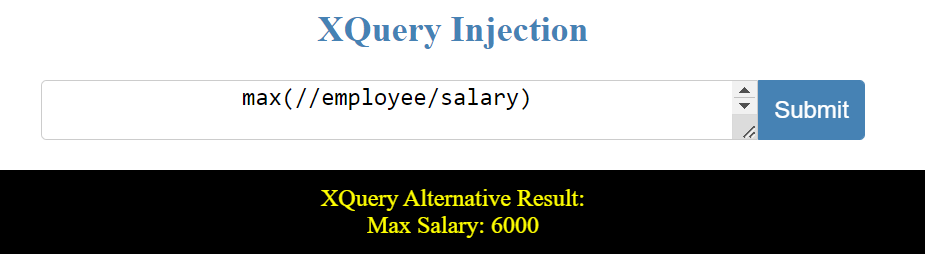
A demo db has been created in xml as shown above.

Input: //employee[name=’Jane’]/salary

xpath can run simple queries but xquery can do much more complex ones.

1. **XQuery:** builds upon XPath, allowing more complex queries, aggregations, and transformations. **XQuery (XML Query Language):**

* **Purpose:** Extends XPath to provide a more comprehensive query language, supporting complex queries, aggregations, and transformations.
* **Types of Queries:**
  + **FLWOR Expressions (For, Let, Where, Order by, Return):** Similar to SQL queries, providing powerful querying capabilities.

Input: max(//employee/salary)

Note: Because PHP does not support XQuery directly, this section demonstrates a simple simulation (hardcoded). It checks if the input query includes specific keywords (max, sum) and performs corresponding operations on the XML data.

\*business logic vulnerabilities which are app specific, the exact cfg of website should be correct. logic errors like typecasting can be exploited\*

Other possible vulnerabilities would be LDAP injections, Insecure Captchas, JWT insecure token management, etc.