October 19, 2026

Penetration Test Report

CONFIDENTIAL

BlackBox



Who we are... Zazmic Security Team

Zazmic is a Premier Google Cloud Partner dedicated to uncovering risks and establishing clear guides for remediation.

What services we are offering:

- Penetration Testing - Black box, Gray Box, White Box
- Agentic Pentest and Hybrid Pentest
- Cloud Security Posture Assessment for GCP, AWS, Azure
- CPA - Cost Posture Assessment for GCP infrastructure
- Application Security Assessment - SAST, DAST, Container Security
- Support SOC2, FedRAMP, ISO 27001, ISO 9001, HIPAA, HITRUST compliance preparation and Audit
- Google SecOps and Google Security Command Center Premium licensing, set up and optimization

Contact Us:

Email: **sales@zazmic.com**
Join our security platform: **https://security.zazmic.com**

Strategic Recommendation

While the current assessment provides a vital overview of your external defenses, Zazmic recommends evolving your testing strategy to include Grey Box Penetration Testing.

Contact us for more information at sales@zazmic.com.

The Limitation of Black Box Testing

Traditional "Black Box" testing only scratches the surface. It does not account for the sophisticated "assume breach" scenarios that modern organizations face.

Why Grey Box is Recommended

- **Deepen the Search:** Identify lateral movement risks and privilege escalation.
- **Simulate Realistic Threats:** Mimic a compromised employee account accurately.
- **Efficiency:** Focus on high-value logic flaws rather than discovery.

We suggest scheduling a Grey Box test to validate the remediation of the risks identified in this report.

Table of Contents

1 Document Control **6**

1.1 Team 6

1.2 List of Changes 6

1.3 Assessment Details 6

2 Executive Summary **7**

2.1 Vulnerability Distribution 7

2.2 Master Findings Table 8

2.3 Scope 10

2.4 User Accounts and Permissions 10

3 Detailed Findings **11**

C-1: JWT Signature Not Verified — Full Token Forgery 11

C-2: JWT Claim Tampering — Admin Privilege Escalation Confirmed 12

C-3: JWT Signing Secret Cracked — Full Token Forgery Possible 13

C-4: Command Injection — /login (username) 14

C-5: SQL Injection (Auth Bypass) — /login (username) 15

C-6: API Fuzzing (sensitive_data_disclosure) — POST /api/v1/forgot-password (HTTP 200) 16

C-7: Negative Value Transfer — /transfer 17

C-8: Exposed Sensitive File — /.env 18

C-9: Exposed Sensitive File — /console 19

C-10: Race Condition: Overdraft — /transfer 20

H-1: Information disclosure: debug endpoint exposes account/user data 21

H-2: Session JWT signed with a weak/default secret (HS256) 22

H-3: Unauthenticated BOLA on /check_balance/<id> 23

H-4: Unauthenticated BOLA on /transactions/<id> 24

H-5: SQL Injection (Error-Based) — /transactions/{accountNumber} (accountNumber) 25

H-6: Mass Assignment — /register 26

H-7: Brute-forceable 3-digit password-reset PIN (PIN leaked in response) 27

H-8: Unrestricted file upload (dangerous extension accepted) 28

H-9: BOLA (idor) — /api/transactions?account_number={id} 29
H-10: BOLA (idor) — /transactions/{accountNumber} 30
H-11: BOLA — Cross-User Resource Access — /transactions/{username} 31
H-12: DOM-based Cross-Site Scripting in Client-Side Rendering 32
M-1: Missing Security Headers 33
M-2: JWT Has No Expiry — Permanent Token 34
M-3: No rate limiting / account lockout on login endpoint 35
M-4: Hidden admin endpoint reachable via predictable path /sup3r_s3cr3t_admin 36
L-1: Framework / version disclosure 37
A Appendix 38





1 Document Control

1.1 Team

Contact	Details	Role
Bradley Clunie	E-Mail: bradley.clunie@zazmic.com	Pentester

1.2 List of Changes

Version	Description	Date
1.0	Initial release	Oct 19, 2026

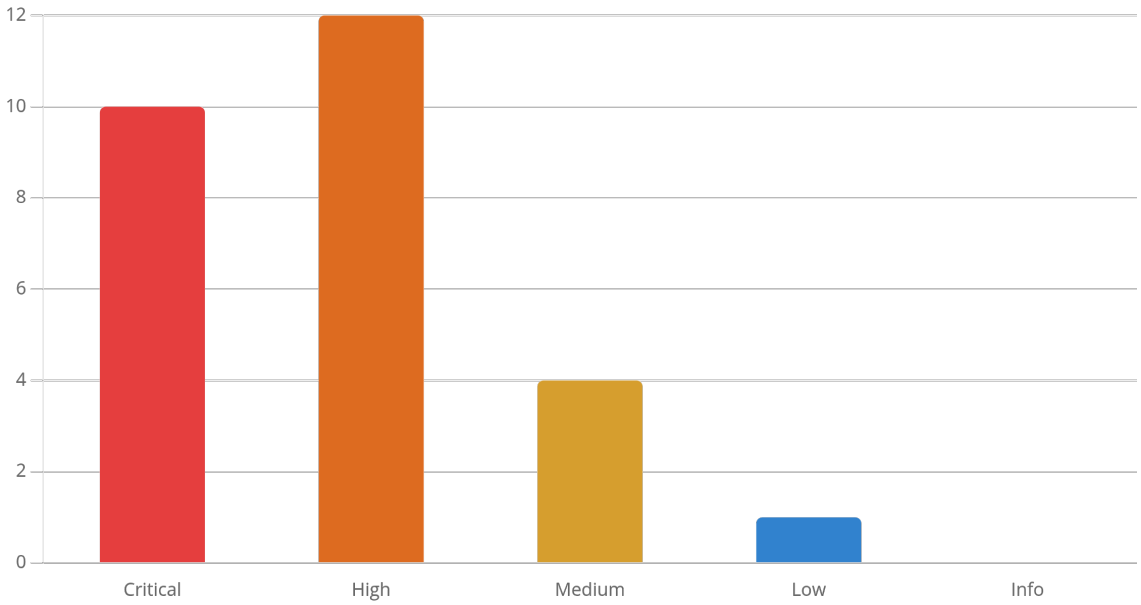
1.3 Assessment Details

Conduct Dates	October 19, 2026 to October 19, 2026
Duration	1 day

2 Executive Summary

This report presents the results of a black-box, unauthenticated penetration test of a web application, performed from the perspective of an external attacker with no credentials or prior knowledge of the system. The assessment identified 27 findings: 10 critical, 12 high, 4 medium, and 1 low. The most serious issues permit full authentication bypass and privilege escalation (session tokens can be forged and their claims tampered with), remote command execution and SQL injection through the login interface, exposure of sensitive configuration files, and business-logic flaws allowing unauthorized movement of funds. Taken together, these would let an unauthenticated attacker take over accounts, read other users data, and manipulate transactions. Detailed findings and prioritised remediation guidance follow.

2.1 Vulnerability Distribution



Critical	High	Medium	Low
10	12	4	1

Calculated Score: Negative/<40 — Security Grade: E

Security Deficit; fundamental flaws requiring immediate rebuild.

In the course of this penetration test **10 Critical**, **12 High**, **4 Medium** and **1 Low** vulnerabilities were identified:

- **Critical-severity issues** require immediate attention and pose the highest risk to the organization's security posture.





- **High-severity issues** represent significant vulnerabilities that could lead to data breaches, unauthorized access, or financial impact.
- **Medium-severity issues** indicate weaknesses that could be exploited in combination with other vulnerabilities or under specific conditions.
- **Low-severity issues** represent minor weaknesses that should be addressed as part of ongoing security hygiene.

2.2 Master Findings Table

ID	Title	State	Severity
C-1	JWT Signature Not Verified — Full Token Forgery	Open	Critical
C-2	JWT Claim Tampering — Admin Privilege Escalation Confirmed	Open	Critical
C-3	JWT Signing Secret Cracked — Full Token Forgery Possible	Open	Critical
C-4	Command Injection — /login (username)	Open	Critical
C-5	SQL Injection (Auth Bypass) — /login (username)	Open	Critical
C-6	API Fuzzing (sensitive_data_disclosure) — POST /api/v1/forgot-password (HTTP 200)	Open	Critical
C-7	Negative Value Transfer — /transfer	Open	Critical
C-8	Exposed Sensitive File — /.env	Open	Critical
C-9	Exposed Sensitive File — /console	Open	Critical
C-10	Race Condition: Overdraft — /transfer	Open	Critical
H-1	Information disclosure: debug endpoint exposes account/user data	Open	High
H-2	Session JWT signed with a weak/default secret (HS256)	Open	High
H-3	Unauthenticated BOLA on /check_balance/<id>	Open	High
H-4	Unauthenticated BOLA on /transactions/<id>	Open	High
H-5	SQL Injection (Error-Based) — /transactions/{accountNumber} (accountNumber)	Open	High
H-6	Mass Assignment — /register	Open	High
H-7	Brute-forceable 3-digit password-reset PIN (PIN leaked in response)	Open	High
H-8	Unrestricted file upload (dangerous extension accepted)	Open	High
H-9	BOLA (idor) — /api/transactions?account_number={id}	Open	High

ID	Title	State	Severity
H-10	BOLA (idor) — /transactions/{accountNumber}	Open	High
H-11	BOLA — Cross-User Resource Access — /transactions/{username}	Open	High
H-12	DOM-based Cross-Site Scripting in Client-Side Rendering	Open	High
M-1	Missing Security Headers	Open	Medium
M-2	JWT Has No Expiry — Permanent Token	Open	Medium
M-3	No rate limiting / account lockout on login endpoint	Open	Medium
M-4	Hidden admin endpoint reachable via predictable path / sup3r_s3cr3t_admin	Open	Medium
L-1	Framework / version disclosure	Open	Low





2.3 Scope

Black-box (unauthenticated) security assessment of a production web application. Testing covered authentication, session management, access control, injection, business logic, file handling, and information exposure. No credentials or source code were provided.

2.4 User Accounts and Permissions

No user accounts or credentials were provided for this engagement. The assessment was conducted entirely as a black-box, unauthenticated test that simulates an external attacker with no prior access to the application, no knowledge of its internal design, and no valid session. All testing was performed against the publicly reachable surface of the application. Where weaknesses allowed authenticated access to be obtained, for example authentication bypass or session-token forgery, those paths were exercised to demonstrate real impact, but at no point were legitimate credentials supplied by the client.

3 Detailed Findings

C-1: JWT Signature Not Verified — Full Token Forgery

Critical

Identified on: October 19, 2026

Affected Components: /api/transactions

Google Security Product: TODO: None

Description

The application fails to cryptographically verify the signature of JSON Web Tokens (JWTs) submitted for authentication. This vulnerability was identified at the /api/transactions endpoint, which incorrectly processes tokens with invalid signatures. An unauthenticated attacker can modify the claims within a JWT payload, such as elevating privileges by setting an administrative flag, and submit it to the server. Because the signature is not checked, the forged token is accepted, allowing the attacker to bypass authentication and authorization controls to access protected resources and functionality.

How to Exploit

1. Obtain a valid JWT used by the application, or observe its structure to craft a sample token.
2. Decode the header and payload components of the token from Base64 to view their JSON content.
3. Modify the JSON payload to include a privileged claim, for example, by adding `"is_admin": true`.
4. Re-encode the modified payload section using Base64.
5. Construct a new, forged token by concatenating the original Base64-encoded header, a period (`.`), the new Base64-encoded payload, a period (`.`), and an arbitrary string for the signature (e.g., `invalid-signature`).
6. Send a request to the `/api/transactions` endpoint with the forged token in the `Authorization: Bearer` header.
7. Observe that the server responds with a status code (e.g., 400 Bad Request) that indicates the request was processed past the authentication stage, rather than rejected with a `401 Unauthorized` or `403 Forbidden` error. This confirms the signature was not validated.

Recommendation

Enforce authentication and authorisation server-side on all sensitive endpoints. Apply rate limiting and use strong session management practices.

C-2: JWT Claim Tampering — Admin Privilege Escalation Confirmed

Critical

Identified on: October 19, 2026

Affected Components: /dashboard

Google Security Product: TODO: None

Description

The application's authentication mechanism relies on JSON Web Tokens (JWTs) signed with a weak, predictable secret key. The secret was identified as "secret123" through offline brute-force attacks. This weakness allows an unauthenticated attacker to forge valid JWTs with arbitrary claims. By creating a token with administrative privileges (e.g., `role=admin`, `is_admin=true`), an attacker can bypass authorization controls. Successful exploitation grants the attacker full administrative access to protected resources, including the `/dashboard` and `/sup3r_s3cr3t_admin` endpoints.

How to Exploit

1. Obtain any valid JWT issued by the application during normal use.
2. Decode the token to identify the payload structure and signing algorithm.
3. Perform an offline dictionary or brute-force attack against the token's signature to discover the secret key. The key was confirmed to be "secret123".
4. Construct a new JWT payload containing claims for elevated privileges, such as `{"role": "admin", "is_admin": true}`.
5. Sign the new payload using the cracked secret "secret123" and the original signing algorithm.
6. Issue a request to the `/dashboard` endpoint, including the forged token in the `Authorization` header (e.g., `Authorization: Bearer [FORGED_TOKEN]`).
7. Observe that the server accepts the token and grants access to the administrative functionality, confirming a successful privilege escalation.

Recommendation

Validate `alg` and all critical claims server-side on every request. Use a well-maintained JWT library that rejects tampered tokens by default (e.g. python-jose, PyJWT ≥2.x).

C-3: JWT Signing Secret Cracked — Full Token Forgery Possible

Critical

Identified on: October 19, 2026

Affected Components: /login

Google Security Product: TODO: None

Description

The application's authentication mechanism utilizes JSON Web Tokens (JWTs) signed with the symmetric HS256 algorithm. The secret key required for this cryptographic signing operation was discovered to be a weak, easily guessable value: "secret123". This allows an unauthenticated attacker to forge valid JWTs with arbitrary payload claims. Consequently, an attacker can bypass authorization controls by creating tokens with elevated privileges, such as an administrative role, leading to a complete compromise of application accounts and data.

How to Exploit

1. Capture a valid JWT issued after a successful authentication request to the /login endpoint. 2. Decode the token and confirm the header specifies the 'HS256' algorithm. 3. Perform an offline dictionary attack against the token's signature to derive the signing secret. The secret will be successfully cracked as "secret123". 4. Construct a new JWT payload with modified claims to grant elevated privileges (e.g., `{"role":"admin"}`). 5. Use the original token header, the new payload, and the cracked secret "secret123" to generate a new signature and create a forged token. 6. Send a request to an access-controlled endpoint using the forged JWT in the Authorization header. 7. Confirm that the application processes the request and grants access based on the privileges defined in the forged token's payload.

Recommendation

Replace the weak JWT secret immediately with a cryptographically random 256-bit value (e.g. `secrets.token_hex(32)`). Rotate all active sessions. Consider switching to RS256 for asymmetric signing.

C-4: Command Injection — /login (username)

Critical

Identified on: October 19, 2026

Affected Components: /login

Google Security Product: TODO: None

Description

An operating system (OS) command injection vulnerability was identified in the `username` parameter of the `/login` endpoint. The application passes user-supplied data from this parameter directly to a system shell command without sufficient sanitization, allowing an unauthenticated attacker to execute arbitrary commands on the server. Successful exploitation grants the attacker the ability to read sensitive files, exfiltrate data, and potentially gain further access to the internal network. This vulnerability provides the attacker with command execution capabilities under the same privileges as the web application process.

How to Exploit

1. Using a tool capable of crafting and sending custom HTTP requests, prepare a GET request to the `/login` endpoint. 2. In the `username` query parameter, insert the payload `; echo CMDINJMARKER`. 3. URL-encode the payload; the resulting request URI will be `/login?username=probe%3B%20echo%20CMDINJMARKER`. 4. Send the crafted GET request to the application server. 5. Inspect the body of the HTTP response returned by the server. 6. Confirm that the response body contains the string `CMDINJMARKER`. The presence of this marker, which is the output of the injected `echo` command, confirms the vulnerability.

Recommendation

The login handler appears to pass the username to a shell command. Avoid calling shell commands with user-controlled input. If necessary, use safe APIs like Python's `subprocess` module with `shell=False` and pass arguments as a list, e.g., `subprocess.run(['/usr/bin/some_command', username])`.

C-5: SQL Injection (Auth Bypass) — /login (username)

Critical

Identified on: October 19, 2026

Affected Components: /login

Google Security Product: TODO: None

Description

The login functionality is vulnerable to SQL injection in the 'username' field. The application fails to properly sanitize or parameterize user-supplied input before incorporating it into an SQL query, allowing an attacker to manipulate the query's logic. By submitting a crafted payload, an attacker can bypass the authentication mechanism entirely and gain unauthorized access to the application. The vulnerability was further leveraged to enumerate the database schema, confirming a PostgreSQL backend and identifying tables containing sensitive information such as 'users', 'transactions', and 'virtual_cards', indicating a high risk of complete data exfiltration.

How to Exploit

1. Using a web request tool, prepare a POST request to the /login endpoint.
2. Set the 'Content-Type' header to 'application/json'.
3. Construct the request body with the following JSON payload to attempt an authentication bypass: {"username": "' OR '1'='1", "password": "password"}.
4. Send the request to the server.
5. Observe that the server responds by granting access, typically indicated by a session token or a redirect to an authenticated area, confirming the bypass.
6. To further confirm the injection point, send a second POST request to the same endpoint with the body: {"username": "'", "password": "password"}.
7. Verify that the server returns a 500 status code and an error message related to a database syntax error, which confirms the unhandled injection of the single quote.

Recommendation

The login query is vulnerable to SQL injection. Use parameterized queries (prepared statements) to prevent attackers from manipulating the SQL statement. For example, in Python with psycopg2, use `cursor.execute("SELECT * FROM users WHERE username = %s", (username,))` instead of string formatting to construct the query.

C-6: API Fuzzing (sensitive_data_disclosure) — POST /api/v1/forgot-password (HTTP 200)

Critical

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

The POST /api/v1/forgot-password endpoint improperly handles malformed requests. When the endpoint is sent a fuzzed payload containing unexpected data types or structures, the application responds with an HTTP 200 OK status instead of an appropriate client error code. This anomalous success response exposes sensitive information, such as verbose error messages or stack traces, in the response body. This information disclosure provides an attacker with internal details about the application's architecture and technology stack, which can be used to identify and exploit other vulnerabilities.

How to Exploit

1. Use a web proxy or API client to construct a manual HTTP request.
2. Set the request method to POST and the target URL to the full path of the /api/v1/forgot-password endpoint.
3. Set the 'Content-Type' header to 'application/json'.
4. In the request body, provide a fuzzed payload. For example, if the endpoint expects a user identifier, send an array instead of a string: `{"email": []}`.
5. Send the request to the application server.
6. Observe the HTTP response from the server.
7. Confirm that the HTTP status code is 200 OK.
8. Inspect the response body and confirm it contains sensitive data, such as a stack trace, internal file paths, or detailed debugging information.

Recommendation

Validate all request payloads against a strict schema. Apply authentication middleware globally. Return generic error messages without stack traces.

C-7: Negative Value Transfer — /transfer

Critical

Identified on: October 19, 2026

Affected Components: /transfer

Google Security Product: TODO: None

Description

The money transfer functionality at the `/transfer` endpoint fails to validate that the transaction amount is a positive number. By submitting a POST request with a negative value in the `amount` field, the application processes the transaction incorrectly. Instead of debiting the sender's account, the negative value is subtracted from the balance, effectively crediting the account. This flaw allows an attacker to arbitrarily increase their own account balance by transferring negative amounts, leading to direct financial fraud.

How to Exploit

1. Log in to the application and note the current account balance.
2. Navigate to the feature responsible for transferring funds to another user.
3. Initiate a transfer to any valid recipient account.
4. Intercept the outbound POST request destined for the `/transfer` endpoint.
5. Modify the value of the `amount` parameter in the request body to a negative number, for example, `-100`.
6. Forward the modified request to the server.
7. Observe the server's response, which should indicate a successful transaction (e.g., an HTTP 200 status with a body containing `"status": "success"`).
8. Refresh your account balance and confirm that it has increased by the absolute value of the negative amount transferred (e.g., by 100.0).

Recommendation

Validate that monetary and quantity fields are strictly positive before processing any transaction. Enforce constraints at both the API layer and the database (CHECK constraint).

C-8: Exposed Sensitive File — /.env

Critical

Identified on: October 19, 2026

Affected Components: /.env

Google Security Product: TODO: None

Description

A sensitive environment configuration file, `.env`, was found to be publicly accessible at the web root of the application. This file contains critical secrets, including database connection credentials, a Flask secret key, and a JWT secret. An attacker with access to this file could directly connect to the application's database, potentially leading to a complete compromise of all application data. The file also reveals internal infrastructure details and indicates that the application is running with debug mode enabled in a production environment, increasing the risk of further information disclosure.

How to Exploit

1. Open a web browser or use a command-line tool capable of making HTTP requests, such as `curl`. 2. Construct the full URL for the target file by appending `/.env` to the application's base hostname. 3. For example, navigate to `https://[application_hostname]/.env`. 4. Send a GET request to this URL. 5. Examine the HTTP response received from the server. 6. Verify that the response status code is 200 OK and that the response body contains the plain text contents of the environment file, including key-value pairs like `SECRET_KEY` and `DB_PASSWORD`.

Recommendation

Remove /.env from the web root and add it to .gitignore. Store secrets in environment variables injected at runtime (e.g. via a secrets manager) and never commit them to source control.

C-9: Exposed Sensitive File — /console

Critical

Identified on: October 19, 2026

Affected Components: /console

Google Security Product: TODO: None

Description

The web application exposes an interactive debugging console at the `/console` endpoint. The response body indicates a Werkzeug Debugger is active, a component intended for development environments. This console allows any user who can access it to execute arbitrary code within the security context of the web application process. Successful exploitation could lead to a complete compromise of the application server, allowing an attacker to access or modify data, interact with backend systems, and potentially pivot to other internal network resources. The exposure of such a powerful administrative tool presents a critical risk to the application and its underlying infrastructure.

How to Exploit

1. Open a web browser or use a command-line tool like `curl`.
2. Construct the full URL by appending the affected path to the application's hostname (e.g., `https://[hostname]/console`).
3. Send an HTTP GET request to the constructed URL.
4. Observe the server's response.
5. Confirm that the HTTP status code is 200 OK.
6. Inspect the HTML content of the response body.
7. Verify that the `<title>` element contains the string `Console // Werkzeug Debugger`, confirming the exposure of the interactive console.

Recommendation

Restrict access to /console via web server ACLs or move it outside the web root. Audit the file for sensitive content and rotate any exposed credentials immediately.

C-10: Race Condition: Overdraft — /transfer

Critical

Identified on: October 19, 2026

Affected Components: /transfer

Google Security Product: TODO: None

Description

A race condition vulnerability was identified in the `/transfer` endpoint. The application fails to properly serialize financial transactions, allowing multiple concurrent transfer requests to be processed against an account's balance simultaneously. The check for sufficient funds is performed for each request before the balance is updated, leading to a state where multiple deductions are approved based on an outdated balance. This results in the account being overdrawn, creating a negative balance and allowing a user to spend more funds than are available, which represents a direct financial loss.

How to Exploit

1. Authenticate to the application and note the starting balance of an account (e.g., $100.00).
2. Prepare a POST request to the `/transfer` endpoint to send an amount nearly equal to the total balance (e.g., $99.99).
3. Using a multi-threaded request utility, configure it to send at least ten identical copies of this POST request concurrently.
4. Execute the concurrent requests against the `/transfer` endpoint.
5. Observe the responses; multiple requests should receive an HTTP 200 OK status, indicating successful transfers.
6. After the requests have completed, check the account's final balance.
7. A negative balance confirms the vulnerability, as the total debited amount exceeded the initial funds available.

Recommendation

To prevent overdrafts, financial transactions must be atomic. Use a database-level lock, such as `SELECT ... FOR UPDATE`, to lock the account row during the transaction. This ensures the balance check and debit operation are performed as a single, indivisible unit, blocking any other concurrent requests on the same account until the transaction is complete.

H-1: Information disclosure: debug endpoint exposes account/user data

High

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

An unauthenticated debug endpoint was discovered at the path `/debug/users`. This endpoint improperly exposes the complete user account table to any client that requests it. The leaked data includes highly sensitive fields such as usernames, passwords in plain text, account numbers, and administrative privilege flags. This presents a critical security risk, as an attacker can leverage this information to gain unauthorized access to all user and administrator accounts, leading to data theft, financial fraud, and complete system compromise.

How to Exploit

1. Use a command-line tool or web browser capable of making HTTP requests.
2. Ensure you are not authenticated to the target application.
3. Construct a GET request to the application's base URL.
4. Set the request path to `/debug/users`.
5. Send the HTTP request.
6. Observe the HTTP response from the server.
7. Verify the server returns an HTTP status code of `200 OK`.
8. Confirm the response body contains a JSON structure listing user accounts and their associated sensitive data, including `password`, `account_number`, and `is_admin` fields.

Recommendation

Enforce authentication and authorisation on all endpoints. Apply deny-by-default access control with server-side role checks.

H-2: Session JWT signed with a weak/default secret (HS256)

High

Identified on: October 19, 2026

Affected Components: /login

Google Security Product: TODO: None

Description

The application utilizes JSON Web Tokens (JWTs) for session management, signed with the HS256 symmetric algorithm. During testing, it was discovered that the tokens are signed with a weak and commonly used default secret, 'secret123'. This allows an attacker to forge valid JWTs for any user account by creating a token payload with a chosen user's identity and signing it with the known secret. Successful exploitation leads to a complete authentication bypass, enabling an attacker to impersonate any user, including administrators, and gain full access to the application's data and functionality.

How to Exploit

1. Using valid credentials, authenticate to the application via the `/login` endpoint to obtain a session JWT. 2. Capture the issued JWT from the server's response. The token will be a string comprised of three Base64-encoded parts separated by periods. 3. Decode the header and payload sections of the token to inspect their contents. Note the algorithm specified in the header is 'HS256'. 4. Modify the payload to impersonate another user, for example by changing a 'sub' or 'username' claim to 'admin'. 5. Create a new HMAC-SHA256 signature for the original header and the modified payload using the secret key 'secret123'. 6. Construct a new, forged JWT by combining the Base64-encoded original header, the Base64-encoded modified payload, and the Base64-encoded new signature, each separated by a period. 7. Submit a request to a protected endpoint using the forged JWT in the `Authorization: Bearer` header. 8. Observe that the application grants access, confirming successful impersonation of the user specified in the forged token.

Recommendation

Enforce authentication and authorisation server-side on all sensitive endpoints. Apply rate limiting and use strong session management practices.

H-3: Unauthenticated BOLA on /check_balance/<id>

High

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

The application's `/check_balance/<id>` endpoint does not require authentication to access its functionality. An unauthenticated attacker can directly query this endpoint by supplying an account identifier in the URL path. Successful requests return sensitive user information, including the account number, current balance, and associated username. This information disclosure constitutes a Broken Object Level Authorization (BOLA) vulnerability, as the system fails to verify that the requester is authorized to access the requested data object. This allows for the enumeration of user accounts and the unauthorized retrieval of their financial details, violating data confidentiality.

How to Exploit

1. Identify a target account identifier, such as `8824356273`.
2. Using a web client or command-line tool, prepare an HTTP GET request for the target application.
3. Set the request path to `/check_balance/8824356273`.
4. Ensure no authentication tokens or session cookies are included in the request headers.
5. Send the request to the server.
6. Observe the server's response, expecting an HTTP `200 OK` status code.
7. Verify that the response body contains a JSON object with the sensitive details for the specified account, such as `{"account_number": "8824356273", "balance": 1000.0, ...}`.

Recommendation

Enforce ownership checks before returning any resource. Query with both resource ID and user ID. Apply deny-by-default access control at the data layer.

H-4: Unauthenticated BOLA on /transactions/<id>

High

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

The application endpoint at /transactions/ does not enforce authentication or authorization checks. This allows an unauthenticated attacker to directly access transaction records by specifying a valid identifier in the URL. By iterating through possible numeric identifiers, an attacker can systematically exfiltrate transaction data for all users of the application. This vulnerability leads to a large-scale breach of sensitive financial information, violating data confidentiality.

How to Exploit

1. Identify a valid transaction or account identifier, such as '8824356273'.
2. Use a command-line terminal or an API client to construct an HTTP request.
3. Prepare a GET request targeting the path /transactions/, replacing with the identifier from step 1.
4. Ensure that no 'Authorization' header, session cookie, or other authentication token is included in the request.
5. Send the request to the target server, for example: GET /transactions/8824356273 HTTP/1.1.
6. Observe the server's response, which will be an HTTP 200 OK status.
7. Inspect the response body. It will contain a JSON object with sensitive transaction details for the requested account, confirming the absence of access control.

Recommendation

Enforce ownership checks before returning any resource. Query with both resource ID and user ID. Apply deny-by-default access control at the data layer.

H-5: SQL Injection (Error-Based) — /transactions/{accountNumber} (accountNumber)

High

Identified on: October 19, 2026

Affected Components: /transactions/{accountNumber}

Google Security Product: TODO: None

Description

The `/transactions/{accountNumber}` endpoint is vulnerable to error-based SQL injection. Supplying a single quote in the `accountNumber` path parameter causes the application to generate a database query with invalid syntax. The backend PostgreSQL database responds with a detailed error message, which is then exposed to the client in the API response body. This behavior confirms that user input is not being properly sanitized before being included in a database query, allowing an attacker to manipulate the query's logic. This could lead to unauthorized access to sensitive data, data modification, or other database-level attacks.

How to Exploit

1. Using a web request tool, construct a GET request to the application.
2. Set the request URL to the vulnerable endpoint, appending a single quote to a valid account number: `the application/transactions/1'`.
3. Send the request to the server.
4. Observe the server's response.
5. Confirm the HTTP status code is `500 Internal Server Error`.
6. Examine the response body and verify it contains a database error message, such as `syntax error at or near "1"`, which confirms the injection.

Recommendation

The query for retrieving transactions is vulnerable to SQL injection. Use parameterized queries to handle the `accountNumber` input. For example, in Python with psycopg2, use `cursor.execute("SELECT ... FROM transactions WHERE from_account = %s OR to_account = %s", (account_number, account_number))` instead of embedding the value directly in the query string.

H-6: Mass Assignment — /register

High

Identified on: October 19, 2026

Affected Components: /register

Google Security Product: TODO: None

Description

The user registration endpoint at `/register` is vulnerable to mass assignment. The application's backend blindly binds all user-provided parameters from the POST request to the internal user object without a proper whitelist of permitted fields. An unauthenticated attacker can exploit this by injecting an `is_admin` field with a value of `True` into the JSON body of the registration request. Successful exploitation results in the creation of a new user account with administrative privileges, leading to a full compromise of the application's user management functions.

How to Exploit

1. Prepare to send a request to the user registration endpoint, `/register`.
2. Construct a `POST` request with a JSON body containing standard registration fields, such as `username` and `password`.
3. Add the parameter `"is_admin": true` to the JSON object in the request body.
4. An example payload would be: `{"username": "testuser", "password": "complexpassword123", "is_admin": true}`.
5. Send the crafted `POST` request to the `/register` endpoint.
6. Observe the server's response, which should be an `HTTP 200 OK`.
7. Inspect the body of the response and confirm that it reflects the new user's administrative status, indicated by the presence of the key-value pair `"is_admin": true`. This verifies the successful creation of a privileged account.

Recommendation

Bind registration to an explicit allow-list of fields; never let clients set privilege fields such as is_admin.

H-7: Brute-forceable 3-digit password-reset PIN (PIN leaked in response)

High

Identified on: October 19, 2026

Affected Components: /forgot-password

Google Security Product: TODO: None

Description

The password reset mechanism at the /forgot-password endpoint is critically flawed. Upon initiating a password reset for a user, the system generates a three-digit numeric PIN required to complete the process. This PIN is not sent via an out-of-band channel (like email) but is instead included directly in the body of the HTTP response to the initial request. Because the PIN is only three digits, it is already trivially brute-forceable, but its direct inclusion in the response completely negates its purpose as a secret. This allows an attacker to immediately obtain the reset token for any user and subsequently change their password, leading to complete account takeover.

How to Exploit

1. Using a web proxy, monitor network traffic from your browser.
2. Navigate to the application's password reset page.
3. Enter a known username, for example 'ma_probe_2a03d80c', and submit the request to initiate a password reset.
4. Intercept the outbound request, which will be a POST request to the /forgot-password endpoint containing the username.
5. Allow the request to be sent to the server and examine the corresponding HTTP 200 OK response.
6. Observe that the body of the response contains the plaintext password reset PIN, for example, a field such as 'pin=566'.
7. This confirms that the secret PIN is leaked to the client, allowing an attacker to proceed with changing the user's password.

Recommendation

Use cryptographically random, time-limited, single-use reset tokens. Return identical responses for registered and unregistered addresses. Apply rate limiting to the reset endpoint.

H-8: Unrestricted file upload (dangerous extension accepted)

High

Identified on: October 19, 2026

Affected Components: /upload_profile_picture

Google Security Product: TODO: None

Description

The file upload functionality at the `/upload_profile_picture` endpoint fails to properly validate file extensions and content types. During testing, the application accepted a file with a `.php` extension and an `application/x-php` content type, saving it to the web root. This allows an attacker to upload a malicious script, such as a web shell, to a publicly accessible location on the server. Successful exploitation can lead to remote code execution, giving an attacker control over the web application, its data, and the underlying server.

How to Exploit

1. Create a file named `floor_probe.php` containing executable PHP code, such as `<?php echo 'vulnerable'; ?>`.
2. Navigate to the profile picture upload feature in the web application.
3. Intercept the file upload request, which is a POST request to `/upload_profile_picture`.
4. Replace the legitimate image file with the `floor_probe.php` file created in step 1.
5. Forward the modified request to the server.
6. Observe the server's response. A successful response will be an HTTP 200 OK with a JSON body containing a `file_path` similar to `"static/uploads/839339_floor_probe.php"`.
7. Access the URL provided in the `file_path` to confirm that the server executes the uploaded PHP script.

Recommendation

Validate uploads by content, enforce an extension/MIME allow-list, store outside the web root, and serve with a non-executable Content-Type / disposition.

H-9: BOLA (idor) — /api/transactions?account_number={id}

High

Identified on: October 19, 2026

Affected Components: /api/transactions?account_number={id}

Google Security Product: TODO: None

Description

The application is vulnerable to Broken Object Level Authorization (BOLA) at the /api/transactions endpoint. The system fails to validate that the authenticated user is authorized to access the data requested via the `account_number` parameter. By modifying the value of this parameter, an authenticated user can bypass authorization controls and directly access transaction data belonging to any other user on the platform. Testing confirmed that a single authenticated session could retrieve data for multiple distinct foreign identifiers, demonstrating a critical failure in access control enforcement. This flaw allows for unauthorized enumeration and exfiltration of sensitive financial information, representing a severe breach of data confidentiality.

How to Exploit

1. Authenticate to the application as a standard user.
2. Using a web proxy, capture the GET request made to retrieve your own account transactions. The request URI will be /api/transactions?account_number={your_account_id}.
3. Note your legitimate account ID from the captured request.
4. Send the captured request to a repeater tool.
5. Modify the value of the `account_number` parameter to an identifier that does not belong to your account (e.g., 1, 2, 1001).
6. Send the modified request.
7. Observe the server's response. A successful 200 OK response containing the transaction details for the targeted foreign account confirms the vulnerability.

Recommendation

Enforce server-side ownership checks for every resource access. Use non-sequential UUIDs as identifiers and apply authentication middleware globally.

H-10: BOLA (idor) — /transactions/{accountNumber}

High

Identified on: October 19, 2026

Affected Components: /transactions/{accountNumber}

Google Security Product: TODO: None

Description

The application lacks object-level authorization controls on the /transactions/{accountNumber} endpoint. This allows an authenticated user to access transaction data associated with other users by manipulating the 'accountNumber' path parameter. During testing, a single user session was able to retrieve sensitive transaction histories for multiple foreign account identifiers, including sequentially guessed numbers and predictable administrative IDs. This direct object reference vulnerability leads to a severe breach of data confidentiality, as any authenticated user can systematically enumerate and exfiltrate the financial data of other accounts.

How to Exploit

1. Authenticate to the application as a standard user (User A).
2. Using the authenticated session, make a legitimate request to view transactions for your own account. For example, send a GET request to /transactions/{UserA_AccountNumber}.
3. Observe the successful response containing User A's transaction data.
4. Identify or guess a different, valid account number belonging to another user (User B). Test cases from the evidence include '1', '2', and '1001'.
5. Using the same authenticated session for User A, send a new GET request to /transactions/{UserB_AccountNumber}, replacing the original account number with User B's.
6. Observe that the server returns a successful (e.g., HTTP 200 OK) response.
7. Confirm that the response body contains sensitive transaction details belonging to User B, verifying that the server failed to perform an authorization check.

Recommendation

Enforce server-side ownership checks for every resource access. Use non-sequential UUIDs as identifiers and apply authentication middleware globally.

H-11: BOLA — Cross-User Resource Access — /transactions/{username}

High

Identified on: October 19, 2026

Affected Components: /transactions/{username}

Google Security Product: TODO: None

Description

The web application is vulnerable to a Broken Object Level Authorization (BOLA) flaw at the `/transactions/{username}` endpoint. The application fails to validate that the authenticated user making the request is authorized to access the transaction data associated with the username provided in the URL path. As a result, any authenticated user can directly access the transaction history of any other user by manipulating the `{username}` parameter in the request URL. This unauthorized access to sensitive financial information constitutes a severe breach of data confidentiality and user privacy.

How to Exploit

1. Create two distinct user accounts. For this example, we will use 'user_A' and 'user_B'.
2. Log in to the application as 'user_A' and record the session token or cookie.
3. Using the session credentials for 'user_A', construct a GET request to the transaction endpoint for 'user_B'.
4. The request URL will be `the application/transactions/user_B`.
5. Send the request.
6. Observe the server's response. The server will incorrectly return an HTTP 200 OK status code along with the transaction data for 'user_B'.
7. This confirms the vulnerability, as the expected behavior would be an authorization denial, such as an HTTP 403 Forbidden response.

Recommendation

Enforce server-side ownership checks before returning any resource: query with both the resource ID and the authenticated user's ID (e.g. `Resource.query.filter_by(id=id, owner_id=current_user.id)`). Use non-sequential UUIDs as resource identifiers to eliminate enumeration.

H-12: DOM-based Cross-Site Scripting in Client-Side Rendering

High

Identified on: October 19, 2026

Affected Components: /static/dashboard.js

Google Security Product: TODO: None

Description

The application is vulnerable to DOM-based Cross-Site Scripting (XSS) in the client-side code responsible for rendering the dashboard. User-supplied data associated with payments and transactions, specifically in the `biller_name`, `category_name`, and `description` fields, is inserted directly into the page's HTML structure without proper output encoding. The vulnerable JavaScript code at `/static/dashboard.js` uses string interpolation to build the HTML, allowing malicious script payloads stored in these fields to be included in the Document Object Model (DOM). This allows an attacker to execute arbitrary JavaScript in the context of a victim's authenticated session, leading to potential session hijacking, data theft, or performing unauthorized actions on behalf of the user.

How to Exploit

1. Log into the application as a standard user.
2. Navigate to a function that allows for the creation of data with a `biller_name` field, such as adding a new bill payment.
3. In the field for 'Biller Name', enter the payload: `<img src=x onerror=alert('XSS-Biller')>`.
4. Save the new entry.
5. Navigate to the dashboard or transaction history page where this bill payment is displayed.
6. Observe that the browser executes the injected JavaScript, causing an alert dialog with the message 'XSS-Biller' to appear.
7. This confirms that the `biller_name` value is rendered without sanitization. The same procedure can be repeated for the `category_name` and `description` fields.

Recommendation

Encode all user-controlled values before inserting them into the DOM, use textContent instead of innerHTML, and apply a strict Content-Security-Policy.

M-1: Missing Security Headers

Medium

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

The web application does not set several recommended HTTP security headers on its responses. Specifically, the 'Content-Security-Policy', 'X-Frame-Options', 'X-Content-Type-Options', and 'Referrer-Policy' headers were found to be missing. The absence of these headers weakens the application's defense-in-depth posture, exposing users to client-side attacks such as Cross-Site Scripting (XSS), clickjacking, and MIME-type sniffing. This could lead to unauthorized actions, session hijacking, or the leakage of sensitive information through browser behavior.

How to Exploit

1. Open a web browser and launch the built-in developer tools.
2. Select the 'Network' tab within the developer tools.
3. Navigate to the main URL of the target application.
4. In the Network tab, select the first request made to the application server, which is typically the HTML document.
5. Examine the 'Response Headers' section in the details pane for that request.
6. Confirm the absence of the 'Content-Security-Policy', 'X-Frame-Options', 'X-Content-Type-Options', and 'Referrer-Policy' headers in the list of returned headers.

Recommendation

Define a Content-Security-Policy header restricting script, style and media sources to trusted origins (e.g. "default-src 'self'; script-src 'self'"). Add 'X-Frame-Options: SAMEORIGIN' (or use CSP frame-ancestors) to prevent the page from being embedded in a cross-origin frame (clickjacking). Add 'X-Content-Type-Options: nosniff' to prevent MIME-type sniffing. Add 'Referrer-Policy: strict-origin-when-cross-origin' to prevent URL parameters from leaking to third-party origins via the Referer header.

M-2: JWT Has No Expiry — Permanent Token

Medium

Identified on: October 19, 2026

Affected Components: /login

Google Security Product: TODO: None

Description

The application issues JSON Web Tokens (JWTs) for session management that lack an expiration claim ('exp'). This was observed in the tokens generated upon successful authentication at the /login endpoint. The absence of an expiration means the token remains valid indefinitely. Consequently, if a token is compromised through leakage or theft, an attacker can use it to maintain persistent access to the victim's account until the user manually logs out or the token is explicitly revoked on the server side, significantly increasing the risk of unauthorized access.

How to Exploit

1. Using an intercepting proxy, monitor the network traffic from your browser.
2. Navigate to the application's login page and enter valid user credentials.
3. Submit the login request to the /login endpoint and capture the server's HTTP response.
4. Locate the JWT returned by the server, commonly found in the 'Authorization' header or the JSON response body.
5. Copy the full JWT string.
6. Decode the JWT payload, which is the second part of the token string (between the first and second dots), using a Base64 decoder.
7. Inspect the decoded JSON payload.
8. Verify that the payload contains claims like 'iat' (issued at) but is missing the 'exp' (expiration time) claim, confirming the vulnerability.

Recommendation

Always set `exp` claim on JWTs. Use short-lived access tokens (≤15 min) with refresh-token rotation to limit the blast radius of token theft.

M-3: No rate limiting / account lockout on login endpoint

Medium

Identified on: October 19, 2026

Affected Components: /login

Google Security Product: TODO: None

Description

The login functionality at the `/login` endpoint does not enforce rate limiting or implement an account lockout mechanism in response to repeated failed authentication attempts. During testing, it was possible to submit eight consecutive invalid login requests without triggering any form of throttling, such as an HTTP 429 response, or an account suspension. This absence of protective controls allows an attacker to perform an unlimited number of login attempts against any user account. Consequently, the application is vulnerable to automated attacks like credential stuffing and password brute-forcing, which can lead to widespread unauthorized account compromise.

How to Exploit

1. Using a web interception proxy, capture a login request.
2. Isolate the POST request sent to the `/login` endpoint. The request body should contain parameters for a username and password.
3. Modify the request to use a known or guessed username and an incorrect password.
4. Send the request and observe the HTTP 401 Unauthorized response, confirming an authentication failure.
5. Re-send the same request from step 4 at least ten times in quick succession.
6. Examine the HTTP status code for each of the ten responses.
7. Confirm that all responses are HTTP 401 Unauthorized and that no HTTP 429 Too Many Requests response is received, which demonstrates the lack of rate limiting.

Recommendation

Implement rate limiting on authentication endpoints (login, password reset). Use exponential back-off and temporary account lock-out after N failed attempts (e.g. 5 attempts → 15 min lockout).

M-4: Hidden admin endpoint reachable via predictable path / sup3r_s3cr3t_admin

Medium

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

An administrative endpoint was discovered at the predictable path /sup3r_s3cr3t_admin. While access to the endpoint's functionality is restricted, as indicated by the HTTP 401/403 response, its existence is confirmed to any unauthenticated user who can discover the path. This exposure of a sensitive interface increases the application's attack surface. Attackers can now target this known administrative entry point for further attacks, such as credential brute-forcing or exploitation of other potential vulnerabilities. Relying on an obscure path name for security is an insufficient control and does not replace the need for robust network-level or application-level access restrictions.

How to Exploit

1. Use a command-line tool capable of making HTTP requests, such as curl.
2. Send a request to a non-existent path to establish a baseline for a "Not Found" response. For example: `curl -sk -o /dev/null -w '%{http_code}' [TARGET_URL]/__ac_canary_zz9p__`
3. Observe that the server returns an HTTP 404 status code.
4. Now, send a request to the discovered administrative endpoint: `curl -sk -o /dev/null -w '%{http_code}' [TARGET_URL]/sup3r_s3cr3t_admin`
5. Observe that the server returns an HTTP 401 or 403 status code.
6. The difference in response codes between the non-existent path (404) and the target path (401/403) confirms the existence of the hidden administrative endpoint.

Recommendation

Apply deny-by-default access control. Enforce authentication and role-based authorisation on every endpoint server-side. Never rely on client-side role flags or hidden routes.

L-1: Framework / version disclosure

Low

Identified on: October 19, 2026

Affected Components: The target application

Google Security Product: TODO: None

Description

The application's HTTP responses include a `Server` header that discloses the specific software components and versions in use. During testing, the banner `Werkzeug/2.0.1 Python/3.9.25` was observed in server responses. This information leakage provides an attacker with precise details about the application's technology stack without requiring active probing. An adversary can use these specific version numbers to search for publicly documented vulnerabilities (CVEs) and corresponding exploits, significantly reducing the time and effort needed to mount a targeted attack.

How to Exploit

1. Use a command-line tool or web proxy capable of inspecting HTTP headers. 2. Send an HTTP GET request to the application's main page or any valid endpoint. 3. Capture the server's response. 4. Inspect the headers within the captured response. 5. Locate the `Server` header. 6. Verify that its value contains the string `Werkzeug/2.0.1 Python/3.9.25`.

Recommendation

Harden your web server configuration: disable unnecessary features, set security headers, restrict HTTP methods, and keep dependencies updated.





A Appendix