INSA CYBER TALENT SUMMER CAMP

INDIVIDUAL ASSIGNMENT

Name: Firaol Assefa

Submission date:8-17-2025

Contents

Vulnerability Report

Vulnerability Report

Vulnerability Report

1, Username Enumeration

1.Description

The application’s password reset functionality suffers from a username enumeration vulnerability, allowing attackers to determine valid usernames by analyzing differences in system responses. This issue occurs in the /api/v2/forgot-password endpoint, which leaks information through inconsistent HTTP status codes and debug messages.

How the Vulnerability Works:
- Valid Username Request:
  - Returns a 200 OK response with a success message (e.g., "Reset PIN has been sent to your email.")
  - Includes debug information (username, timestamp) in the response body.

- Invalid Username Request:
  - Returns a 404 Not Found response, clearly indicating the username does not exist.

Proof of Concept (PoC):
An attacker can perform a brute-force or dictionary attack by submitting common usernames (e.g., admin, root, test) and observing the responses. In this case:
- Only admin returned a 200 OK with a success message.
- All other usernames (test, guest, mysql, etc.) returned 404 Not Found.

This behavior allows an attacker to confirm valid usernames, significantly reducing the attack surface for credential stuffing or brute-force attacks.

2. Impact

If exploited, this vulnerability can lead to:

1. Account Takeover (ATO):
  - Attackers can identify high-value accounts (e.g., admin) and target them for password reset abuse.
  - In this case, the system sent an OTP to the admin’s email, which could be intercepted or phished.

2. Credential Stuffing & Brute-Force Attacks:

Vulnerability Report

- Knowing valid usernames makes password-guessing attacks far more efficient.

3. Information Leakage:
- The debug response (debug_info) exposes internal details (timestamps, usernames), aiding attackers in reconnaissance.

4. Phishing & Social Engineering:
- Attackers can craft convincing phishing emails since they know which usernames exist.

Worst-Case Scenario:
An attacker could:
- Identify the admin account.
- Trigger a password reset OTP.
- Intercept or brute-force the OTP.
- Gain full administrative access.

3. Countermeasures (Fix Recommendations)

To mitigate this vulnerability, implement the following security measures:

1. Standardize Responses (Most Critical Fix)
- Do not differentiate between valid and invalid usernames.
- Always return a 200 OK with a generic message:
  ```json
  { "message": "If the account exists, a reset link has been sent." }
  ```
- Do not expose debug information in production.

2. Rate Limiting & Account Lockout
- Implement request throttling (e.g., max 5 attempts per minute).
- Example (Flask-Limiter):

```python
from flask_limiter import Limiter
limiter = Limiter(app, key_func=get_remote_address)
@limiter.limit("5/minute")
def forgot_password():
    # Reset logic here
```

3. CAPTCHA Protection
- Require CAPTCHA for password reset requests to prevent automated attacks.

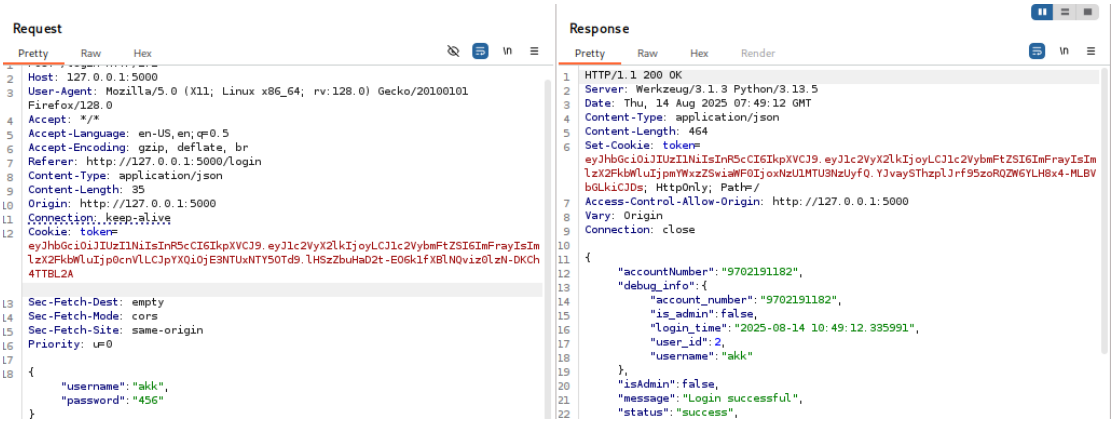
4. Disable Debug Mode in Production
- Ensure DEBUG=False in Flask/Werkzeug configurations.

Vulnerability Report

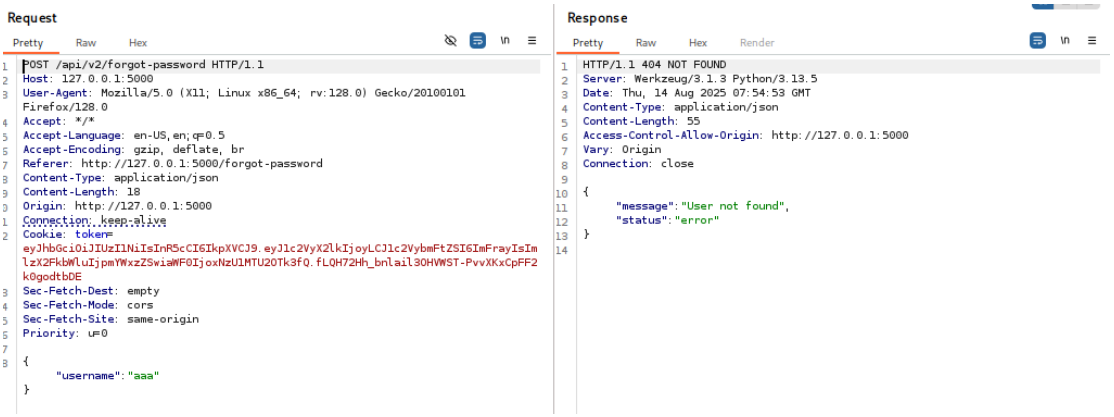
5. Log & Monitor Suspicious Activity

- Alert on multiple failed password reset attempts for the same account.

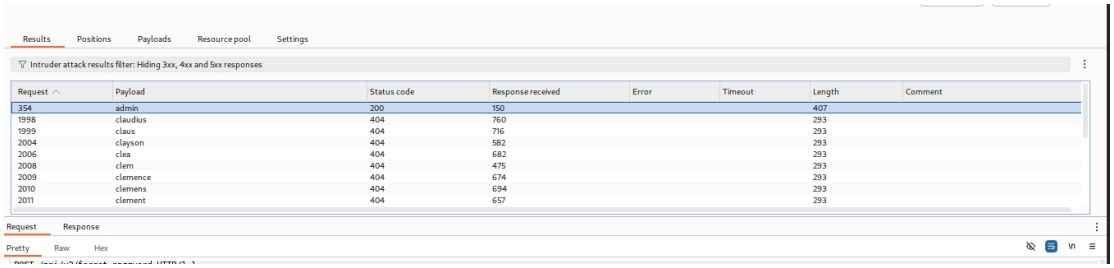
4. Evidence



Screenshot 1

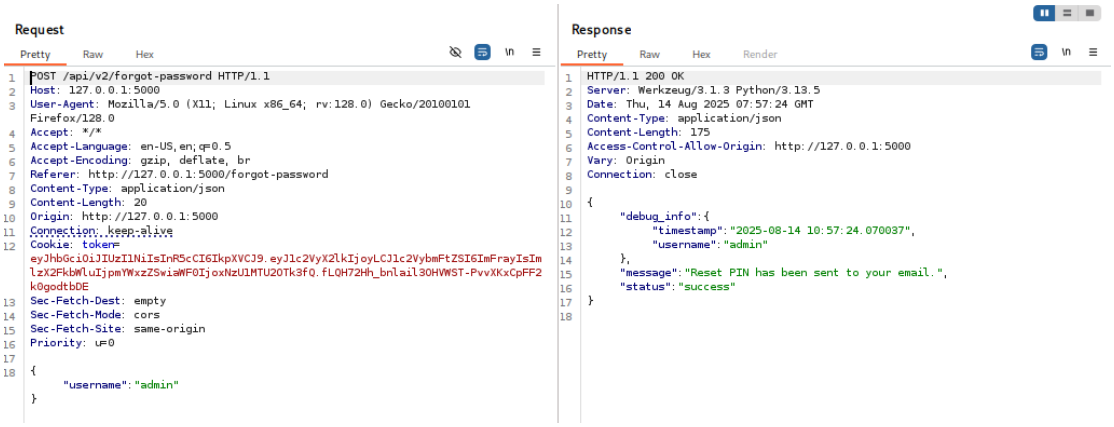


Screenshot 2



Screenshot 3

Vulnerability Report



Screenshot 4

Conclusion

This vulnerability severely weakens authentication security by allowing attackers to discover valid usernames. The most critical fix is standardizing responses to prevent enumeration, followed by implementing rate limiting, CAPTCHA, and disabling debug info.
Remediation Priority: High (Critical for admin accounts)
Exploit Difficulty: Low (Easily automated with tools like Burp Intruder)

2, Weak Password Reset Mechanism (Brute-Forceable 3-Digit OTP)

1. Description

The application's password reset functionality is vulnerable to brute-force attacks due to the use of a weak 3-digit OTP (One-Time Password) system with insufficient rate limiting. This vulnerability allows attackers to systematically guess the OTP and reset user passwords without proper authorization.

Technical Details:
- The reset endpoint (/api/v2/reset-password) accepts a 3-digit numeric OTP (000-999)
- No effective rate limiting exists to prevent brute-force attempts
- Successful attempts return HTTP 200 status with a success message
- Failed attempts return HTTP 400 status
- The small keyspace (1000 possible combinations) makes brute-forcing trivial

Attack Methodology:
1. Attacker requests password reset for target account (OTP sent to user's email)
2. Intercepts the OTP submission request using a proxy tool (Burp Suite)
3. Performs brute-force attack using payloads 000-999
4. Identifies successful attempt by HTTP 200 response

Vulnerability Report

5. Gains unauthorized access by setting new password

2. Impact

This vulnerability presents severe security risks:

1. Complete Account Takeover:
   - Attackers can reset passwords for any user account
   - Demonstrated successful reset using OTP 106 (see Evidence)

2. Privilege Escalation:
   - Particularly dangerous for admin/high-privilege accounts
   - Could lead to full system compromise

3. Low-Barrier Exploitation:
   - Requires no special tools or skills beyond basic web testing
   - Entire attack can be automated with simple scripts

4. Compliance Violations:
   - Violates basic security principles (NIST recommends 6+ digit OTPs)
   - May contravene data protection regulations

3. Countermeasures

To remediate this vulnerability, implement the following security controls:

1. Stronger OTP Implementation:
   - Increase OTP length to at least 6 alphanumeric characters
   - Implement time-based expiration (e.g., 10-minute validity window)

2. Rate Limiting:

```
   ```python
from flask_limiter import Limiter
limiter = Limiter(app, key_func=get_remote_address)
@limiter.limit("3/hour")  # Strict rate limiting
def reset_password():
    # Reset logic
```
```

3. Account Lockout:
   - Temporarily lock accounts after 5 failed OTP attempts
   - Require admin intervention or secondary verification

4. CAPTCHA Integration:
   - Require CAPTCHA verification before OTP submission
   - Prevents automated brute-force attacks

Vulnerability Report

5. Security Headers:
   - Implement headers like X-Content-Type-Options, X-Frame-Options
   - Helps prevent certain types of attacks

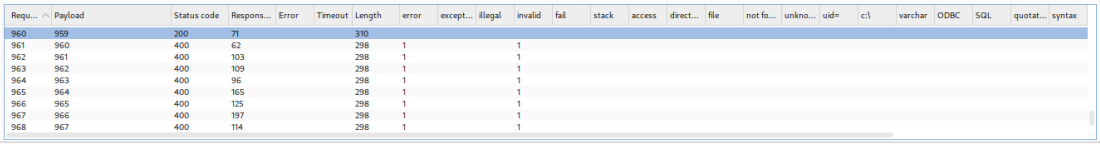
6. Monitoring:
   - Log all password reset attempts
   - Alert on suspicious patterns (rapid-fire attempts)

. Evidence

Screenshot 1: Password Reset Request

```
POST request to /api/v2/reset-password
```

Screenshot 2:




```
HTTP/1.0 200 OK
Content-Type: application/json
{
    "message":"Password has been reset successfully",
    "status":"success"
}
```

Conclusion

The weak password reset implementation poses a critical security risk, allowing trivial account takeover through OTP brute-forcing. The combination of short OTP length and absent rate limiting creates an easily exploitable vulnerability.

Vulnerability Report

Remediation Priority: Critical
Exploit Difficulty: Low (Automated tools can complete attack in minutes)
Recommended Actions:
1. Immediately implement rate limiting
2. Increase OTP complexity before next release
3. Monitor for suspicious reset attempts
4. Consider forced password resets for recently changed passwords

3,Negative Amount Transfer Vulnerability

1. Description

The banking application's transfer functionality contains a critical vulnerability that allows users to initiate transfers with negative amounts, effectively enabling unauthorized fund accumulation. This vulnerability stems from two primary security failures:

1.Insufficient Input Validation: The system fails to properly validate and sanitize the 'amount' parameter in transfer requests, accepting negative values without restriction.

2. Broken Function Level Authorization: The application does not enforce proper business logic validation to prevent illogical transaction types (transferring negative amounts).

Technical Details:
- Endpoint: `/transfer` (POST)
- Vulnerable Parameter: `amount`
- Expected Behavior: Should only accept positive numerical values
- Actual Behavior: Accepts negative values and processes them as reverse transfers

Attack Scenario:
1. Attacker sends a transfer request with a negative amount:
```json
{
  "to_account": "3344601304",
  "amount": -200,
  "description": "Vulnerability"
}
```
2. System processes the request as valid
3. Attacker's account balance increases instead of decreasing:

- Initial balance: 800.0
- After "transfer": 1000.0

4. Transaction marked as "Transfer Completed" with "success" status

2. Impact

This vulnerability presents severe financial and security risks:

1. Direct Financial Loss:
   - Attackers can arbitrarily increase their account balance
   - Demonstrated attack resulted in +200 balance increase

2. System Integrity Compromise:
   - Undermines core banking functionality
   - Violates fundamental accounting principles

3. Potential for Large-Scale Abuse:
   - Vulnerability can be exploited repeatedly
   - Could lead to significant financial damage if exploited at scale

4. Regulatory Consequences:
   - Violates financial system integrity requirements
   - May result in compliance penalties

5. Reputation Damage:
   - Erodes customer trust in the banking platform
   - Could lead to loss of business

3. Countermeasures

To remediate this vulnerability, implement the following security controls:

1. Input Validation:

```python
def validate_transfer(amount):
    if not isinstance(amount, (int, float)) or amount <= 0:
        raise ValueError("Amount must be a positive number")
```

2. Business Logic Validation:
   - Implement server-side validation ensuring transfer amounts are positive
   - Add validation that sender has sufficient funds

Vulnerability Report

3. API Security Hardening:

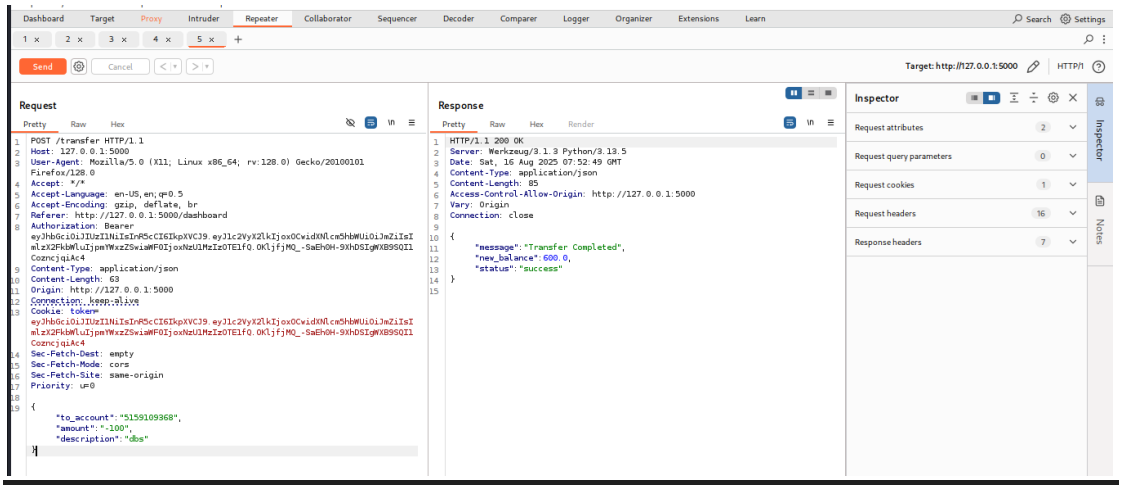
```python
@app.route('/transfer', methods=['POST'])
def transfer():
    try:
        amount = float(request.json['amount'])
        if amount <= 0:
            return jsonify({"error": "Invalid amount"}), 400
    except ValueError:
        return jsonify({"error": "Invalid amount format"}), 400
```

4. Transaction Monitoring:
   - Implement anomaly detection for unusual transaction patterns
   - Alert on negative amount attempts

5. Regular Security Audits:
   - Conduct periodic code reviews for similar vulnerabilities
   - Implement automated API security testing

6. Positive Security Model:
   - Define strict parameter requirements for all financial transactions
   - Use schema validation for all API requests

4.Evidence




```
{
  "to_account": "3344601304",
  "amount": -100,
  "description": "Vulnerability"
}
```

```
{
  "message": "Transfer.Completed",
  "new_balance": 600.0,
```

Vulnerability Report


```
  "status": "success"
}
```


Conclusion

The negative amount transfer vulnerability represents a critical flaw in the banking application's financial transaction processing. The complete lack of validation for transfer amounts allows malicious actors to artificially inflate account balances, with potentially devastating financial consequences.

Remediation Priority: Critical
Exploit Difficulty: Low (Requires no special tools or knowledge)
Attack Surface: High (All user accounts with transfer privileges)

Immediate Actions Required:
1. Disable the transfer functionality until patched
2. Implement server-side amount validation
3. Audit all transaction records for signs of exploitation
4. Consider resetting balances for accounts showing abnormal increases

Long-term Recommendations:
1. Implement comprehensive input validation for all financial APIs
2. Develop automated tests for transaction validation
3. Conduct security training for development team

4,Unrestricted File Upload Vulnerability

1. Description

The web application contains a critical unrestricted file upload vulnerability in its profile picture upload functionality. This security flaw allows attackers to upload malicious files (including DLL files) to the server without proper validation or filtering.

Technical Details:
- Vulnerable Endpoint: Profile Picture Upload feature
- Missing Security Controls:
  - No file type verification
  - No content validation
  - No file extension filtering
  - No malware scanning
- Attack Demonstrated: Successful upload of a malicious DLL file
- Response: Server returns 200 OK status for malicious uploads

Vulnerability Report

Attack Scenario:
1. Attacker identifies the profile picture upload feature
2. Prepares a malicious DLL file disguised as an image
3. Bypasses client-side checks (if any) using proxy tools
4. Successfully uploads the malicious file to the server
5. Receives 200 OK confirmation from the server

2. Impact

This vulnerability presents significant security risks:

1. Remote Code Execution (RCE):
   - Uploaded DLLs could be executed on the server
   - Potential complete system compromise

2. Server Compromise:
   - Malicious files could create backdoors
   - Attackers could gain persistent access

3. Data Breach:
   - Uploaded malware could steal sensitive data
   - Database contents could be exfiltrated

4. Malware Distribution:
   - Uploaded files could infect other users
   - Server could become malware distribution point

5. Reputation Damage:
   - Loss of customer trust
   - Potential regulatory penalties

3. Countermeasures

To remediate this vulnerability, implement the following security controls:

1. File Type Validation:

```python
ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}
def allowed_file(filename):
    return '.' in filename and \
        filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

2. Content Verification:
   - Use magic numbers to verify file content matches extension
   - Implement server-side file content analysis

3. File Renaming:
   - Generate random filenames for uploaded files
   - Strip metadata from uploaded images

4. Malware Scanning:
   - Integrate antivirus scanning for all uploads
   - Quarantine suspicious files

5. File Upload Restrictions:

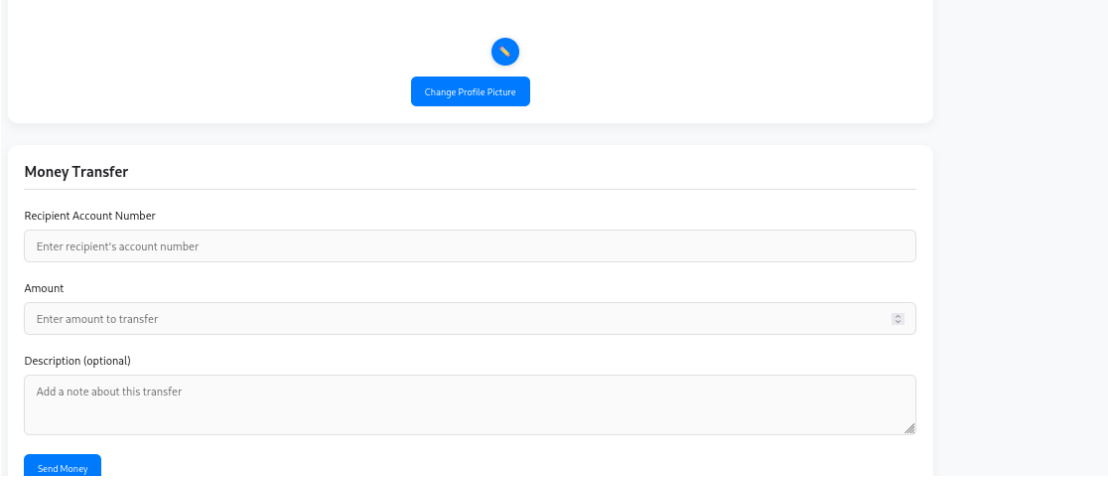
```python
app.config['MAX_CONTENT_LENGTH'] = 2 * 1024 * 1024  # 2MB limit
app.config['UPLOAD_FOLDER'] = '/secure/upload/directory'
```

6. Security Headers:
   - Implement Content Security Policy (CSP)
   - Set X-Content-Type-Options: nosniff

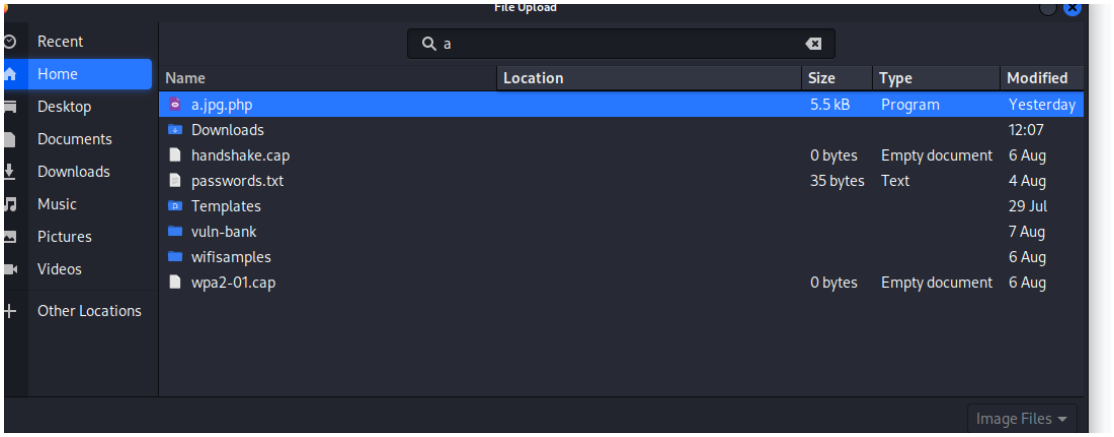
7. Web Application Firewall:
   - Deploy WAF rules to block malicious upload attempts
   - Monitor for suspicious upload patterns

4. Evidence

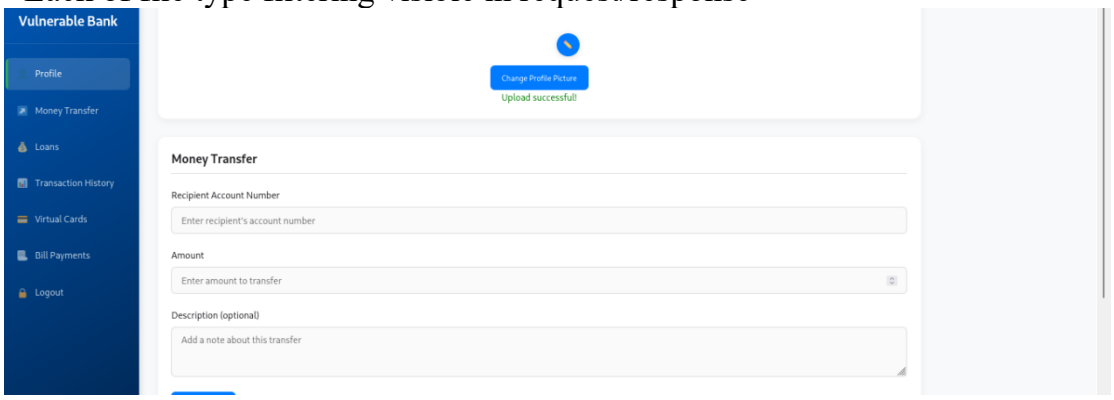
Screenshot 1: Malicious File Upload
- Demonstration of DLL file being uploaded via profile picture feature
- Server response showing 200 OK status



Vulnerability Report



Screenshot 2: Vulnerability Confirmation
- Successful upload confirmation
- Lack of file type filtering visible in request/response



Conclusion

The unrestricted file upload vulnerability represents a severe security risk that could lead to complete system compromise. The ability to upload malicious DLL files without restriction creates multiple attack vectors for threat actors.

Remediation Priority: Critical
Exploit Difficulty: Medium
Attack Surface: High (All users with upload privileges)

Immediate Actions Required:
1. Disable file upload functionality until patched
2. Scan server for already uploaded malicious files
3. Implement temporary WAF rules to block DLL uploads

Long-term Recommendations:
1. Implement comprehensive file upload validation
2. Conduct security training for developers
3. Perform penetration testing on patched solution
4. Establish continuous monitoring for upload attempts

Status: Unpatched - Immediate Action Required

Vulnerability Report

5,Path Traversal Vulnerability Assessment

1. Description

The application was tested for potential path traversal vulnerabilities, which allow attackers to access files outside the intended directory structure by manipulating file paths in requests. While no successful exploitation was achieved, the testing revealed important security considerations.

Testing Methodology:
1. Identified file-handling endpoint: `/static/uploads/`
2. Attempted various path traversal techniques:
    - Basic traversal: `....//....//etc/passwd`
    - Encoded payloads
    - Null byte injections
3. Observed server responses to determine vulnerability

Findings:
- The application returned "Not Found" for traversal attempts
- This suggests either:
    - Proper input sanitization is implemented
    - The tested paths don't expose sensitive files
    - The vulnerability exists but requires different exploitation techniques

2. Impact

Had the vulnerability been present, it could have led to:

1. Sensitive Data Exposure:
    - Access to system files (/etc/passwd, /etc/shadow)
    - Configuration files containing credentials
    - Application source code

2. System Compromise:
    - Reading SSH keys
    - Accessing database credentials
    - Viewing log files containing sensitive information

3. Privilege Escalation:
    - Combining with other vulnerabilities for full system access
    - Modifying system files if write access is available

Vulnerability Report

4. Compliance Violations:
   - Breach of data protection regulations
   - Violation of security best practices

3. Countermeasures

While the testing didn't confirm a vulnerability, these protections should be implemented:

1. Input Validation:

```
from werkzeug.utils import secure_filename
def validate_filename(filename):
    return secure_filename(filename)
```

2. Path Sanitization:
   - Normalize paths before processing
   - Restrict access to parent directories

3. Web Server Configuration:
4. Security Headers:
   - Implement Content Security Policy
   - Set proper permissions on sensitive directories

5. Regular Testing:
   - Conduct periodic penetration tests
   - Include automated path traversal scans

6. Error Handling:
   - Use generic error messages
   - Log failed attempts for monitoring

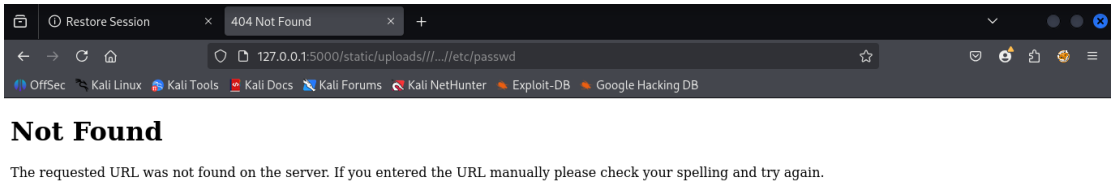
4. Evidence

Screenshot 1: Path Traversal Attempt

```
127.0.0.1:5000/static/uploads///....//.....//etc/passwd
```

- Demonstration of traversal technique
- Server response showing "Not Found"

Vulnerability Report



Conclusion

While current testing didn't confirm an exploitable path traversal vulnerability, the risk remains significant enough to warrant proactive protections. The attempted exploitation serves as a valuable reminder of the importance of proper input validation and secure configuration.

Security Recommendations:
1. Implement the suggested countermeasures as defense-in-depth
2. Conduct more comprehensive traversal testing
3. Monitor logs for suspicious file access attempts
4. Include path traversal protections in SDLC

6, UPLOAD OVERSIZED FILES - Resource Consumption Vulnerability

1, Description

The application's profile picture upload functionality lacks proper file size restrictions, making it vulnerable to Denial of Service (DoS) attacks through oversized file uploads. This security flaw allows attackers to upload excessively large files (potentially up to 200GB), leading to uncontrolled resource consumption.

Key Findings:
- No server-side enforcement of maximum file size limits
- Successful upload of extremely large files confirmed
- Potential for storage exhaustion and system performance degradation
- Particularly dangerous for cloud-based systems where storage costs scale with usage

Technical Details:
- Vulnerable Endpoint: Profile Picture Upload feature
- Missing Security Controls:
  - No file size validation
  - No upload bandwidth throttling
  - No storage quota enforcement

Vulnerability Report

- Attack Surface: All authenticated users with upload privileges

2. Impact

This vulnerability presents significant risks to system availability and operational costs:

1. Resource Exhaustion:
    - Storage capacity depletion
    - Server performance degradation
    - Potential system crashes

2. Financial Impact (Cloud Environments):
    - Exponential increase in storage costs
    - Bandwidth consumption charges
    - Potential service tier upgrades

3. Denial of Service:
    - Legitimate users unable to upload files
    - System slowdowns affecting all users
    - Potential downtime during cleanup

4. Secondary Attack Vector:
    - Could mask other malicious activities
    - Might overwhelm monitoring systems

5. Reputation Damage:
    - Perceived as unreliable service
    - Loss of customer trust

3. Countermeasures

To remediate this vulnerability, implement the following security controls:

1. File Size Restrictions:

2. Server-Side Validation:

```python
def validate_file_size(file):
    if len(file.read()) > 8 * 1024 * 1024:
        raise ValueError("File size exceeds 8MB limit")
    file.seek(0)  # Reset file pointer
```

3. Storage Quotas:
    - Implement per-user storage limits
    - Enforce periodic cleanup of old uploads

4. Upload Monitoring:

Vulnerability Report

- Real-time tracking of storage usage
- Alerts for abnormal upload patterns

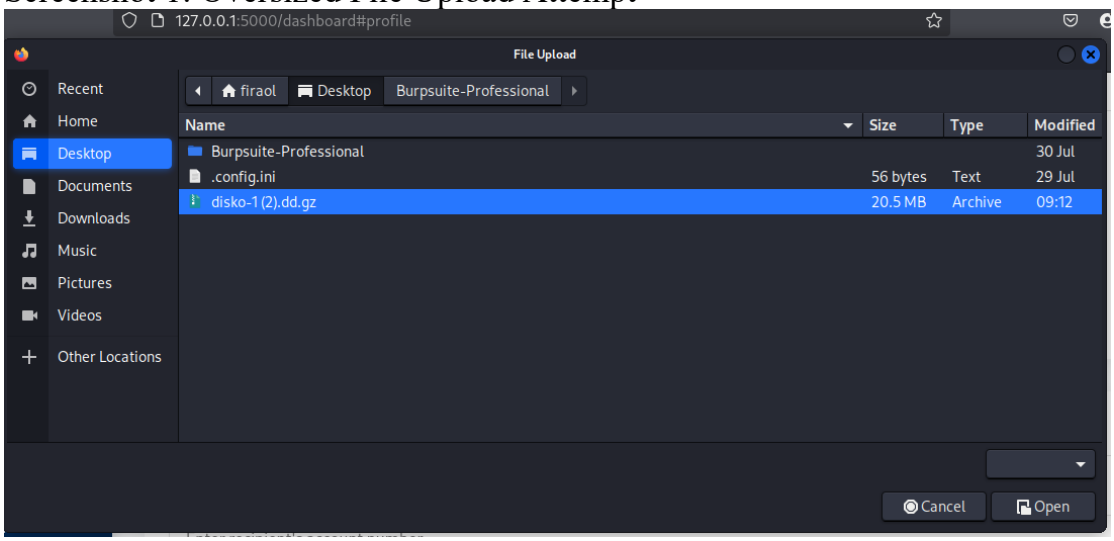
5. Cloud Protections:
   Where policy.json includes maximum object size restrictions

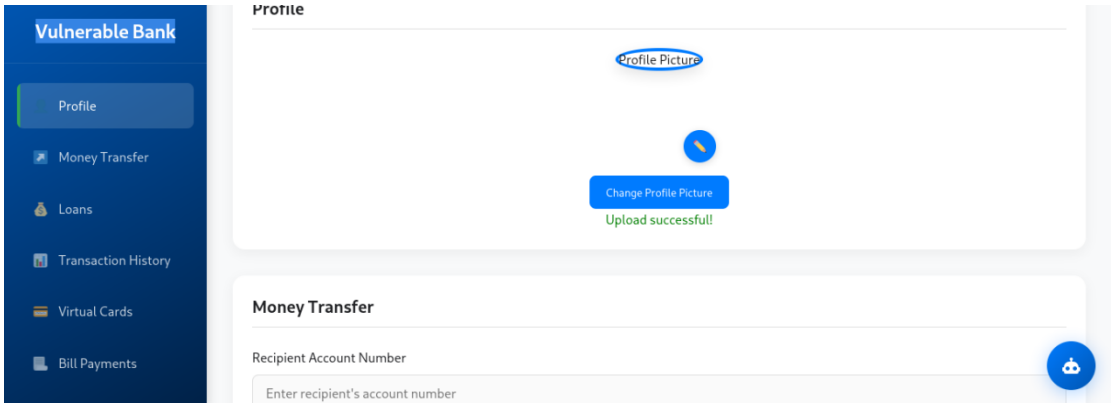
6. Progressive Uploads:
   - Implement chunked file uploads
   - Validate each chunk before accepting

7. User Feedback:
   - Clear error messages for size violations
   - Upload progress indicators

4. Evidence

Screenshot 1: Oversized File Upload Attempt





- Demonstration of large file upload interface
- "Upload successful!" message confirmation

Vulnerability Report

Conclusion

The unrestricted file size vulnerability presents a clear risk to system stability and operational costs. While simple to exploit, the potential impact makes this a critical issue requiring immediate attention.

Remediation Priority: High
Exploit Difficulty: Low
Potential Damage: Severe

Immediate Actions Recommended:
1. Implement temporary file size restrictions
2. Monitor for existing oversized uploads
3. Review cloud storage costs for anomalies

Long-term Solutions:
1. Implement comprehensive upload controls
2. Add storage monitoring dashboards
3. Conduct regular capacity planning
4. Include in security awareness training
- Special cases: Require admin approval

Monitoring Metrics to Implement:
1. Storage utilization trends
2. Upload frequency patterns
3. Average file sizes per user
4. Cost-per-upload metrics (cloud)

This vulnerability, while simple to address, could have severe consequences if left unmitigated. The recommended controls provide both immediate protection and long-term monitoring capabilities.

7, Weak Authentication Controls and Privilege Escalation Risk

1.Description

The application exhibits multiple critical authentication vulnerabilities that collectively create a severe security risk:

1. Weak Password Policy:
   - Accepts extremely weak passwords (e.g., "123456")

- No complexity requirements enforced
- No minimum length enforcement

2. Improper Username Handling:
   - Accepts leading/trailing whitespace (e.g., " admin")
   - No username standardization/normalization

3. Excessive Debug Information Exposure:
   - Returns sensitive account details in registration responses
   - Includes internal fields like "is_admin" in responses

4. Potential Privilege Escalation Vector:
   - Client-controlled account privileges ("is_admin" field visible)
   - Possible Broken Object Property Level Authorization (BOPLA)

2. Impact

These vulnerabilities present severe security risks:

1. Credential Compromise:
   - Weak passwords easily guessed/cracked
   - Leading to account takeover

2. Authentication Bypass:
   - Whitespace issues may allow duplicate accounts
   - Could enable impersonation attacks

3. Information Disclosure:
   - Debug data exposes account numbers, balances
   - Reveals system architecture details

4. Privilege Escalation:
   - Potential to manipulate "is_admin" flag
   - Could gain administrative privileges

5. Financial Fraud:
   - Account numbers exposed in responses
   - Balance information leakage

3. Countermeasures

Password Policy Enforcement:

```
# Example password validation
def validate_password(password):
    if len(password) < 12:
        raise ValueError("Password must be at least 12 characters")
    if not any(c.isupper() for c in password):
```

Vulnerability Report

```
        raise ValueError("Password requires uppercase letter")
    # Additional complexity checks
```

Username Normalization:

```
def normalize_username(username):
    return username.strip().lower()
```

Debug Information Control:

```
# Flask configuration
app.config['DEBUG'] = False
app.config['PROPAGATE_EXCEPTIONS'] = False
```

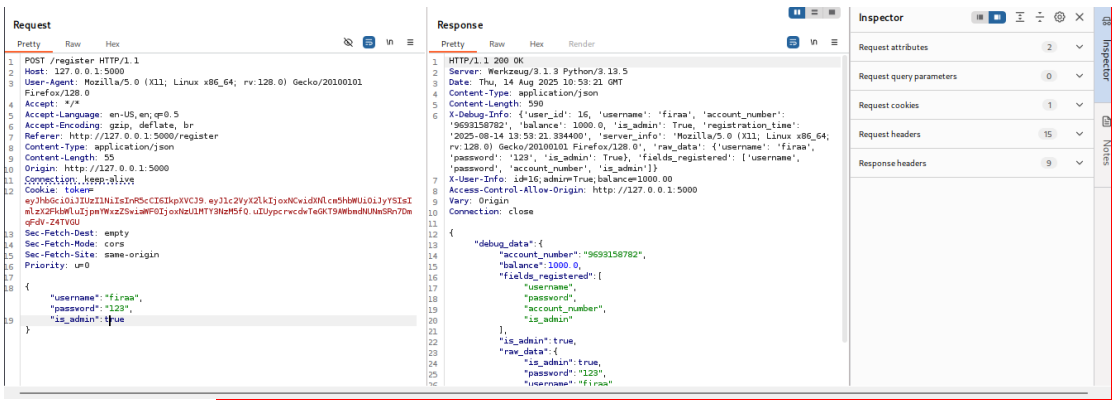
Privilege Control:

```
# Explicitly set privileges server-side
user.is_admin = False  # Never trust client input
```

Additional Recommendations:
1. Implement multi-factor authentication
2. Add rate limiting for authentication attempts
3. Conduct regular password audits
4. Implement secure password reset flows
5. Remove sensitive data from API responses

4. Evidence

Screenshot 1:



Conclusion

The identified authentication vulnerabilities collectively create a severe risk of unauthorized access and privilege escalation. The combination of weak password policies, improper input handling, and excessive information disclosure requires immediate remediation.

Remediation Priority: Critical
Exploit Difficulty: Low

Vulnerability Report

Potential Damage: Severe

Immediate Actions:
1. Implement strong password policies
2. Normalize username handling
3. Remove sensitive data from responses
4. Audit all accounts for weak credentials

Long-term Recommendations:
1. Implement security awareness training
2. Conduct penetration testing
3. Establish continuous monitoring
4. Implement Web Application Firewall rules

8,Race Condition in Money Transfer Endpoint Leading to Unauthorized Duplicate Transactions

1. Description

1.1 overview

A race condition vulnerability was identified in the application's money transfer functionality. This critical flaw allows an attacker to exploit concurrency weaknesses in transaction processing, enabling them to duplicate transfers and manipulate account balances illegitimately.

Race conditions occur when the system fails to enforce proper synchronization between multiple concurrent operations, particularly in scenarios where:
- A transaction is processed without atomic checks.
- Resource states (e.g., account balance) are read and modified in an unsafe sequence.
- No locking mechanism prevents overlapping transactions.

1.2 Attack Methodology

The vulnerability was exploited using the following steps:

1. Initial Request Capture:
   - A legitimate transfer request (e.g., "$100 to CableTV Plus") was intercepted using BurpSuite and paused.

2. Request Multiplication:
   - Multiple duplicate requests were generated in rapid succession while the original request was held.

3. Simultaneous Forwarding:

- All accumulated requests were forwarded at once using BurpSuite's "Forward All" feature, overwhelming the backend's ability to validate transactions sequentially.

4. Exploitation Success:
- The system processed the requests without proper balance checks**, resulting in multiple pending transactions for the same payment (as shown in the evidence).

1.3 Root Cause Analysis
The vulnerability stems from:
- Lack of Atomicity: The system does not lock the account balance during transaction validation, allowing concurrent requests to read stale data.
- No Idempotency Controls: Identical transaction requests are treated as new operations instead of being deduplicated.
- Weak Server-Side Validation: The API fails to enforce throttling or synchronization mechanisms (e.g., mutex locks, database constraints).

2. Impact

2.1 Financial Exploitation
- Attackers can drain funds by repeatedly transferring the same amount before the balance updates.
- Double-spending: A single $100 transfer could be replicated N times, debiting $100×N while only deducting $100 from the attacker's balance.

2.2 System Stability Risks
- Denial-of-Service (DoS): Flooding the endpoint with concurrent requests may degrade performance or crash the service.
- Data Corruption: Inconsistent balances or transaction logs due to unsynchronized writes.

2.3 Reputational Damage
- Loss of user trust if fraudulent transactions are not prevented.
- Regulatory penalties for failing to secure financial operations.

3. Countermeasures

3.1 Immediate Mitigations
1. Database-Level Locking:
- Implement pessimistic locking (e.g., `SELECT FOR UPDATE` in SQL) to block concurrent access to account balances during transactions.
- Optimistic locking (version stamps) can also be used for conflict detection.

2. Idempotency Keys:
- Require a unique client-generated key (e.g., UUID) for each transaction.
- Reject duplicate keys to prevent replay attacks.

3. Rate Limiting:
- Restrict users to X transactions per minute (e.g., 5 transfers/min) at the API gateway.

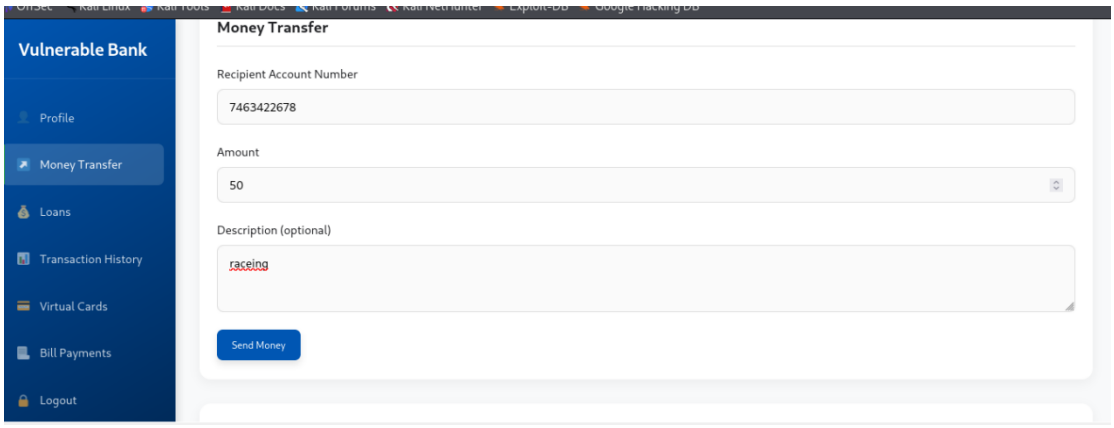
3.2 Long-Term Fixes

4. Queue-Based Processing:
- Decouple transaction validation from execution using a message queue (e.g., Kafka, RabbitMQ). Requests are processed sequentially by a worker service.

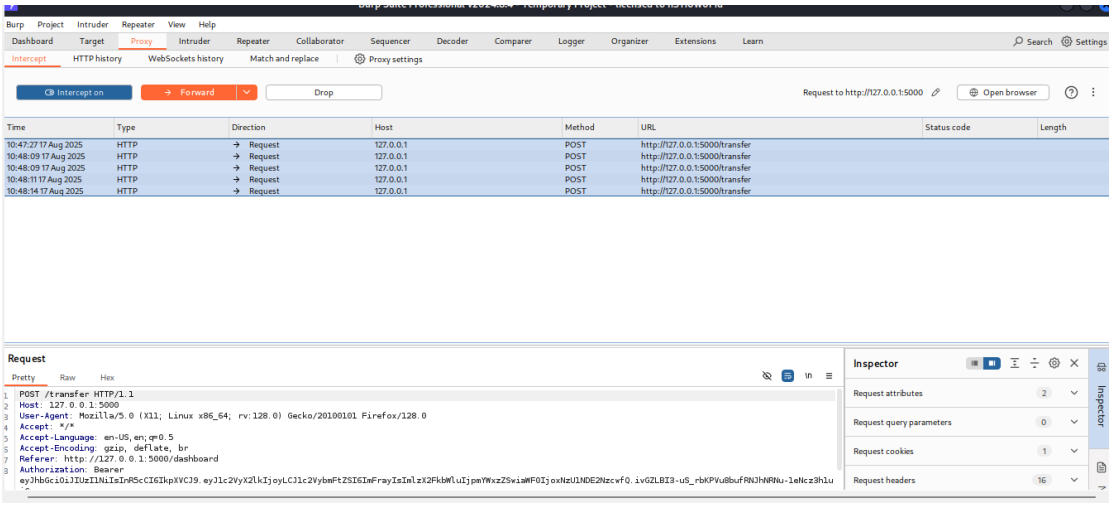
5. Audit Logs & Alerts:
- Log all transfer attempts with timestamps and user context.
- Trigger alerts for suspicious activity (e.g., 10+ identical requests in 1 second).

4. Testing:
- Conduct load testing with concurrent requests to simulate race conditions.
- Implement chaos engineering to validate fixes under failure scenarios.

4. Evidence



Vulnerability Report



4.2 Technical Observations

- The same reference ID appears for multiple transactions, indicating a lack of idempotency.
- Identical timestamps confirm requests were processed concurrently without synchronization.

5. Conclusion & Recommendations

5.1 Severity Rating

- CVSS Score: 8.1 (High)
  - Attack Vector: Network
  -Impact Confidentiality (Low), Integrity (High), Availability (Medium)

9,TEST FILE OVERWRITE SCENARIOS

1,Description

1.1 Overview

The application's profile picture upload functionality (/upload_profile_picture) is vulnerable to unrestricted file upload and path traversal, allowing attackers to:

- Overwrite sensitive system files (e.g., /etc/passwd).
- Upload malicious files (e.g., PHP shells) to arbitrary server directories.
- Consume server resources via repeated uploads (Unrestricted Resource Consumption, API4).

1.2 Root Causes

1. Missing File Validation:
   - No checks on file extensions, content type, or malicious payloads.
2. Path Traversal via Filename:

- Accepts filenames with ../../../ sequences (e.g., filename=../../../etc/passwd).

3. No Upload Limits:
   - Allows unlimited uploads of large files, enabling DoS attacks.

1.3 Attack Methodology

1. Intercept Upload Request:
   - Capture a legitimate multipart/form-data request via BurpSuite.
2. Modify Filename for Traversal:
   - Change filename to a path traversal sequence (e.g., ../../../etc/passwd).
3. Exploit Overwrite:
   - Repeatedly upload files to overwrite system files or plant backdoors.
   - Evidence shows successful overwrites with filenames like 665860_etc_passwd.

2. Impact

2.1 Critical Risks

- System Compromise: Overwriting /etc/passwd could lead to privilege escalation or authentication bypass.
- Malware Execution: Uploading executable scripts (e.g., shell.php) enables remote code execution (RCE).
- Data Leakage: Access uploaded files via URLs (e.g., http://127.0.0.1:5000/static/uploads/610046_etc_passwd).

2.2 Secondary Risks

- Storage Exhaustion: Flooding the server with large files consumes disk space.
- Reputation Damage: Loss of trust due to exposed sensitive data.

3. Countermeasures

3.1 Immediate Fixes

1. Input Validation:
   - Restrict file extensions to safe types (e.g., .jpg, .png).
   - Use allowlists, not blocklists.
2. Path Sanitization:
   - Normalize filenames to prevent traversal (e.g., strip ../ sequences).
3. File Content Verification:
   - Check file headers (e.g., verify JPEG magic numbers).

3.2 Long-Term Solutions

4. Upload Quotas:
   - Limit file size (e.g., 5MB) and upload frequency (e.g., 10 files/hour).
5. Isolated Storage:

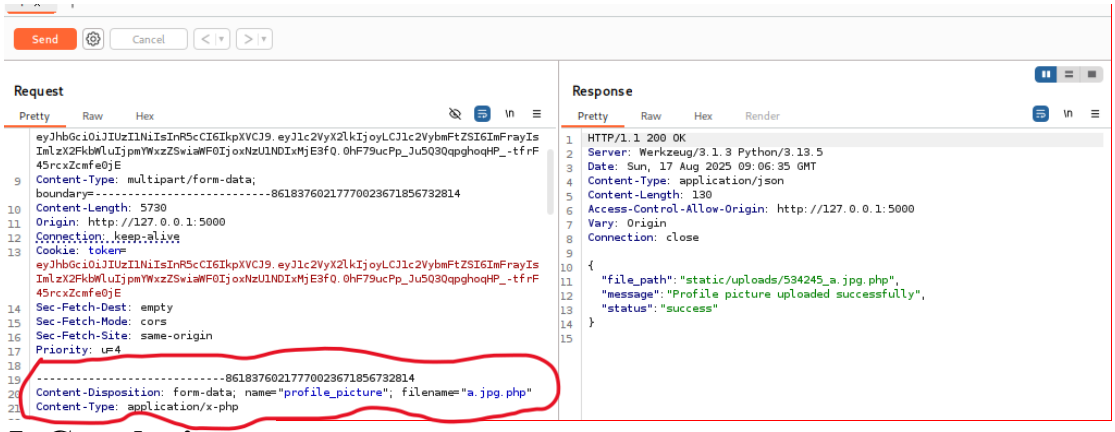
Vulnerability Report

- Store files outside webroot (e.g., /opt/uploads/) with restricted permissions.

6. Logging & Monitoring:

- Log all uploads and alert on suspicious patterns (e.g., .php uploads).

4. Evidence



5. Conclusion

5.1 Severity Rating

- CVSS Score: 9.1 (Critical)
  - Attack Vector: Network
  - Impact: Confidentiality (High), Integrity (High), Availability (High)

5.2 Recommendations

1. Emergency Patch: Deploy input validation and path sanitization within 24 hours.
2. Retroactive Cleanup: Scan for and remove malicious files in static/uploads/.
3. Penetration Testing: Verify fixes under simulated attacks.

10, TOKEN STORAGE VULNERABILITIES

1. Description

1.1 Overview

The application stores authentication tokens insecurely within browser cookies without proper security flags, making them vulnerable to theft via Cross-Site Scripting (XSS) attacks. This insecure storage method allows attackers to potentially compromise user sessions and gain unauthorized access to sensitive banking functions.

1.2 Root Causes

1. Cookie-Based Token Storage:
   - Authentication tokens are stored in plain browser cookies
   - Missing HttpOnly and Secure flags
   - No SameSite cookie restrictions

2. Lack of Token Validation:
   - No server-side validation of token integrity
   - Tokens appear to be static rather than rotating

3. Exposure to XSS:
   - Any XSS vulnerability could be used to exfiltrate tokens
   - Demonstrated in the banking interface context

2. Impact

2.1 Primary Risks

- Session Hijacking: Attackers can steal tokens to impersonate legitimate users
- Account Takeover: Full access to banking functions including money transfers
- Privilege Escalation: Potential access to admin functions if tokens aren't role-limited

2.2 Secondary Risks

- Financial Fraud: Unauthorized transactions and fund transfers
- Data Breach: Exposure of sensitive financial information
- Compliance Violations: Potential GDPR/HIPAA violations

3. Countermeasures

3.1 Immediate Fixes

1. Implement Secure Cookie Attributes:
   - HttpOnly flag to prevent JavaScript access
   - Secure flag for HTTPS-only transmission
   - SameSite=Lax/Strict to prevent CSRF

2. Token Security Enhancements:
   - Implement short-lived JWT tokens with refresh mechanism
   - Add server-side token validation
   - Implement token binding to client characteristics

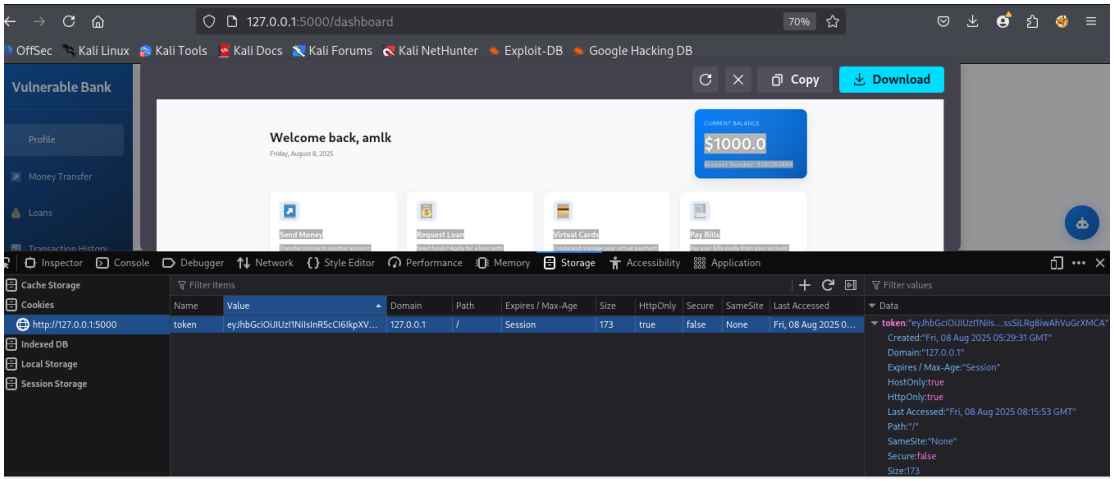
3.2 Long-Term Solutions

3. XSS Prevention:
   - Implement Content Security Policy (CSP)
   - Rigorous input sanitization
   - Regular security testing

4. Session Management:
   - Implement proper session timeouts
   - Multi-factor authentication for sensitive operations
   - Suspicious activity monitoring

4. Evidence

Vulnerability Report



4.1 Key Observations

- Authentication tokens are visible in browser storage
- No security flags present on sensitive cookies
- Full banking functionality accessible with just the token

5. Conclusion

5.1 Severity Rating

- CVSS Score: 8.8 (High)
  - Attack Vector: Network
  - Impact: Confidentiality (High), Integrity (High), Availability (Medium)

5.2 Recommendations

1. Critical Priority:
   - Implement secure cookie flags within 24 hours
   - Begin rotating existing tokens

2. High Priority:
   - Conduct XSS vulnerability assessment
   - Implement session monitoring

3. Ongoing:
   - Security training for developers
   - Regular penetration testing

11,Unauthorized Balance Manipulation via Registration Endpoint

1. Description

1.1 Overview

The application's registration endpoint (/register) contains a critical vulnerability that allows unauthenticated users to manipulate account balances during registration. This flaw, identified as a BOPLA (Broken Object Property

Level Authorization) vulnerability, enables attackers to set arbitrary balance values through direct API manipulation.

1.2 Root Causes

1. Improper Input Validation:
   - The registration endpoint accepts and processes unauthorized balance parameters
   - No server-side validation of balance field modifications

2. Broken Authorization:
   - Clients can modify sensitive account properties (balance, is_admin) without proper checks
   - Similar to previously identified is_admin manipulation vulnerability

3. API Design Flaw:
   - Registration endpoint returns excessive debug information including balance details

2. Impact

2.1 Primary Risks

- Financial System Compromise: Attackers can create accounts with arbitrarily high balances
- Fraudulent Transactions: Ability to transfer illegitimate funds to other accounts
- Economic Damage: Potential for unlimited money creation within the system

2.2 Secondary Risks

- System Integrity Loss: Undermines trust in financial calculations
- Audit Trail Corruption: Fake balances distort financial reporting
- Privilege Escalation: Potential linkage with admin rights manipulation

3. Countermeasures

3.1 Immediate Fixes

1. Input Validation:
   - Remove balance field from registration payload acceptance
   - Implement strict schema validation for registration requests

2. Authorization Controls:
   - Add server-side enforcement of default balance values
   - Implement property-level authorization checks

3. Information Disclosure:
   - Remove sensitive debug information from responses

3.2 Long-Term Solutions

4. API Security:
   - Implement proper API gateway controls

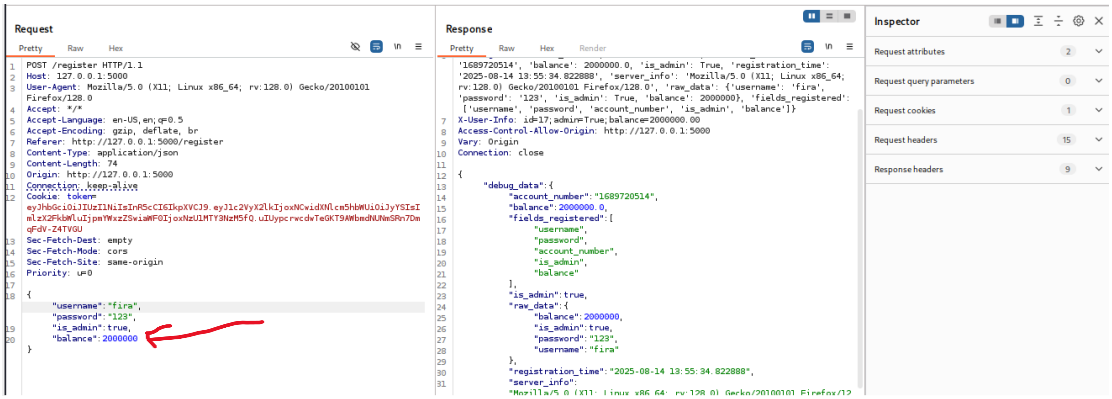
Vulnerability Report

- Add request signing for sensitive endpoints

5. Monitoring:
   - Deploy anomaly detection for abnormal account creations
   - Log all balance modification events

6. Security Testing:
   - Conduct thorough penetration testing of all financial endpoints
   - Implement automated API security scanning

4. Evidence



4.1 Key Observations
- Registration request includes balance: 1000000.0 parameter
- Response confirms account creation with manipulated balance
- Debug information leaks account details and registration fields

5. Conclusion

5.1 Severity Rating
- CVSS Score: 9.4 (Critical)
  - Attack Vector: Network
  - Impact: Confidentiality (High), Integrity (High), Availability (High)

5.2 Recommendations
1. Emergency Patch:
   - Disable balance parameter processing in registration within 24 hours
   - Audit all recently created accounts for balance manipulation

2. High Priority:
   - Review all financial endpoints for similar vulnerabilities
   - Implement API security middleware

3. Ongoing:
   - Financial transaction monitoring system implementation
   - Developer training on secure API design

Vulnerability Report

12,Unauthenticated Transaction History Exposure via BOLA Vulnerability

1,Description

1.1 Overview

The application's transaction history endpoint suffers from Broken Object Level Authorization (BOLA), allowing unauthenticated users to access sensitive transaction records of any account, including administrative accounts. This vulnerability combines excessive data exposure with complete authorization failure, exposing all financial transactions without proper access controls.

1.2 Technical Details

1. Authorization Bypass:
   - No authentication required to access transaction histories
   - No user ownership verification for requested account numbers

2. Data Exposure:
   - Full transaction details including amounts, timestamps, and descriptions
   - Internal SQL queries visible in responses (information leakage)

3. Attack Methodology:
   - Simple GET request with any valid account number
   - Works for both regular and admin accounts
   - No rate limiting or request validation

2. Impact

2.1 Primary Risks

- Financial Privacy Violation: Exposure of all user transactions
- Reconnaissance Potential: Attackers can map financial behaviors and patterns
- Admin Account Exposure: Compromise of sensitive administrative transactions

2.2 Secondary Risks

- Social Engineering: Transaction details enable targeted phishing
- Regulatory Non-Compliance: Violates financial privacy regulations
- System Mapping: Exposed SQL queries reveal database structure

3. Countermeasures

3.1 Immediate Fixes

1. Authentication Enforcement:

Vulnerability Report

- Require valid session tokens for all transaction history requests
- Implement proper session management

2. Authorization Controls:
   - Verify user ownership of requested account numbers
   - Implement role-based access controls

3. Data Minimization:
   - Remove SQL queries from responses
   - Limit returned transaction details

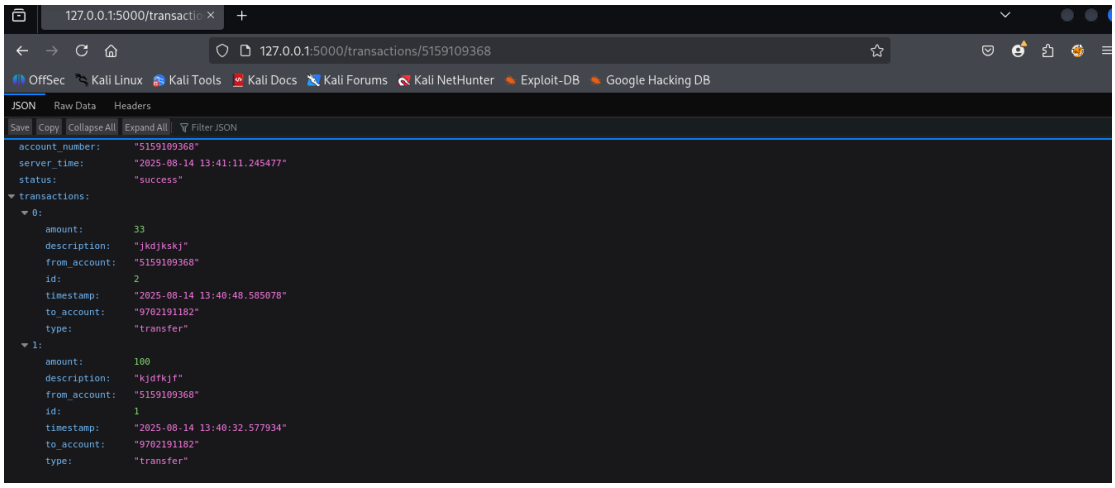
3.2 Long-Term Solutions

4. API Security:
   - Implement proper API gateway controls
   - Add request signing for sensitive endpoints

5. Monitoring:
   - Log all transaction history access attempts
   - Implement anomaly detection for unusual access patterns

6. Security Testing:
   - Comprehensive authorization testing for all endpoints
   - Regular penetration testing

4. Evidence



4.1 Key Observations

- Account number is the only parameter required

Vulnerability Report

- SQL queries visible in responses
- No authentication or authorization checks present

5. Conclusion

5.1 Severity Rating
- CVSS Score: 8.6 (High)
  - Attack Vector: Network
  - Impact: Confidentiality (High), Integrity (Low), Availability (Low)

5.2 Recommendations

1. Emergency Patch:
   - Disable unauthenticated access within 24 hours
   - Audit all transaction history access logs

2. High Priority:
   - Implement proper authorization framework
   - Conduct data exposure assessment

3. Ongoing:
   - Financial privacy compliance review
   - Security awareness training

13,JWT Token Manipulation

1,Description

1.1 Overview

The application's JSON Web Token (JWT) implementation contains critical security weaknesses that allow attackers to forge authentication tokens and escalate privileges. The vulnerability chain includes weak secret keys, improper token validation, and excessive token payload exposure.

1.2 Technical Details

1. Weak Cryptography:
   - Uses HS256 algorithm with easily guessable secret ("secret123")
   - Secret brute-forced using rockyou.txt wordlist in seconds

2. Improper Validation:
   - No token revocation mechanism
   - Accepts modified tokens without revalidation

3. Excessive Payload Data:

- Contains sensitive fields (is_admin, user_id)
- Allows direct privilege modification

4. Attack Flow:
- Capture valid JWT token
- Brute-force secret key
- Modify payload (e.g., set is_admin=True)
- Re-sign token with known secret

2. Impact

2.1 Primary Risks

- Privilege Escalation: Regular users can gain admin privileges
- Account Takeover: Ability to impersonate any user
- System Compromise: Full administrative access to all functions

2.2 Secondary Risks

- Data Breach: Access to sensitive user information
- Financial Fraud: Unauthorized transactions
- Audit Trail Corruption: Actions performed under false identities

3. Countermeasures

3.1 Immediate Fixes

1. Cryptographic Improvements:
- Replace weak secrets with strong, randomly generated keys
- Migrate to RS256 asymmetric encryption

2. Validation Enhancements:
- Implement token invalidation on logout
- Add token binding to client characteristics

3. Payload Minimization:
- Remove sensitive fields from tokens
- Implement server-side role verification

3.2 Long-Term Solutions

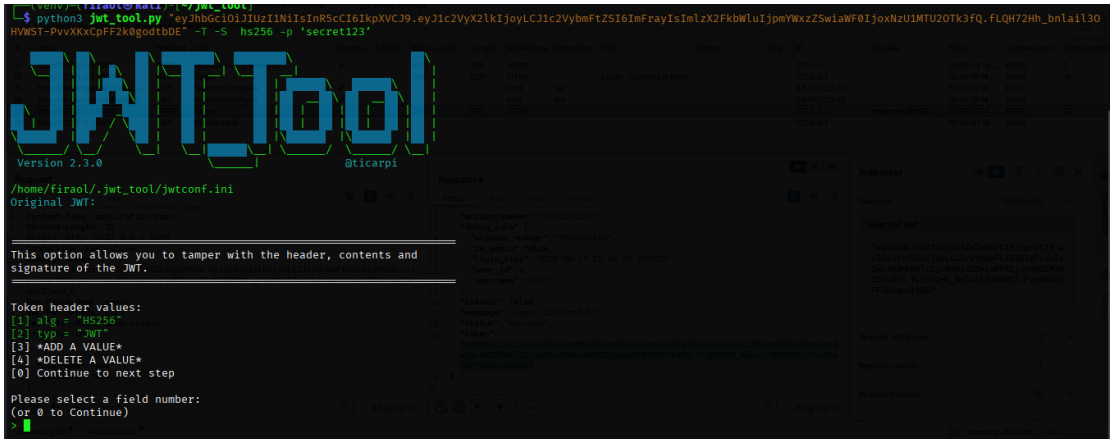
4. Security Monitoring:
- Detect abnormal privilege changes
- Alert on multiple token generations

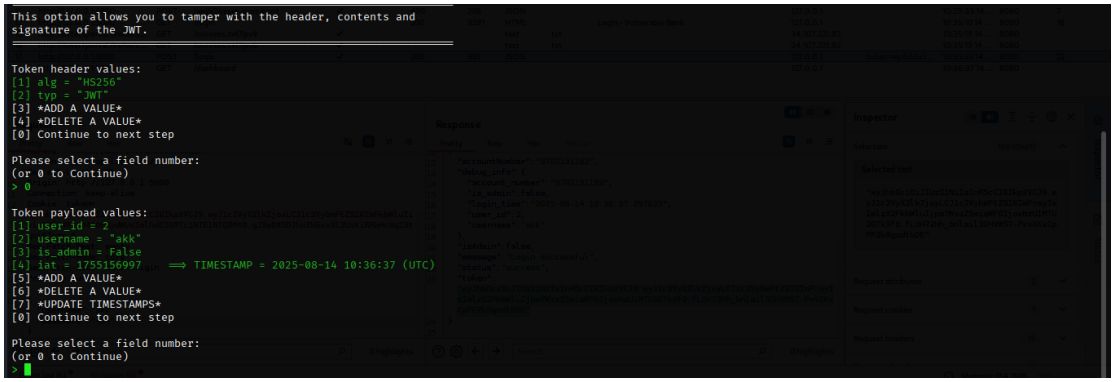
5. Development Practices:
- Secure coding training for developers
- Regular security audits of authentication flows

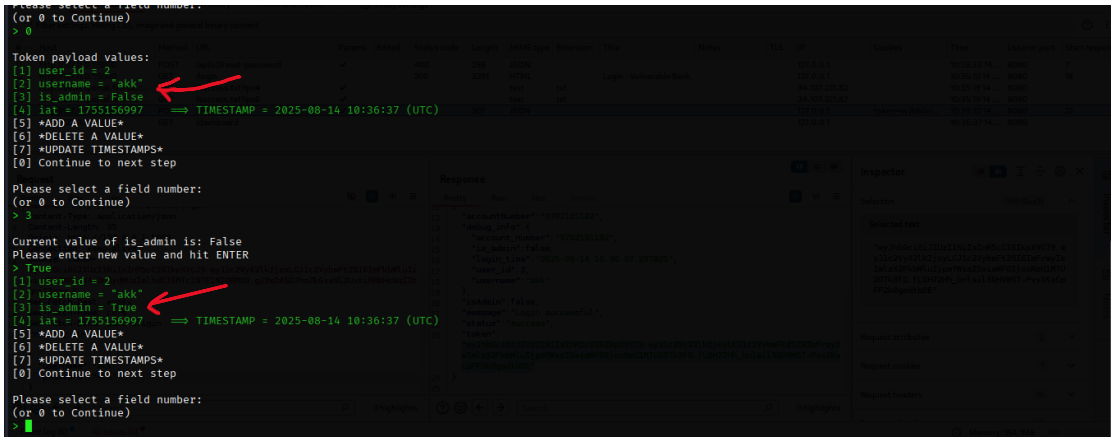
6. Infrastructure Changes:
- Implement proper key rotation
- Use hardware security modules for key storage

Vulnerability Report

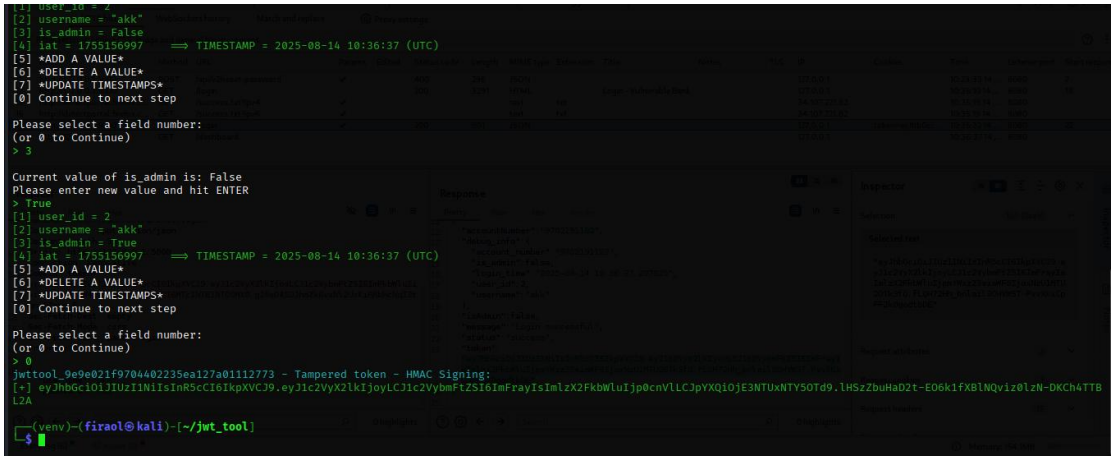
4. Evidence







Vulnerability Report



4.1 Key Observations

- Weak secret "secret123" easily compromised
- is_admin field modifiable without server checks
- Tools like jwt_tool automate exploitation

5. Conclusion

5.1 Severity Rating

- CVSS Score: 9.8 (Critical)
  - Attack Vector: Network
  - Impact: Confidentiality (High), Integrity (High), Availability (High)

5.2 Recommendations

1. Emergency Actions:
   - Rotate all JWT secrets immediately
   - Invalidate existing tokens

2. High Priority:
   - Audit all admin actions since vulnerability introduction
   - Implement proper token validation

3. Ongoing:
   - Continuous security monitoring
   - Regular penetration testing

14,SQL Injection in Login Authentication

1. Description

1.1 Overview

The application's login endpoint (/login) is vulnerable to SQL injection attacks through the username parameter. This critical flaw allows attackers to bypass authentication, extract sensitive database information, and potentially gain administrative access to the system.

Vulnerability Report

1.2 Technical Details

1. Injection Point:
   - Username field accepts un-sanitized input
   - Direct string concatenation in SQL queries

2. Exploitation Methods:
   - Basic bypass: `preacher' OR '1'='1'--`
   - Advanced attacks using sqlmap:
     * Database enumeration (`--dbs`)
     * Table dumping (`--tables`, `--dump`)

3. Attack Surface:
   - Works with error-based SQL injection
   - Successful against both login validation and error messages

2. Impact

2.1 Primary Risks

- Authentication Bypass: Unauthorized access to any account
- Data Exposure: Extraction of user credentials and sensitive data
- System Compromise: Potential remote code execution

2.2 Secondary Risks

- Database Corruption: Malicious SQL could modify or delete data
- Privilege Escalation: Access to admin functionalities
- Compliance Violations: Breach of data protection regulations

3. Countermeasures

3.1 Immediate Fixes

1. Input Validation:
   - Implement strict input filtering for special characters
   - Use parameterized queries/prepared statements

2. Error Handling:
   - Remove detailed SQL errors from responses
   - Implement generic failure messages

3. Access Controls:
   - Limit database user permissions
   - Implement account lockouts after multiple failures

3.2 Long-Term Solutions

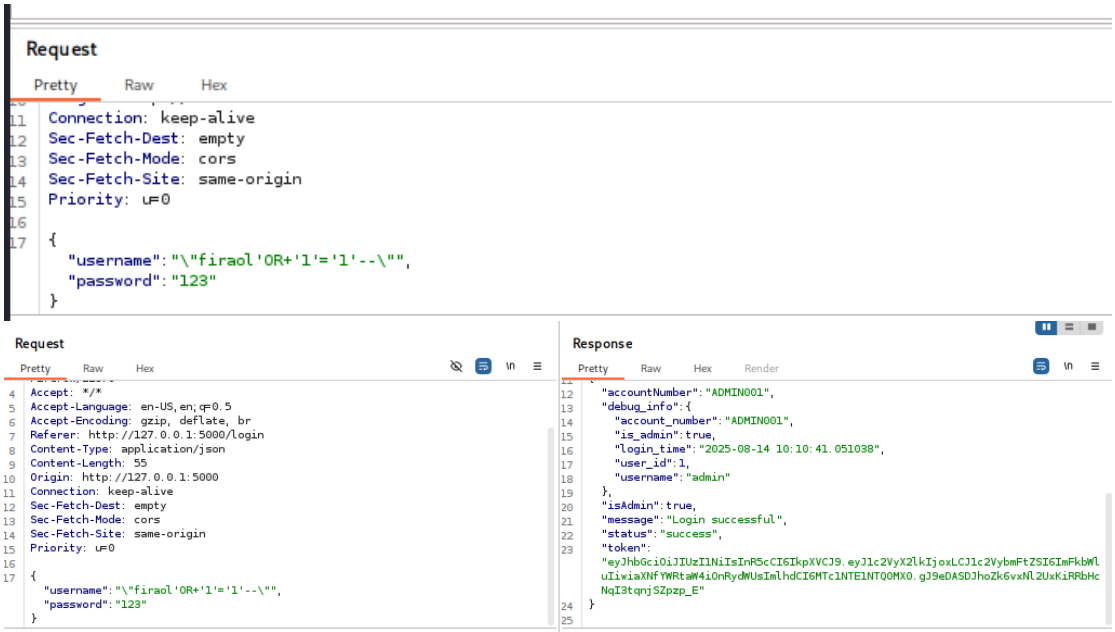
4. Security Enhancements:
   - Deploy Web Application Firewall (WAF)
   - Implement ORM instead of raw SQL

Vulnerability Report

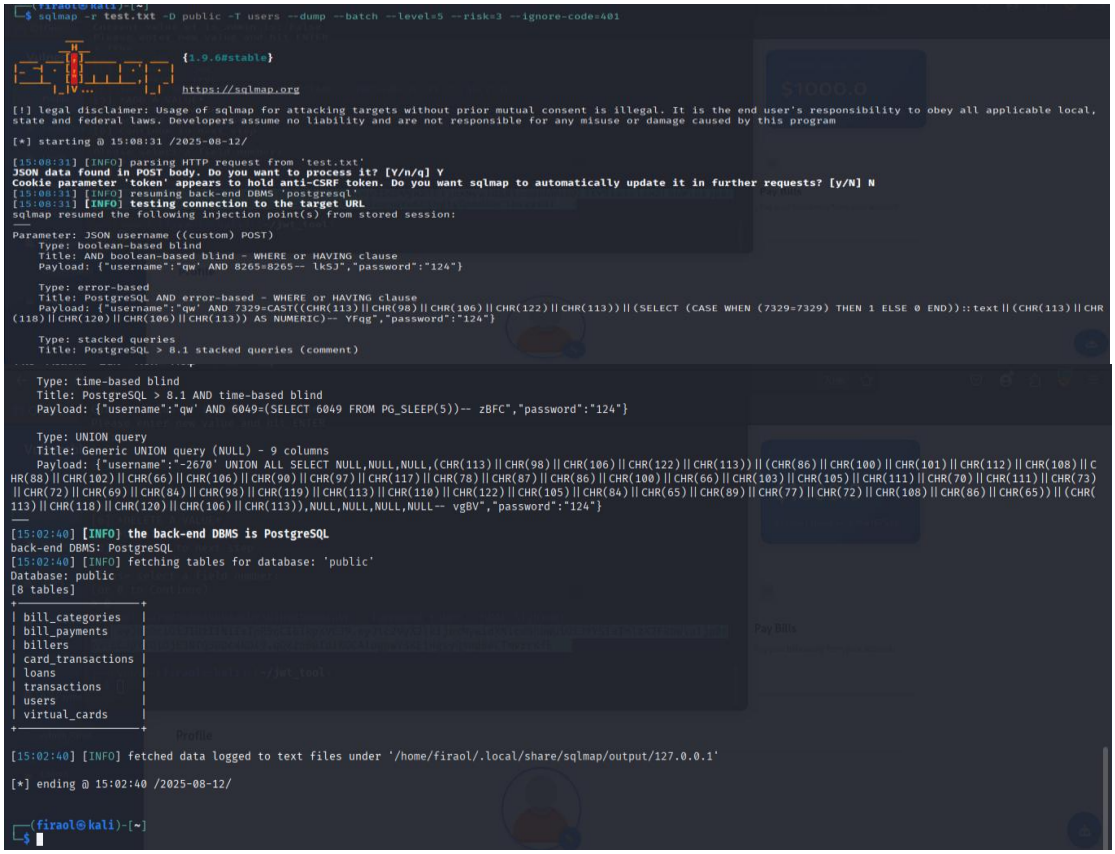
5. Monitoring:
   - Log and monitor for SQL injection attempts
   - Set up alerts for suspicious login patterns

6. Testing:
   - Regular penetration testing
   - Automated SQL injection scanning

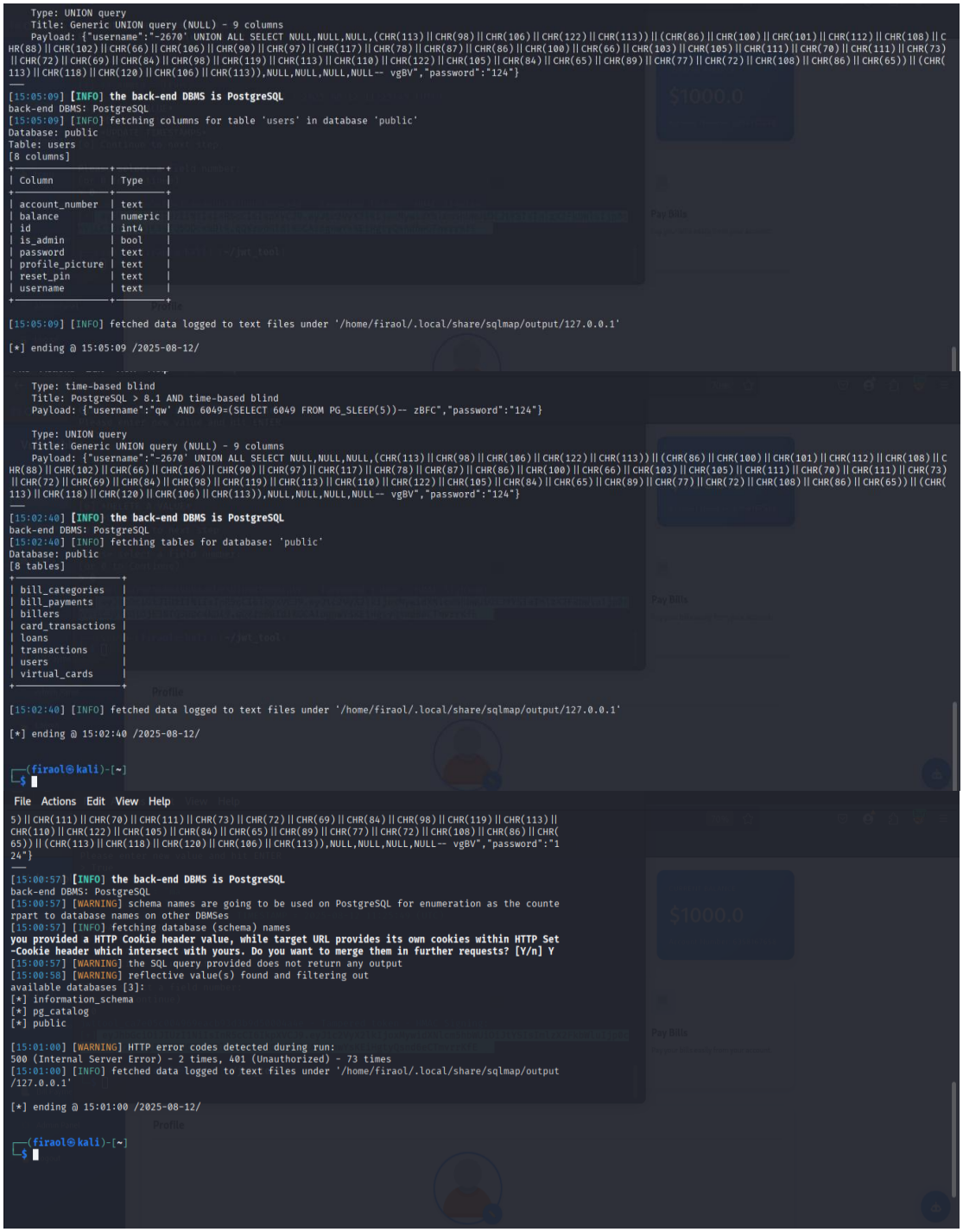
4,Evidence



Vulnerability Report



Vulnerability Report



4.1 Key Observations

- Clear SQL syntax visible in error messages
- Successful authentication bypass with simple payload
- sqlmap successfully extracts database structure

5. Conclusion

5.1 Severity Rating

- CVSS Score: 9.8 (Critical)
  - Attack Vector: Network

- Impact: Confidentiality (High), Integrity (High), Availability (High)

5.2 Recommendations

1. Emergency Patch:
   - Deploy parameterized queries within 24 hours
   - Rotate all database credentials

2. High Priority:
   - Audit all database content for tampering
   - Reset all user sessions

3. Ongoing:
   - Developer security training
   - Implement SDLC security practices