Aldercrest

Web Application Penetration Test Report

hound

TARGET

aldercrest.targets.hound-agents.com (Aldercrest online banking web application and JSON API, hosted at 54.225.12.30 and 50.19.141.238, port 443)

REPORT DATE

2026-08-27

Table of Contents

Executive Report ---- 4

Summary ---- 4

Business Impact ---- 4

Vulnerabilities ---- 5

Technical Report ---- 10

Vulnerabilities ---- 10

VULN-01 - Password Reset Flow Allows Complete Takeover of Any Account ---- 10

VULN-02 - Session Token Signatures Are Never Verified, Allowing Anyone to Act as Any Customer or Administrator ---- 13

VULN-03 - Unauthenticated SQL Injection in the Balance Lookup Gives Full Database Control as a Superuser ---- 15

VULN-04 - Unauthenticated SQL Injection in Transaction History Returns Every Customer's Ledger ---- 19

VULN-05 - SQL Injection in the Sign-In Form Allows Sign-In as Any Customer Without a Password ---- 21

VULN-06 - Customer Passwords Are Stored as Readable Text ---- 23

VULN-07 - Account Balances and Transaction Histories Are Served Without Any Authentication ---- 25

VULN-08 - Loan Approval Creates Money at an Attacker-Chosen Amount and Can Be Replayed ---- 27

VULN-09 - Card Limit Endpoint Builds Database Commands From Client-Supplied Field Names ---- 30

VULN-10 - Stored Cross-Site Scripting in Transfer Descriptions Steals Customer Sessions ---- 33

VULN-11 - Session Tokens Never Expire and Cannot Be Revoked ---- 36

VULN-12 - Negative Transfer Amounts Create Money and Push Other Accounts Below Zero ---- 38

VULN-13 - Any Signed-In Customer Can Read Any Other Customer's Transaction History ---- 41

VULN-14 - Full Card Numbers, Security Codes and Expiry Dates Are Returned and Stored Unprotected ---- 43

VULN-15 - Error Messages Reveal Database Structure, Full Queries and Internal Program Detail ---- 45

VULN-16 - AI Support Endpoints Disclose System Configuration and the Database Layout Without Authentication ---- 48

VULN-17 - Simultaneous Transfers Let a Customer Spend the Same Money Several Times ---- 50

VULN-18 - Any File Type Can Be Uploaded and Is Served From the Bank's Own Domain ---- 53

VULN-19 - Other Websites Can Change State Inside a Customer's Signed-In Session ---- 56

VULN-20 - Any Signed-In Customer Can Change Another Customer's Payment Card ---- 58

VULN-21 - Bill Payments Ignore Published Rules, Never Complete, and Leave No Statement Record ---- 61

VULN-22 - Loan Applications Accept Any Amount, Including Negative and Near-Billion-Dollar Values ---- 63

VULN-23 - Banking Pages Can Be Hidden Inside Another Website to Capture Clicks ---- 66

Hardening Report ---- 69

Defense-in-depth ---- 69

HARDEN-01 - No Security Response Headers Are Sent on Any Route ---- 69

HARDEN-02 - Session Cookie Is Issued Without Secure and With SameSite=None ---- 70

HARDEN-03 - The Server Reflects Any Requesting Origin in Its Cross-Origin Headers ---- 71

HARDEN-04 - The Sign-In Endpoint Has No Lockout, Backoff or Throttle ---- 72

HARDEN-05 - Request Throttling Is Keyed on Values the Caller Chooses ---- 73

HARDEN-06 - Uploads Have No Size Limit or Storage Quota ---- 75

HARDEN-07 - The Interactive Debugger Is Reachable From the Internet ---- 76

HARDEN-08 - The Host Header Is Not Validated and Redirects Downgrade to Plain HTTP ---- 77

Supply Chain Hygiene ---- 78

HARDEN-09 - The Werkzeug Web Library Is Several Major Versions Out of Date ---- 78

HARDEN-10 - The Python Runtime Is Past End of Life ---- 79

Best Practice ---- 80

HARDEN-11 - A Development Web Server Is Serving Production Banking Traffic ---- 80

Coverage Report ---- 82

Attack Surface Discovered ---- 82

Testing Coverage Matrix ---- 83

Exploitation Attempts ---- 85

Appendix: Methodology, Scope, and Limitations ---- 92

Assessment Limitations ---- 92

Executive Report

Summary

The assessment identified 23 verified vulnerabilities across Aldercrest's internet-facing web application and API: 3 Critical, 11 High and 9 Medium. Three Critical findings establish attack paths that require no valid account and allow an external attacker to take over customer accounts, forge sessions as any customer or administrator, and execute database commands with superuser privileges. In its current state, Aldercrest cannot reliably protect customer identities, financial records or transaction integrity.

The combined attack paths permit unauthorized access to customer records, transaction histories, passwords and complete payment-card credentials. Testing also demonstrated account impersonation, session theft, unauthorized changes to other customers' cards, balance manipulation, repeated loan payouts and concurrent spending of the same funds. Storing passwords and complete card credentials in readable form materially increases the potential impact of a compromise.

The findings reflect systemic control failures across authentication, authorization, input handling, session management, transaction processing and sensitive-data protection. Several independent paths reach the same customer and financial consequences, so isolated fixes will not close the overall exposure. The affected controls should be remediated together and the critical workflows retested before production use.

Business Impact

Financial integrity and fraud. Attackers can manipulate balances, approve repeated loan payouts, spend the same funds more than once and move money after stealing a customer's session. These paths create direct and repeatable loss scenarios and undermine confidence in the ledger.

Customer and data exposure. An external attacker can access customer identities, account details, balances, transaction histories, readable passwords and complete payment-card credentials. Database-superuser access extends the exposure beyond individual application records.

Regulatory and contractual exposure. If production data were affected by comparable exploitation, the demonstrated access and storage practices would require assessment under applicable breach-notification, privacy and payment-card obligations.

Customer trust and fraud enablement. The bank's own web address can serve attacker-controlled pages, and malicious sites can cause actions within a customer's signed-in session. These conditions make credible phishing and customer-directed fraud easier to conduct.

Vulnerabilities

VULN-01: The password reset feature lets anyone take over any account.

A stranger who knows only a username can get a new password set on that account. Asking to reset a password hands back the secret reset code in the reply, and even when the code is hidden there is nothing to stop guessing all 1,000 possible codes in under eight seconds. This means any customer account, including the administrator account, can be stolen outright.

> RECOMMENDED ACTION: Never return the reset code in an API reply. Use a long random code sent to the account owner, make it expire, and lock the reset endpoint after a few wrong tries.

VULN-02: The server never checks that a login token is genuine.

Anyone can write their own token that says "I am the administrator" and the server accepts it. We reached the admin panel and read every customer's name, account number, balance and admin flag. We also acted as other customers. Changing the secret key would not fix this.

> RECOMMENDED ACTION: Turn on signature checking for every token on every request, then rotate the signing key.

VULN-03: The balance lookup page gives full control of the banking database to strangers.

One web request with no login reads any table, changes any record, and reads files from the database server. The database account used by the application has the highest possible database privileges.

> RECOMMENDED ACTION: Use parameterised database queries on this endpoint and give the application a limited database account instead of a superuser one.

VULN-04: The transaction history page returns every customer's transactions to strangers.

One crafted request with no login returns the entire bank ledger, including who paid whom, amounts, dates and descriptions.

> RECOMMENDED ACTION: Use parameterised queries, require sign-in, and return only records that belong to the caller.

VULN-05: The sign-in form can be tricked into logging an attacker in as any customer.

Typing a small piece of database text into the username box removes the password check. We received a real, valid session for a customer account without knowing the password.

> RECOMMENDED ACTION: Use parameterised queries for the credential lookup and never build the query from raw form input.

VULN-06: Customer passwords are stored as readable text.

The stored value is the password itself. Because the database is already readable by strangers through **VULN-03**, every customer password can be collected with no cracking work at all.

> RECOMMENDED ACTION: Store passwords using a modern password hash with a per-user salt, and require all customers to set a new password once this is done.

VULN-07: Account balances and transaction histories are open to the public.

Two pages need no sign-in at all, and account numbers run in a simple sequence, so anyone can walk through them and read the whole customer base.

> RECOMMENDED ACTION: Require sign-in on both pages and return only the caller's own account.

VULN-08: Loan approval creates money out of nothing and can be repeated.

Using a self-written admin token, an attacker approves a loan and the chosen amount is added to an account. Approving the same loan again pays out again, every time.

> RECOMMENDED ACTION: Verify token signatures, block approval of loans that are already approved, and stop one person approving their own loan.

VULN-09: The card limit feature builds database commands from field names the caller sends.

This lets any signed-in caller run their own database commands and rewrite any payment card record, including its spendable balance, its card number and who owns it.

> RECOMMENDED ACTION: Never use client-supplied field names as database column names. Accept only the one documented field and check card ownership.

VULN-10: A message sent with a $1 transfer can steal the receiver's session.

The description text is shown back to the receiver without being made safe, so attacker code runs in their browser and sends their session token away. We used a stolen token to read card data and move money.

> RECOMMENDED ACTION: Escape all user text before showing it and add a content security policy.

VULN-11: Session tokens never expire and cannot be cancelled.

A stolen token works forever. Changing the password does not stop it, and there is no sign-out on the server.

> RECOMMENDED ACTION: Add an expiry to every token and add a way to cancel sessions, including automatically on password change.

VULN-12: Sending a negative amount lets a customer raise their own balance.

The check compares the size of the amount but the update uses the sign, so money flows backwards. There is no floor on the receiving account, which can be pushed below zero.

> RECOMMENDED ACTION: Reject amounts that are zero or negative, and check that both accounts stay valid after the change.

VULN-13: Any signed-in customer can read any other customer's transaction history.

Changing an account number in the request is enough. The server never checks who owns the account.

> RECOMMENDED ACTION: Take the account from the signed-in session, not from the request, and check ownership on every read.

VULN-14: Full card numbers, security codes and expiry dates are handed out and stored in plain form.

This is everything needed for online card fraud, and it is returned on every request rather than once at issue. Card records belonging to other people, and records that do not exist, are also reachable.

> RECOMMENDED ACTION: Show only the last four digits, never keep the security code after the card is created, and check card ownership on every request.

VULN-15: Error messages reveal how the database is built.

Failed requests return the full database command, table names, column names and internal program details. Two of these pages need no sign-in.

> RECOMMENDED ACTION: Return a short generic error with a reference number and keep the detail in server logs only.

VULN-16: The AI support endpoints reveal system settings and the database layout to anyone.

One request with no sign-in returns the assistant's instructions, which contain an accurate map of the database, including the columns holding passwords, balances and admin flags. Simple malformed requests return the same information plus internal error text.

> RECOMMENDED ACTION: Require sign-in on this endpoint, remove the database layout from the assistant's instructions, and return generic errors.

VULN-17: Sending several transfers at the same moment lets a customer spend the same money twice.

In just over half our attempts, several payments went through against a balance that could only cover one, pushing the account thousands of dollars below zero. The reply always showed the wrong balance, hiding the problem.

> RECOMMENDED ACTION: Make the balance check and the balance update a single locked database operation, and report the true balance in the reply.

VULN-18: Any file type can be uploaded and is then served from the bank's own website.

We uploaded web pages that ran in the bank's own web address and read a signed-in customer's session token. Uploaded program files were not run, so this does not give control of the server.

> RECOMMENDED ACTION: Allow only real image types, check the file contents, and serve uploads from a separate domain as downloads.

VULN-19: Other websites can make changes inside a customer's signed-in session.

A page a customer visits while signed in can quietly change their profile picture or switch off their payment card. One version of this attack needs no scripting at all, just a single click.

> RECOMMENDED ACTION: Add an anti-forgery token to every state-changing request, set the session cookie to same-site, and check the origin of incoming requests.

VULN-20: Any signed-in customer can change another customer's payment card.

We froze someone else's card and changed their spending limit by supplying a different card number in the request.

> RECOMMENDED ACTION: Check that the card belongs to the caller before any read or change.

VULN-21: Bill payments break the published rules and never actually complete.

A payment below the biller's published minimum was accepted and stored. Every bill payment in the system is stuck as "pending" forever, money leaves the account straight away, and no record appears in the customer's statement.

> RECOMMENDED ACTION: Enforce the published minimum on the server, add a step that completes or cancels a payment, and write a matching statement entry for every debit.

VULN-22: Loan applications accept any amount at all.

Negative, zero and near billion-dollar amounts were accepted and saved exactly as sent. This is the step that lets an attacker pick the amount created by VULN-08.

> RECOMMENDED ACTION: Set a minimum and maximum loan amount and reject anything outside it.

VULN-23: The banking pages can be hidden inside another website to trick clicks.

One ordinary click on a fake "claim prize" button switched off a customer's payment card. The attacker cannot read the hidden page, but they can make the customer act on it.

> RECOMMENDED ACTION: Send the headers that stop other sites framing your pages, and add a confirmation step to card controls.

Technical Report

Vulnerabilities

The 23 confirmed vulnerabilities below are ordered by CVSS v4.0 base score, highest first.

VULN-01

Password Reset Flow Allows Complete Takeover of Any Account

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/api/v1/forgot-password, /api/v1/reset-password, /api/v2/forgot-password, /api/v2/reset-password

CVSS V4.0 BASE SCORE:

9.2 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

A single unauthenticated request containing only a username returns the live password-reset code, letting anyone on the internet set a new password and take over any customer or administrator account outright. Where the code is not returned, an unauthenticated attacker exhausts the entire three-digit reset code space in under eight seconds and sets a password of their choosing with the same result.

Description

The password reset workflow has two independent failures that lead to the same outcome.

First, `POST /api/v1/forgot-password` returns the live reset PIN in the response body, in a field named `debug_info.pin`. The request is unauthenticated and is keyed only on a username. No proof of account ownership is required.

Second, the version 2 endpoints were introduced to hide the PIN, and they do hide it. But the PIN is only three decimal digits, which is 1,000 possible values, and `POST /api/v2/reset-password` has no rate limit, no lockout, no backoff and no CAPTCHA. The entire keyspace can therefore be walked blind. The version 2 change is not a fix, because the PIN never needed to be guessed slowly in the first place.

The PIN itself is validated correctly and is single use. Those are the only working controls in the flow, and disclosure or exhaustive guessing defeats both.

Evidence

Path 1: the reset code is published. An unauthenticated `POST /api/v1/forgot-password` carrying only a username returned HTTP 200 with the reset code in the response body:


```
{"status":"success",
 "message":"Reset PIN has been sent to your email.",
 "debug_info":{"pin":"299","pin_length":3,"timestamp":"...","username":"<test account>"}}
```


The returned value was checked against the `users.reset_pin` column read through the injection point in **VULN-03**. The stored value matched exactly, and it had changed from a previous value to `299` as a direct result of the request, which proves the published value is the genuine live token and not a decoy. A deliberately wrong PIN of `000` was submitted first as a control and was correctly rejected with HTTP 400 `Invalid reset PIN`, confirming the PIN is genuinely enforced. The real PIN then returned HTTP 200 with `reset_success: true`. A sign-in with the attacker-chosen password succeeded with HTTP 200 and returned a valid session token, while a sign-in with the original password began failing with HTTP 401 in the same sequence. After use, `reset_pin` was set to NULL.

Path 2: the hidden code is guessed. A fresh PIN was generated through `POST /api/v2/forgot-password`, which does not disclose it, so the value was never observed. All 1,000 candidate values `000` through `999` were then submitted to `POST /api/v2/reset-password`:

MEASUREMENT	RESULT
Total requests	1,000
Elapsed time	7.57 seconds (132.1 requests per second)
Responses	999 x HTTP 400, 1 x HTTP 200
HTTP 429 responses	0
Latency by quartile	0.008 / 0.007 / 0.007 / 0.008 seconds (flat, no backoff)
Rate-limit headers	None present

The single HTTP 200 was the correct PIN, and the password change took effect. This was confirmed by a successful sign-in with the new password and by reading the stored credential value.

Reproduction Steps

Prerequisite: a test account you control, and its username. Do not run this against a live customer account, because it changes that account's password.

1. Record the current password of the test account and confirm it works: `POST /login` with the correct username and password returns HTTP 200 with a token.
2. Send `POST /api/v1/forgot-password` with body `{"username":"<test account>"}` and no `Authorization` header.

   Expected result: HTTP 200, and the response body contains `debug_info.pin` with a three-digit value.

3. Send `POST /api/v1/reset-password` with `{"username":"<test account>","reset_pin":"000","new_password":"<value you will not use>"}`, using a PIN you know is wrong.

   Expected result: HTTP 400 `Invalid reset PIN`. This control proves the PIN is enforced.

4. Send `POST /api/v1/reset-password` again with the PIN from step 2 and a new password of your choosing.

   Expected result: HTTP 200 with `reset_success: true`.

5. Send `POST /login` with the new password.

   Expected result: HTTP 200 with a valid token. Sending the original password now returns HTTP 401.

6. For the second path, send `POST /api/v2/forgot-password` for the same account, which generates a new PIN without disclosing it. Then loop `000` through `999` against `POST /api/v2/reset-password` with a new password of your choosing.

   Expected result: 999 responses of HTTP 400, one HTTP 200, no HTTP 429 at any point, and the password change takes effect.

7. Cleanup: run the reset flow once more to restore the account's original password, then confirm with a successful sign-in and confirm the temporary password now returns HTTP 401.

Business Impact

Any account in the bank can be taken over from the internet by someone who knows only a username. Usernames are short, ordinary words and are also listed in full inside the admin panel reachable through **VULN-02**, so they are not a secret. The administrator account is included. The attacker gains everything the account holder has: balances, statements, card details, and the ability to move money. Because the legitimate password stops working, the customer is locked out of their own account and will notice only after the damage is done.

Remediation

1. Remove the `debug_info` object from all responses of the forgot-password and reset-password endpoints. A reset code must never appear in an API response.
2. Retire the version 1 endpoints entirely. Hiding the code in version 2 while leaving version 1 live means the vulnerable path is still reachable.
3. Replace the three-digit PIN with a cryptographically random token of at least 128 bits, delivered out of band to a channel the account owner already controls.
4. Give the token a short expiry (for example 15 minutes), keep it single use, and bind it to the account.
5. Add rate limiting and lockout on both the forgot-password and reset-password endpoints, keyed on a value the caller cannot choose. See **HARDEN-05** in the Hardening Report for why the current throttling key is not safe.
6. Invalidate all existing sessions for the account when a password is reset. See **VULN-11**, which currently makes this impossible without a schema change.
7. Verify the fix by confirming that the reset response contains no code, that 1,000 sequential guesses trigger lockout well before completion, and that a session issued before the reset no longer works after it.

VULN-02

Session Token Signatures Are Never Verified, Allowing Anyone to Act as Any Customer or Administrator

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/ (token issuance at POST /login; every Bearer-protected endpoint, verified against GET /sup3r_s3cr3t_admin)

CVSS V4.0 BASE SCORE:

9.2 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Remotely reachable over the internet with no credentials, no user interaction and no evasion required, and it was demonstrated to yield both complete read access to every customer's account data and the ability to change server-side state as any user or administrator.

Description

The application issues HS256 JSON Web Tokens containing the claims `{user_id, username, is_admin, iat}`. The server parses those claims and makes authorization decisions from them, but it never verifies the HMAC signature. The behaviour matches a token library decoding with signature verification switched off while still constraining the algorithm field.

The practical effect is that an attacker with no account and no credentials can write a token that asserts any `user_id` and `is_admin: true`, and the server will believe it.

An important detail for remediation: the signing secret in use is a weak dictionary-style value, and that is a genuine weakness in its own right. It is not, however, the defect that allows the bypass. Rotating the secret would remediate nothing, because no secret is needed.

Evidence

The same token payload was submitted to `GET /sup3r_s3cr3t_admin` under seven different signature conditions, together with two control cases. All responses were captured in a single request batch:

CASE	TOKEN	RESULT
A (control)	No token at all	HTTP 401 `Token is missing`
B (control)	A genuine, server-issued token with `is_admin: false`	HTTP 403 `Access Denied`
C	Self-written token, `is_admin: true`, signed with the correct secret	HTTP 200, 11,475-byte admin panel

CASE	TOKEN	RESULT
D	Identical payload signed with a deliberately wrong secret	HTTP 200, 11,475 bytes, body byte-identical to C
E	Signature replaced with the literal string `AAAA`	HTTP 200, 11,475 bytes
F	Signature left empty	HTTP 200, 11,475 bytes
G	Header changed to `alg: none`	HTTP 401 `Invalid token`
H	Header changed to `alg: HS512` , wrong secret	HTTP 401 `Invalid token`

Case B is the decisive control. A genuine non-admin token is correctly refused, which proves the admin check exists and fires. Case D is the second decisive control: a byte-identical response to case C proves the signature is not consulted at all. Only the algorithm field is constrained.

The behaviour is application-wide and not limited to the admin panel. A garbage-signature token asserting a different customer's `user_id` returned HTTP 200 on `GET /api/virtual-cards` (returning that customer's unmasked card number `7135********3430`), on `GET /api/transactions` , and on `GET /api/bill-payments/history` . The admin panel body itself disclosed the complete user table: six accounts with usernames, account numbers, balances (including an administrator balance of $1,000,000.00) and admin flags, plus live forms for creating administrators, deleting accounts and approving loans.

The forged token also drove state changes. Tokens with wrong, garbage and empty signatures were each accepted on `POST /admin/approve_loan/{loan_id}` and each credited the full loan amount (see **VULN-08**).

Reproduction Steps

Prerequisite: none. No account and no credential is required.

1. Construct a JSON Web Token locally with header `{"typ":"JWT","alg":"HS256"}` and payload `{"user_id": 2,"username":"<any username>","is_admin":true,"iat":<current epoch>}` .
2. Sign it with any string at all, for example `not_the_real_secret` , or append four arbitrary characters as the signature segment.
3. Send `GET /sup3r_s3cr3t_admin` with header `Authorization: Bearer <your token>` .

   Expected result: HTTP 200 with the full admin panel HTML.

4. Control: repeat step 3 with no `Authorization` header.

   Expected result: HTTP 401 `Token is missing` .

5. Control: repeat step 3 with a genuine token obtained from a normal sign-in on a non-admin test account.

   Expected result: HTTP 403 `Access Denied` . The contrast between steps 3 and 5 is the proof.

6. Optional confirmation that the defect is application-wide: repeat step 1 with `is_admin: false` and a `user_id` belonging to another test account, then call `GET /api/virtual-cards` .

   Expected result: HTTP 200 returning that account's card data.

No cleanup is required. All of the above are read-only requests.

Business Impact

The application has no working authorization boundary. Anyone on the internet can become any customer or the administrator, at will, at no cost. That gives complete read access to the entire customer base and the ability to act as any account holder. Every other finding in this report that begins with the words "any signed-in customer" is effectively unauthenticated in practice, because the sign-in requirement can be satisfied for free. This single defect is what turns several medium findings into serious ones.

Remediation

1. Enable signature verification on every token decode path in the application. This is the fix. Confirm that the decode call is not passing an option that disables verification.
2. Pin the accepted algorithm to a single value and reject every other algorithm, including `none`.
3. Only after verification is on, rotate the signing key to a high-entropy random value held in a secrets store, not in source code or configuration files, and invalidate all existing tokens.
4. Continue to make authorization decisions server-side. Where practical, look up the current `is_admin` value from the database using the verified subject rather than trusting a claim carried in the token.
5. Verify the fix by repeating the case matrix above. A token signed with a wrong secret, a garbage signature and an empty signature must all return HTTP 401, while a genuine token continues to work.

VULN-03

Unauthenticated SQL Injection in the Balance Lookup Gives Full Database Control as a Superuser

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/check_balance/{account_number} (path parameter `account_number`)

CVSS V4.0 BASE SCORE:

9.2 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

An unauthenticated internet request executes attacker-chosen SQL as a database superuser, giving complete read and write control over the banking database and the ability to read files on the database host; no service disruption was performed.

Description

The `account_number` path segment is concatenated directly into a SQL statement with no parameterisation and no input filtering. The endpoint requires no authentication; the published API specification declares it as an open route and live testing confirmed this.

Four layers that should each have stopped this are missing or ineffective:

1. The query is built by string concatenation rather than a prepared statement.
2. The application connects to PostgreSQL 13.23 as the `postgres` superuser, so no file or function privilege restriction applies.
3. The database driver permits multiple statements in one string, so stacked queries execute.
4. Autocommit is off, which is the one partial control present. It is defeated by appending `COMMIT` inside the same injected string.

The result is not a limited data-read flaw. It is arbitrary SQL execution as a database superuser, from the internet, with no credentials.

Evidence

The query is concatenated. A single quote appended to a valid account number returned HTTP 500 with the database's own error text, disclosing the statement shape:

```
{"message":"unterminated quoted string at or near \"'SEED00...'\"
 LINE 1: ...username, balance FROM users WHERE account_number='SEED00...","status":"error"}
```

The returned data is genuine, not reflected input. Three independent proofs were used, because the `account_number` field in the response does echo the request and would not be sufficient on its own. Attacker-chosen sentinel integers were placed in the numeric column, and server-side function results in the text column:

INJECTED READ	RETURNED IN [redacted]	RETURNED IN [redacted]
`version()`	424242.0	`PostgreSQL 13.23 (Debian 13.23-1.pgdg13+1) on x86_64-pc-linux-gnu ...`
information_schema.tables	111.0	`users|loans|transactions|virtual_cards|card_transactions|bill_categories|billers|bill_payments`
`information_schema.columns` for `users`	222.0	`id|username|password|account_number|balance|is_admin|profile_picture|reset_pin`

The version banner is produced by a server-side function call and appears nowhere in the request. The sentinel integers were chosen arbitrarily and returned exactly, proving control of the output columns. A tautology payload against a valid account number returned a different account's real data (`{"username":"admin","balance":1000000.0}`), proving the WHERE clause is fully attacker-controlled. A `sqlite_version()` probe failed with `function sqlite_version() does not exist`, confirming the backend is PostgreSQL.

The database role is a superuser. Confirmed three separate ways: `pg_user.usesuper` returned true, `current_setting('is_superuser')` returned `on`, and `current_database() || current_user || session_user` returned `vulnerable_bank|postgres|postgres`.

Stacked queries execute. A four-point timing ladder was run against a plain-lookup baseline of 0.004 to 0.017 seconds:

INJECTED STATEMENT	ELAPSED
`; SELECT pg_sleep(0)`	0.004 s
`; SELECT pg_sleep(2)`	2.007 s
`; SELECT pg_sleep(5)`	5.010 s
`; SELECT pg_sleep(8)`	8.006 s

A data-dependent control ruled out network variance: `SELECT CASE WHEN (SELECT count(*) FROM users)=6 THEN pg_sleep(5) ELSE pg_sleep(0) END` took 5.010 seconds, while the same statement with `=999` took 0.005 seconds.

Writes persist. A negative control was run first. An `UPDATE ... WHERE id=<test account>` with no `COMMIT` did not persist; the balance was unchanged on read-back. The identical statement with `; COMMIT` appended did persist, and the change was read back through two channels independent of the write path: the injection channel itself, and the application's own `GET /check_balance` endpoint. A row-by-row before-and-after comparison of the users table showed that only the two test-account rows changed and the four other rows were untouched. All writes were bound to test-account rows and all monetary effects were reversed.

Files on the database host are readable. `pg_read_file()` returned the contents of four files chosen at test time, including `/etc/os-release` (`Debian GNU/Linux 13 (trixie)`), `/proc/version`, and the PostgreSQL client authentication configuration file, plus a directory listing of the PostgreSQL data directory. A read of `/etc/shadow` was refused at the operating-system level, which is a file-permission limit rather than a database restriction.

Reproduction Steps

Prerequisite: none. Use read-only payloads unless you are testing on a system where writes are acceptable.

1. Send `GET /check_balance/<valid account number>` with no `Authorization` header and record the normal response as a baseline.
2. Send `GET /check_balance/<valid account number>'` (one trailing single quote).

   Expected result: HTTP 500 containing PostgreSQL error text and a fragment of the SELECT statement.

3. Send a UNION read carrying a server-side function and a sentinel integer, for example a path of `X' UNION SELECT version(), 424242 -- -` (URL-encoded).

   Expected result: HTTP 200 with `balance` equal to `424242.0` and `username` containing the PostgreSQL version banner. The sentinel proves you control the output columns.

4. Confirm the database role with a UNION read of `current_user || '|' || current_setting('is_superuser')`.

   Expected result: `postgres|on`.

5. Confirm stacked-query execution with a timing ladder: send `X'; SELECT pg_sleep(0)-- -` then `X'; SELECT pg_sleep(5)-- -` and compare elapsed time against the baseline from step 1.

   Expected result: roughly 0 seconds and roughly 5 seconds respectively.

6. Only if write testing is authorised, and only against a record you own: run the write with no `COMMIT` first and confirm on read-back that nothing changed, then run it again with `; COMMIT` appended and confirm the change through a second, independent channel.
7. Cleanup: reverse any write with a further committed statement, then confirm the original value through the application's own endpoint rather than through the injection channel.

Business Impact

The whole banking database is under the control of anyone on the internet. They can read every customer record, every balance, every transaction and every stored password. They can also change any value, including balances and the administrator flag, which means they can grant themselves genuine administrator rights rather than forged ones. Because the database role is a superuser, they can additionally read files from the database server. Command execution on that server is a realistic next step from this position; it was deliberately not attempted during this assessment and should be assumed reachable until the injection is fixed.

Remediation

1. Rewrite the balance lookup to use a parameterised query. The account number must be passed as a bound parameter, never concatenated into the statement.
2. Audit every other database call in the application for the same pattern. **VULN-04**, **VULN-05** and **VULN-09** are separate instances of it, so the pattern is systemic rather than isolated.
3. Create a dedicated database role for the application with only the privileges it needs, and stop connecting as `postgres`. This alone removes the file-read capability and would have contained the impact.
4. Configure the database driver to reject multi-statement strings where the driver supports it.
5. Require authentication on the balance lookup and scope it to the caller's own account. See **VULN-07**.
6. Stop returning database error text to clients. See **VULN-15**.
7. Verify the fix by re-running steps 2, 3 and 5 of the reproduction above. The single-quote probe must return a generic error, the UNION probe must return no injected data, and the timing ladder must show no delay.

VULN-04

Unauthenticated SQL Injection in Transaction History Returns Every Customer's Ledger

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/transactions/{account_number} (path parameter `account_number`)

CVSS V4.0 BASE SCORE:

8.7 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

An unauthenticated internet request with a trivially crafted account number returns every customer's transaction history in one response; only data disclosure was demonstrated, with no modification or disruption.

Description

This is a second, distinct injection point with a distinct sink. The `account_number` path segment is interpolated into both sides of an `OR` condition in the ledger query, again with no parameterisation and no authentication. Removing the condition returns the entire transactions table in one response.

The endpoint additionally ships a purpose-built `query` field in its error responses that contains the complete raw SQL statement, which hands an attacker the exact query shape (see also **VULN-15**).

Evidence

A naive test against an account that legitimately has transactions would prove nothing here, because at test time every transaction in the database involved one particular account. The decisive comparison therefore used accounts that legitimately return zero rows:

REQUEST (ALL UNAUTHENTICATED)	ROWS RETURNED	ACCOUNTS PRESENT IN THE RESULT
Baseline: an existing account with no transactions	0	none
Injected: same account with `' OR '1'='1` appended	30	three distinct accounts
Baseline: an account number that does not exist	0	none
Injected: same non-existent account with `' OR '1'='1` appended	30	three distinct accounts

An independent count taken through the separate injection point in **VULN-03** returned `SELECT count(*) FROM transactions = 30`, which confirms the injected result is the entire table and not a superset of one account.

A sample disclosed row, with the account numbers of accounts the caller does not own:

{"id":30,"from_account":"SEED0000..","to_account":"SEED0000..","amount":1.0,
"description":"...","timestamp":"2026-08-27 04:03:46.580441","type":"transfer"}

The error path discloses the full statement. `GET /transactions/<account>'` returned HTTP 500 with a dedicated `query` field:

"query": "SELECT id, from_account, to_account, amount, timestamp, transaction_type, description
FROM transactions WHERE from_account='...'' OR to_account='...''
ORDER BY timestamp DESC"

Reproduction Steps

Prerequisite: none. All steps are read-only.

1. Send `GET /transactions/<an account number that has no transactions>` with no `Authorization` header.

   Expected result: HTTP 200 with an empty `transactions` array. This is the honest baseline.

2. Repeat with `GET /transactions/<same account number>' OR '1'='1` (URL-encoded).

   Expected result: HTTP 200 with the full ledger, containing rows for accounts other than the one requested.

3. Repeat step 2 with an account number that does not exist at all.

   Expected result: the same full ledger. This rules out any chance that the rows legitimately belong to the requested account.

4. Send `GET /transactions/<any account number>'` (one trailing single quote).

   Expected result: HTTP 500 with a `query` field containing the complete SELECT statement.

5. Optional cross-check: count the transactions table through the injection point in **VULN-03** and confirm the count equals the number of rows returned in step 2.

No cleanup is required.

Business Impact

Every payment ever recorded by the bank is readable by anyone on the internet, in a single request, with no account and no account number guessing. That exposes who each customer pays, how much, how often and with what description. This is a complete breach of transaction confidentiality for the entire customer base, and it is the kind of data that supports targeted fraud, blackmail and competitor intelligence long after the technical flaw is fixed.

Remediation

1. Rewrite the ledger query as a parameterised statement with the account number bound as a parameter.
2. Require authentication on this endpoint and return only rows belonging to the signed-in caller, derived from the session rather than from the URL.
3. Remove the `query` field from error responses entirely. It exists only to help debugging and should not be present in a deployed service.
4. Verify the fix by repeating steps 2 and 4 of the reproduction. The injected request must return either an authentication error or zero rows, and the error probe must return a generic message with no SQL text.

VULN-05

SQL Injection in the Sign-In Form Allows Sign-In as Any Customer Without a Password

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/login (JSON body fields `username` and `password`); the same injection class is also reachable through `GET /api/transactions?account_number=`

CVSS V4.0 BASE SCORE:

8.7 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

An unauthenticated attacker logs in as any customer without knowing the password by placing a comment terminator in the username field, obtaining a genuine session; the session was exercised for data access only during testing.

Description

The credential lookup builds its SQL by concatenating both the username and the password directly into the statement. A comment terminator placed at the end of the username removes the password comparison from the query, so the lookup matches on username alone and the application issues a genuine, correctly signed session token.

This is separate from **VULN-02**. Here the server issues a real token of its own accord, so the attack works even if signature verification is switched back on.

Evidence

A negative control was run first to establish that password checking normally works, which is what isolates SQL injection as the bypass mechanism rather than a lucky guess:

REQUEST	RESULT
Correct username, deliberately wrong password	HTTP 401 `Invalid credentials`
Username with a trailing comment terminator, random wrong password	HTTP 200 with a valid session token

The successful response body contained a genuine token and full account context:


```
{"status":"success","message":"Login successful","accountNumber":"SEED0000..","isAdmin":false,
 "debug_info":{"user_id":2,"username":"<victim>","is_admin":false,"account_number":"SEED0000..",
               "login_time":"2026-08-27 04:13:59"},
 "token":"eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9..."}
```


The token was checked cryptographically and its signature is authentic under the server's real signing key, confirming it was genuinely issued by the server rather than accepted through the separate signature defect. Used as a bearer credential it returned HTTP 200 on `GET /dashboard`, on `GET /api/virtual-cards` (returning the unmasked card number), and on `GET /api/bill-payments/history`.

The concatenation is visible in the error path. Sending a username of `<name>'` returned HTTP 500 disclosing the raw statement, including the submitted password as a literal string:


```
{"error":"syntax error at or near \"wrongpass123\"
 LINE 1: ... FROM users WHERE username='<name>'" AND password="wrongpass1...",
 "message":"Login failed"}
```


That disclosure also proves the password is compared as a plain string literal in SQL, which independently corroborates **VULN-06**.

Reproduction Steps

Prerequisite: a username. No password is required.

1. Send `POST /login` with `{"username":"<test account>","password":"definitely_wrong_zz9"}`.

   Expected result: HTTP 401 `Invalid credentials`. This control proves the password check normally works.

2. Send `POST /login` with `{"username":"<test account>'--","password":"<any random string>"}`.

   Expected result: HTTP 200 with a `token` field.

3. Use the returned token as `Authorization: Bearer <token>` against `GET /api/bill-payments/history`.

   Expected result: HTTP 200 with that account's data, confirming the session is real and usable.

4. Send `POST /login` with `{"username":"<test account>'","password":"anything"}`.

   Expected result: HTTP 500 disclosing the raw SELECT statement with both submitted values inlined.

No cleanup is required. No state is changed by these steps.

Business Impact

Anyone who knows or can guess a username can sign in as that customer without a password. The session issued is a real one, so it works everywhere the customer's own session works. Combined with **VULN-11**, that session then never expires. From there the attacker can read statements and card details and move money. Because the failed-login path also prints the submitted password back inside the SQL statement, the sign-in endpoint additionally leaks credential handling detail to unauthenticated callers.

Remediation

1. Rewrite the credential lookup as a parameterised query. Neither the username nor the password may be concatenated into SQL.
2. Look up the user by username only, then compare the supplied password against a stored password hash in application code. See **VULN-06**.
3. Return a generic error on failure and stop echoing exception text. See **VULN-15**.
4. Remove the `debug_info` object from sign-in responses.
5. Apply the same review to `GET /api/transactions?account_number=`, which is reachable by the same injection class.
6. Verify the fix by repeating steps 2 and 4 of the reproduction. The comment-terminator sign-in must return HTTP 401, and the quote probe must return a generic error.

VULN-06

Customer Passwords Are Stored as Readable Text

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com, `users.password` database column (readable through the injection point in **VULN-03**)

CVSS V4.0 BASE SCORE:

8.7 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Because passwords are stored as readable text and the database is already readable by an unauthenticated internet attacker, every customer credential in the system is recoverable with no cracking effort; the demonstrated impact is credential disclosure only.

Description

The `users.password` column holds the password exactly as the customer set it. There is no hash, no salt and no algorithm prefix. This is not a weak hashing choice; there is no hashing at all.

Because the database is already readable without any credentials through **VULN-03**, this is not a hypothetical exposure that would only matter after some future breach. Every password in the system is recoverable today.

Evidence

The stored value was read three times for a single test account, each time against a password that was independently known to be in force at that moment. The middle value was a string invented roughly sixty seconds before the read, which eliminates any possibility of a pre-existing artefact or a lucky match:

READ	PASSWORD KNOWN TO BE IN FORCE	VALUE FOUND IN USERS.PASSWORD	STORED LENGTH
1	The account's password at the start of testing	Byte-identical to it	19
2	A string invented about 60 seconds earlier and set through the reset flow	Byte-identical to it	18
3	The account's restored password	Byte-identical to it	16

In every case the stored length equals the exact character count of the password. Those lengths are inconsistent with every standard hash encoding: bcrypt is a fixed 60 characters with a `$2b$` prefix, MD5 is 32 hex characters, SHA-1 is 40, SHA-256 is 64, and Argon2 or PHC formats run to roughly 95 characters with algorithm markers. No salt and no algorithm prefix are present.

This is independently corroborated by the sign-in error path described in **VULN-05**, which discloses `SELECT * FROM users WHERE username='...' AND password='...'`. The password is compared as a plain string literal inside the SQL statement, which is only possible if the stored value is the password itself.

The same query also showed that `users.reset_pin` is stored in readable form and is set to NULL after a successful reset.

Reproduction Steps

Prerequisite: a test account whose current password you know, and the injection point from **VULN-03**. Use a test account only.

1. Read the stored value and its length for your test account through a UNION read of `password` and `length(password)` from the `users` table.

   Expected result: the returned string matches your known password exactly, and the length equals its character count.

2. Change the test account's password to a new value you invent at that moment, using the normal reset flow.
3. Repeat the read from step 1.

   Expected result: the stored value now matches the string you invented seconds earlier, again byte for byte. This step is what rules out a coincidental match.

4. Restore the account's original password and repeat the read once more to confirm the restored state.

Cleanup: confirm the test account signs in successfully with its original password and that the temporary password now returns HTTP 401.

Business Impact

Every customer password held by the bank is exposed and usable immediately, with no cracking work and no delay. Because most people reuse passwords, the damage extends well beyond this application to the customers' email, other banks and payment services. This also removes any defence in a dispute, because the bank cannot argue that a stolen credential would have been protected at rest. Recovery requires a forced password change for every customer, not just a code fix.

Remediation

1. Add a modern password hashing function with a per-user salt and a tuned work factor. Argon2id is the current preferred choice; bcrypt or scrypt are acceptable.
2. Change the sign-in path to fetch the user by username and verify the supplied password against the stored hash in application code, never inside a SQL comparison.
3. Migrate existing accounts. Because the stored values are readable, they can be hashed in place in a single pass, but every customer should still be required to set a new password, since all current passwords must be treated as compromised.
4. Apply the same treatment to `users.reset_pin`, and stop returning it anywhere. See **VULN-01**.
5. Verify the fix by reading the `users.password` column for a test account after setting a known password. The stored value must not match the password, must be a fixed-length hash with an algorithm prefix, and must differ between two accounts that share the same password.

VULN-07

Account Balances and Transaction Histories Are Served Without Any Authentication

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/check_balance/{account_number} and /transactions/{account_number}

CVSS V4.0 BASE SCORE:

8.7 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Customer balances and complete transaction histories are served to anyone on the internet with no credentials at all, and account numbers are sequential and enumerable, so the whole customer base is readable; no modification was possible through this path.

Description

This is a missing-authentication defect in its own right, separate from the injection findings that affect the same two endpoints. Even with the SQL flaws in **VULN-03** and **VULN-04** fully fixed, any internet user could still read any customer's balance and full ledger simply by supplying that customer's account number. The published API specification declares both routes as open, and live testing confirmed it.

Account numbers follow a short, predictable, sequential pattern, so an attacker does not need to know them in advance. The keyspace was walked and the real accounts were identified by which numbers returned data rather than HTTP 404.

Evidence

The account number keyspace was enumerated with no authentication. Fifteen sequential candidates were requested; five returned live account data and the rest returned HTTP 404 `Account not found`. Nothing throttled, blocked or logged a challenge during the walk.

Reads succeeded in every credential state, which is what distinguishes this from an authorization flaw:

CREDENTIAL PRESENTED	RESULT
No token at all	HTTP 200 with the account's real balance and username
A valid token belonging to a different customer	HTTP 200 with the same data
A garbage token	HTTP 200 with the same data

Balances for accounts outside the test account set were returned, including one at $10,000.00 and the administrator account at $1,000,000.00. Reading a ledger for another account returned a 29,769-byte response containing the full transaction history.

The same data is also readable from any third-party website in a visitor's browser, because the server reflects arbitrary origins in its cross-origin headers and these endpoints need no credentials. See **HARDEN-03**.

Reproduction Steps

Prerequisite: none.

1. Send `GET /check_balance/SEED000001` with no `Authorization` header and no cookies.

   Expected result: HTTP 200 with `username` and `balance` for that account.

2. Increment the numeric portion and repeat for a short range, for example the first fifteen values.

   Expected result: live accounts return HTTP 200 with data and unused numbers return HTTP 404 `Account not found`. The difference between the two responses is the enumeration oracle.

3. Send `GET /transactions/<an account number found in step 2>` with no credentials.

   Expected result: HTTP 200 with that account's complete transaction history.

4. Control: repeat step 1 with a valid token for a different account, and again with a deliberately malformed token.

   Expected result: identical HTTP 200 responses in both cases, confirming the endpoint ignores credentials entirely rather than mis-checking them.

No cleanup is required.

Business Impact

The bank's customer base is effectively a public directory. Anyone can walk the account numbers and pull down names, balances and complete payment histories for every customer, with no account, no password and no rate limit to slow them down. This alone is a reportable data breach, and it needs no skill to carry out. It also feeds the other findings: knowing which account numbers are live is the first step in the stored-scripting attack in **VULN-10**, which requires only a recipient account number.

Remediation

1. Require authentication on both endpoints.
2. Derive the account from the signed-in session rather than from the URL, and reject any request for an account the caller does not own.
3. Replace sequential account numbers with values that cannot be guessed, or at minimum return an identical response for "not your account" and "does not exist" so the endpoints cannot be used to enumerate.
4. Add rate limiting on these endpoints, keyed on a value the caller cannot choose. See **HARDEN-05**.
5. Verify the fix by repeating steps 1 and 3 of the reproduction with no credentials (expect HTTP 401) and with a valid token belonging to a different customer (expect HTTP 403 or an empty result).

VULN-08

Loan Approval Creates Money at an Attacker-Chosen Amount and Can Be Replayed

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/admin/approve_loan/{loan_id}

CVSS V4.0 BASE SCORE:

8.7 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

An unauthenticated attacker can approve loans with a self-asserted administrator token and have an amount of their own choosing credited to an account, repeatedly, creating money out of nothing; some internal loan and account detail is echoed back in the process.

Description

Two independent failures combine here.

First, the administrator check on this endpoint does work when it is given a genuine token. A real non-admin token is correctly refused with HTTP 403. The problem is that the check reads a claim inside a token whose signature is never verified (**VULN-02**), so an attacker satisfies it for free.

Second, there is no terminal-state check. A loan already in `approved` status is approved again on every request, and the full amount is credited again each time. There is also no separation of duties: one identity can approve another user's loan, and the response records the approver's name against the other user's account.

The credited amount tracks the amount on the loan record, and **VULN-22** shows that amount is completely unbounded. The attacker therefore chooses the figure.

Evidence

The authorization ladder. All requests were sent to the same endpoint, and the balance delta was measured after each:

TOKEN PRESENTED	RESULT	BALANCE DELTA
No token	HTTP 401 `Token is missing`	none
Malformed string `aaa.bbb.ccc`	HTTP 401 `Invalid token`	none
Genuine non-admin token	HTTP 403 `Access Denied`	none
`is_admin: true` , signed with a deliberately wrong secret	HTTP 200 `Loan approved successfully`	+$1,000.00
`is_admin: true` , signature replaced with the literal `AAAA`	HTTP 200	+$1,000.00
`is_admin: true` , signature left empty	HTTP 200	+$1,000.00

The third row is the control that proves the admin gate itself is functioning. The last three rows prove the gate is satisfied by a self-written claim.

Replay against an already-approved loan. Starting from a loan whose status was already `approved` , and where the server's own response body echoed `loan_details.status: "approved"` :

- Replay 1: HTTP 200, balance +$1,000.00
- Replay 2: HTTP 200, balance +$1,000.00

Loan status never changes and no idempotency key is used. The handler's own output shows the missing check, because it reports the pre-update status while crediting the money.

The attacker chooses the amount. Two competing explanations were tested with predictions declared in advance: either the endpoint credits a hardcoded $1,000, or it credits the amount on the loan record. A loan was filed for $7,777.77 and approved with a self-written token. The balance moved from $9,989.00 to $17,766.77, a delta of exactly +$7,777.77. The hardcoded explanation predicted $10,989.00 and missed by $6,777.77. Amount tracking was observed at three distinct face values in separate sessions: $1,000, $2,000 and $7,777.77.

Cross-user approval. A token asserting one identity approved a loan belonging to a different user. The response contained `{approved_by: "<first identity>", user_id: <second user>, loan_amount: 2000.0}` and the second user's balance rose by exactly $2,000.00.

All monetary effects were reversed. Balances finished the engagement at their starting values.

Reproduction Steps

Prerequisite: a loan record you own, in any status. This test creates money, so run it only against a test account and reverse it afterwards.

1. Record the test account's balance through `GET /check_balance/<account number>`.
2. Control: send `POST /admin/approve_loan/{loan_id}` with a genuine non-admin token.

   Expected result: HTTP 403 `Access Denied`, no balance change. This proves the admin gate fires.

3. Build a token locally with payload `{"user_id":<your test user>,"username":"<name>","is_admin":true,"iat":<now>}` and sign it with any arbitrary string.
4. Send `POST /admin/approve_loan/{loan_id}` with that token.

   Expected result: HTTP 200 `Loan approved successfully`, and the balance rises by exactly the loan's face value.

5. Send the identical request again without changing anything.

   Expected result: HTTP 200 again, and the balance rises by the same amount a second time. This is the missing terminal-state check.

6. Cleanup: reverse the credited amount and confirm the restored balance through `GET /check_balance`. Note that the loan row itself cannot be returned to `pending`, because the API has no route to reject, unapprove or delete a loan.

Business Impact

This is direct, repeatable, unlimited money creation by someone who does not even need an account. The attacker files a loan for any figure they like, approves it themselves with a token they wrote, and repeats the approval as many times as they want. Each repeat pays out again. Losses are immediate, are limited only by how long the attacker keeps sending requests, and appear in the ledger as legitimate approved loans, which makes them hard to spot and hard to reverse.

Remediation

1. Fix token signature verification first. See **VULN-02**. Nothing else on this endpoint matters until that is done.
2. Add a terminal-state check so a loan in `approved` (or any non-pending state) cannot be approved again. Make the approval operation idempotent.

3. Add a separation-of-duties check so the approving identity cannot be the borrower, and record the approver from the verified session rather than from a request field.
4. Recheck the current `is_admin` value in the database using the verified subject rather than trusting the claim in the token.
5. Bound the loan amount at the point of application. See **VULN-22**.
6. Add a route to reject or reverse a loan. There is currently no way for an operator to undo an approval, which is itself an operational gap.
7. Verify the fix by repeating steps 4 and 5 of the reproduction. A self-signed token must return HTTP 401, and a genuine admin approving the same loan twice must succeed once and be refused the second time.

VULN-09

Card Limit Endpoint Builds Database Commands From Client-Supplied Field Names

AFFECTED ASSET:
https://aldercrest.targets.hound-agents.com/api/virtual-cards/{card_id}/update-limit (JSON body keys)

CVSS V4.0 BASE SCORE:
8.6 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:
A logged-in caller can run arbitrary database statements as a superuser through a card-management endpoint, reading data from any table and writing to any card row regardless of which card the URL names or who owns it.

Description

The handler takes each key from the caller's JSON body and interpolates it directly into the SQL `SET` clause as a column name. The published API specification declares that this endpoint accepts exactly one field, a numeric `limit`, so every column reached is outside the documented contract.

This has two levels of impact.

At the simple level it is mass assignment. Any column of the `virtual_cards` row can be written by naming it as a JSON key, including the card's spendable balance, its card number, its security code and the user it belongs to.

At the deeper level it is a general-purpose SQL execution primitive. Because the key is placed into the statement as text, an attacker can close the assignment, add their own WHERE clause, append further statements, and embed subqueries. Multi-statement batches execute, and the database role is the same `postgres` superuser used by the unauthenticated injection in **VULN-03**.

Two precise limits are worth recording, because they narrow remediation. The value position is not injectable; values are safely bound by the database driver. And the injection is strictly key-only, which means the fix is confined to this one endpoint. Sibling endpoints were tested and do not share the defect.

Evidence

Keys become column names. A body key that is not a real column returns the generated statement with line numbers:

```
{"status":"error",
 "message":"column \"vrf5_probe_col\" of relation \"virtual_cards\" does not exist
        LINE 3:            SET vrf5_probe_col = 1.0
                           ^"}
```

Further probes disclosed lines 3, 4 and 5 of the template verbatim: `SET {key} = {value}`, `WHERE id = 3`, `RETURNING *`.

Values are safe; only keys are not. A bogus column with a string value rendered as `SET vrf5_probe_col = 'HELLO'`, correctly quoted. A classic value-breakout payload of `x' , cvv = '000` submitted as a card number was stored literally as the card number, and the security code column was untouched.

Every column is writable, across ownership boundaries. Acting as one test account against a card owned by a different test account, with each change confirmed in the true owner's own signed-in view and through an independent database read, and each change restored before the next was attempted:

FIELD WRITTEN	BEFORE	AFTER	CONFIRMED BY
`current_balance`	$0.00	$99,999.00	Owner's own card listing showed balance 99999.0
user_id	owner A	owner B	Card vanished from owner A's list and appeared in owner B's list
`card_number`	original value	`ATTACKERCARD`	Owner's own card listing
is_frozen	false	true	Owner's own card listing

Arbitrary SQL, not just mass assignment. Multi-statement execution was confirmed with an eight-point interleaved timing ladder against a plain-update baseline of 0.008 to 0.017 seconds:

INJECTED STATEMENT	ELAPSED
`pg_sleep(0)`	0.011 s
pg_sleep(2)	2.011 s and 2.012 s
`pg_sleep(5)`	5.015 s
`pg_sleep(5)` followed by a deliberate syntax error	0.008 s (control: the batch is parsed before any of it runs)

A data-dependent oracle ruled out network variance: a conditional sleep evaluating to true took 5.016 seconds and the same statement evaluating to false took 0.014 seconds.

Cross-table reads succeeded through a subquery written into a readable card column, returning `postgres|vulnerable_bank|on` (the connecting role, the database name, and superuser status) and the complete list of tables in the schema.

The URL does not scope the write. A POST to card 3's URL, carrying an attacker-authored WHERE predicate, changed card 1's limit from 500.00 to 555.00 while card 3 itself stayed at 1000.00. Repeated across an ownership boundary, a request to card 3's URL changed a different owner's card 2 from 1000.00 to 1234.00, confirmed in that owner's own session. All values were restored.

Reproduction Steps

Prerequisite: any syntactically valid session token and a card record you own. Restore every value you change.

1. Read the current state of your card through `GET /api/virtual-cards` and record its limit and balance.
2. Send `POST /api/virtual-cards/{your card id}/update-limit` with body `{"zzz_probe_col":1}`.

   Expected result: HTTP 500 containing `column "zzz_probe_col" of relation "virtual_cards" does not exist` and the generated `SET` line. This confirms the key reaches SQL as a column name.

3. Send the same endpoint with body `{"card_limit":777}`.

   Expected result: HTTP 200 `Card updated successfully`, and `GET /api/virtual-cards` shows the new limit. This confirms undocumented columns are writable.

4. Confirm multi-statement execution safely with a timing probe. Send a key of the form `card_limit = <current value>, card_limit` first to observe the duplicate-column error, then a key that appends `; SELECT pg_sleep(5)` and measure the elapsed time against a plain update.

   Expected result: roughly five seconds versus a baseline under a fifth of a second.

5. Confirm the value position is safe: send `{"card_number":"x' , cvv = '000"}` against a test card.

   Expected result: the literal string is stored as the card number and the security code is unchanged.

6. Cleanup: restore the limit, balance, card number and security code to the values recorded in step 1 and confirm through `GET /api/virtual-cards` as the card's owner. Note that a literal `%` character in a crafted key aborts the request, so avoid it.

Business Impact

A single card management endpoint gives a signed-in caller the same power over the database that the unauthenticated injection gives an outsider. In practical terms, an attacker can top up any card's spendable balance to any figure and then spend it, take ownership of any customer's card, rewrite card numbers and security codes, and read any table in the bank. The sign-in requirement is not a real barrier, because **VULN-02** lets anyone create a valid-looking session for free. The endpoint's own error messages hand the attacker the exact query template needed to build the payload.

Remediation

1. Stop building SQL from request body keys. Accept only the single documented field, map it to a fixed column name in code, and bind the value as a parameter.
2. Add a server-side allow-list of writable fields for this resource and reject any body containing an unexpected key.
3. Add an ownership check so the caller can only modify a card they own, and use the URL card id as the sole row selector.
4. Give the application a limited database role instead of the `postgres` superuser. See **VULN-03**.
5. Stop returning generated SQL and line numbers in error responses. See **VULN-15**.
6. Verify the fix by repeating steps 2, 3 and 4 of the reproduction. The probe key must be rejected with a generic validation error, undocumented columns must not change, and the timing probe must show no delay.

VULN-10

Stored Cross-Site Scripting in Transfer Descriptions Steals Customer Sessions

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/dashboard (transaction history section), written through the `description` field of `POST /transfer`

CVSS V4.0 BASE SCORE:

8.2 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:P/VC:H/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Any ordinary customer account can plant executable code into another customer's account history over the internet, and the victim only has to view their own dashboard for their session to be stolen and used to read card data and move money.

Description

The `description` field of a money transfer is stored exactly as submitted and is later written into the dashboard using a raw HTML assignment at `static/dashboard.js` line 252, inside a template that is assigned with `innerHTML` at line 261. There is no escaping on write, no encoding on render, and no content security policy to catch the failure.

The recipient does not have to accept, approve or have any relationship with the sender. Anyone can send anyone else a transfer, and account numbers are enumerable through **VULN-07**. The victim only has to look at their own transaction history.

A note on payload shape: a plain `<script>` tag is stored but does not run, because HTML inserted with `innerHTML` does not execute script elements. An event-handler payload such as `<img src=x onerror=...>` does run, and that is the realistic form of the attack.

Evidence

An unsolicited $1 transfer was sent from one test account to another, carrying an image tag with an `onerror` handler in the description. The stored value was compared byte for byte against what was sent and was identical, held as ledger row 65.

The victim then signed in normally through the real sign-in form in a Firefox browser, with no cookie or token injection of any kind. On loading their own dashboard:

- The payload was present in the live DOM as a real `<img>` element, matched by `document.querySelector('img[onerror*="..."]')`, inside four separate transaction-description nodes.
- The injected code executed in the victim's origin, setting a marker variable that was read back from the page.
- The code read `localStorage.jwt_token` and sent it to a listener controlled by the test. The listener recorded one hit carrying the victim's live session token, whose claims decoded to the victim's own user id.
- No content security policy header is served on `/dashboard`, so no violation could be raised and nothing blocked the execution.

The stolen token was then replayed to confirm it is a working credential and not just a string:

REPLAY	RESULT
`GET /api/virtual-cards` with the stolen token	HTTP 200, returning the victim's full unmasked card number and security code
`GET /api/bill-payments/history`	HTTP 200
`POST /transfer` with the stolen token	HTTP 200 `Transfer Completed`, money moved out of the victim's account
Same token after the victim performed an in-app sign-out	HTTP 200, still valid (see **VULN-11**)

The payload persisted after submission and re-fired on every dashboard render, including when the dashboard was loaded inside a frame. The API has no route to delete a transaction, so application-level cleanup is unavailable.

Reproduction Steps

Prerequisite: two test accounts, one acting as sender and one as victim. Use a browser for the victim, not a command-line client, because the proof depends on real rendering.

1. As the sender, send `POST /transfer` with body `{"to_account":"<victim account number>","amount":1,"description":"<img src=x onerror=\"window.__probe=1\">"}`.

   Expected result: HTTP 200 `Transfer Completed`.

2. Confirm the payload was stored unchanged by reading `GET /transactions/<victim account number>` and comparing the stored description byte for byte against what you sent.

   Expected result: identical.

3. In a clean browser profile, sign in as the victim through the real sign-in form and open the dashboard.
4. Read back `window.__probe` in the page console.

   Expected result: the value is `1`, proving the payload executed in the victim's origin.

5. To confirm the impact, replace the payload with one that reads `localStorage.getItem('jwt_token')` and sends it to a listener you control, then repeat steps 1 to 3 and confirm the listener receives the victim's token.
6. Confirm the token is a working credential by using it as `Authorization: Bearer` against `GET /api/virtual-cards`.

   Expected result: HTTP 200 with the victim's card data.

7. Cleanup: the transaction rows you create cannot be deleted through the API. Record the ledger row ids you created so they can be purged server-side, and reverse the $1 transfer with a matching transfer in the other direction.

Business Impact

Any customer, or anyone who mints a session for free through **VULN-02**, can take over any other customer's account by sending them one dollar. The victim does nothing wrong; they simply look at their own statement. The stolen session gives full access to card details and the ability to move money, and because sessions never expire (**VULN-11**) the access is permanent. If a member of staff or an administrator ever views a transaction history containing a payload, the same theft applies to their session.

Remediation

1. Change the render path at `static/dashboard.js` line 252 to insert user-controlled text as text, using `textContent` or an equivalent, rather than through `innerHTML`.
2. Encode all user-controlled values on output across the application, not just in this one place.
3. Add a content security policy that blocks inline event handlers and inline script. This would have stopped the attack independently of the encoding fix. See **HARDEN-01**.
4. Validate and restrict the `description` field on write, for example to a length limit and a plain-text character set.
5. Move the session token out of `localStorage` and into a cookie marked `HttpOnly`, `Secure` and `SameSite`, so a scripting flaw cannot read it. See **HARDEN-02**.
6. Verify the fix by repeating steps 1 to 4 of the reproduction. The payload must appear as visible literal text in the dashboard and `window.__probe` must be undefined.

VULN-11

Session Tokens Never Expire and Cannot Be Revoked

AFFECTED ASSET:
https://aldercrest.targets.hound-agents.com/login (token issuance) and all token-protected endpoints

CVSS V4.0 BASE SCORE:
7.1 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:
An attacker who has obtained a customer's session token keeps full remote access to that account indefinitely, and the account owner cannot revoke it even by changing their password; the demonstrated impact was complete read access to the account.

Description

Issued tokens carry only `{user_id, username, is_admin, iat}`. There is no `exp`, no `nbf` and no `jti`, so nothing bounds their lifetime and nothing identifies an individual token for revocation.

There is also no sign-out on the server. The client-side sign-out only clears the token from browser storage; the token itself remains valid.

Most seriously, revocation is not merely missing but structurally impossible in the current design. The database has eight tables and none of them is a session or token store, and the `users` table has no `token_version`, no `session_epoch` and no `password_changed_at` column. The server retains no state that could be used to reject a previously issued token.

Evidence

A fresh token was obtained through a normal sign-in and decoded. Its claim set was exactly `['iat','is_admin','user_id','username']`, with `exp`, `nbf` and `jti` all absent.

The account's password was then changed. Before drawing any conclusion from the old token, the password change was proved genuinely effective three separate ways, because a silently failed reset would make a surviving token prove nothing:

CHECK	RESULT
Sign-in with the old password	HTTP 401 Invalid credentials
Sign-in with the new password	HTTP 200 Login successful
Stored credential fingerprint	Changed, and matched the fingerprint of the new password

The pre-change token was then replayed:

REQUEST WITH THE PRE-CHANGE TOKEN	RESULT
`GET /api/virtual-cards`	HTTP 200, returning the account's full unmasked card number
`GET /api/bill-payments/history`	HTTP 200
`GET /api/transactions?account_number=...`	HTTP 200 with transaction data
Same token sent as a cookie only, with no `Authorization` header	HTTP 200
Control: no token at all	HTTP 401 `Token is missing`

The no-token control is what proves those HTTP 200 responses are genuine authenticated access rather than an ungated endpoint.

Separately, a token stolen through the stored-scripting path in **VULN-10** continued to return HTTP 200 after the victim performed an in-app sign-out, and tokens issued more than an hour earlier and superseded by several newer sign-ins were also still accepted.

No sign-out route exists at any variant: `GET /logout`, `POST /logout`, `GET /api/logout` and `POST /api/v1/logout` all return HTTP 404, and the published 26-path API specification contains no path matching logout, revoke, session or token.

Reproduction Steps

Prerequisite: a test account you control.

1. Sign in and record the returned token. Decode it and confirm the claim set contains no `exp`, `nbf` or `jti`.
2. Call `GET /api/virtual-cards` with the token and confirm HTTP 200.
3. Change the account's password through the normal reset flow.
4. Prove the change took effect: sign in with the old password (expect HTTP 401) and with the new password (expect HTTP 200). This control is essential.
5. Replay the token recorded in step 1 against `GET /api/virtual-cards`.

   Expected result: HTTP 200 with full account data.

6. Control: repeat step 5 with no `Authorization` header.

   Expected result: HTTP 401 `Token is missing`.

7. Confirm no revocation surface exists by requesting `/logout`, `/api/logout` and `/api/v1/logout` with both GET and POST.

   Expected result: HTTP 404 for all.

8. Cleanup: restore the test account's original password and confirm with a successful sign-in.

Business Impact

Once a session token is captured, the attacker has that account forever. There is nothing the customer or the bank can do about it through the application. The customer's natural response to a suspected compromise, changing their password, is proved not to work. Support staff have no way to end a session, and the bank has no way to force a mass sign-out during an incident.

This is not a theoretical prerequisite. Two separate, working token-theft paths are established in this report (**VULN-10** and **VULN-18**), and a third route lets an attacker create tokens without stealing anything (**VULN-02**). Fixing revocation requires a database change, so it should be planned rather than treated as a quick patch.

Remediation

1. Add an `exp` claim to every issued token with a short lifetime, and add a refresh mechanism if longer sessions are needed.
2. Add a `jti` claim and a server-side store of revoked identifiers, or add a `token_version` column to the `users` table that is incremented on password change and compared on every request. Either approach requires a schema change.
3. Add a sign-out route that revokes the presented token server-side.
4. Invalidate all outstanding sessions automatically when a password is changed or reset.
5. Verify the fix by repeating the reproduction. After a password change, the pre-change token must return HTTP 401, and a token older than the configured lifetime must also return HTTP 401.

VULN-12

Negative Transfer Amounts Create Money and Push Other Accounts Below Zero

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/transfer (JSON body field `amount`)

CVSS V4.0 BASE SCORE:

7.1 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Any ordinary customer can inflate their own balance without bound and drive any other customer's balance arbitrarily negative by sending a negative-amount transfer, giving the attacker direct control over account balances.

Description

The transfer handler checks whether the requested amount exceeds the sender's balance, but it compares the magnitude of the amount rather than the signed value. The update that follows applies the signed value. A negative amount therefore passes the guard and then runs the arithmetic backwards: the sender is credited and the recipient is debited.

There is also no floor on the receiving account. A recipient can be driven below zero and then further below zero with repeated requests.

The root cause was isolated rather than inferred from the symptom, and that distinction matters for the fix.

Evidence

The guard compares magnitude, not sign. Against a sender balance of $18,987.00, two amounts of equal magnitude and opposite sign were submitted:

AMOUNT SUBMITTED	RESULT	BALANCE MOVEMENT
-28,987.00	HTTP 400 `Insufficient funds`	none
+28,987.00	HTTP 400 `Insufficient funds`	none

Both were rejected identically. A signed comparison would have accepted or rejected them differently. This is what proves the guard is `abs(amount) <= balance`.

Normal behaviour, established first as a control. A legitimate +$10.00 transfer moved the sender from $18,987.00 to $18,977.00 and the recipient from $13,976.00 to $13,986.00, in the expected direction, and was reversed cleanly.

The defect. An amount of -25 returned HTTP 200 `Transfer Completed`:

ACCOUNT	BEFORE	AFTER	DELTA
Sender	$18,987.00	$19,012.00	+$25.00
Recipient	$13,976.00	$13,951.00	-$25.00

No recipient floor. The recipient was first drained legitimately to $20.00. A -50 transfer then left them at -$30.00, and a further -40 transfer left them at -$70.00. Neither request was refused.

Both accounts were restored to a net change of exactly $0.00.

Reproduction Steps

Prerequisite: two test accounts, one as sender and one as recipient. This changes balances, so restore them afterwards.

1. Record both balances through `GET /check_balance/<account number>` for each.

2. Control: send a legitimate `POST /transfer` of `{"to_account":"<recipient>","amount":10}` and confirm the sender falls by $10 and the recipient rises by $10. Reverse it.
3. Root-cause probe: send an amount whose magnitude exceeds the sender's balance, first negative and then positive, for example -28987 and +28987.

   Expected result: both return HTTP 400 `Insufficient funds` with no balance movement. The identical treatment is the proof.

4. Send `POST /transfer` with `{"to_account":"<recipient>","amount":-25}`.

   Expected result: HTTP 200 `Transfer Completed`. Read both balances again and confirm the sender rose by $25 and the recipient fell by $25.

5. Cleanup: send a matching positive transfer to restore both accounts, then confirm the balances match the values recorded in step 1.

Business Impact

Any customer can print money for themselves in one request, with no limit other than how many times they choose to send it. They can also push any other customer's balance below zero without that customer's involvement or consent, which produces false overdrafts, failed payments and fee charges for innocent people. The bank's ledger becomes unreliable, and because each request looks like an ordinary completed transfer, the activity is not obviously fraudulent in the transaction record.

Remediation

1. Reject any transfer amount that is not strictly greater than zero, before any balance check.
2. Change the funds check to compare the signed value against the sender's balance, not the magnitude.
3. Add a check that the receiving account's balance remains valid after the update, and reject transfers that would take it below zero unless overdraft is an intended product feature.
4. Enforce these rules as database constraints as well as in application code, so a future code path cannot bypass them.
5. Apply the same amount validation to the other money workflows. **VULN-22** shows the loan endpoint has the same gap.
6. Verify the fix by repeating step 4 of the reproduction. A negative amount must return a validation error and no balance may move.

VULN-13

Any Signed-In Customer Can Read Any Other Customer's Transaction History

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/api/transactions?account_number={value}

CVSS V4.0 BASE SCORE:

7.1 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Any logged-in customer can read any other customer's complete transaction history simply by changing an account number in the request; the demonstrated impact is disclosure only.

Description

This endpoint takes the account number from a query parameter supplied by the caller instead of deriving it from the signed-in session. No check is made that the requested account belongs to the caller.

Authentication is genuinely enforced here, which is what isolates the defect as a missing authorization check rather than a missing authentication check.

Evidence

Authentication was isolated from authorization first:

CREDENTIAL	RESULT
No token	HTTP 401 `Token is missing`
Malformed token	HTTP 401 `Invalid token`
A valid token, requesting the caller's own account	HTTP 200 with the caller's ledger
A valid token, requesting a different customer's account	HTTP 200 with that customer's ledger

The crossing was demonstrated in both directions between two test accounts: one account's token returned the other's 29-row ledger, and the reverse request returned the first account's 34-row ledger. All four accounts outside the test account set also returned HTTP 200 with data rather than a denial.

Reproduction Steps

Prerequisite: one valid session token and a second account number.

1. Control: send `GET /api/transactions?account_number=<any account>` with no `Authorization` header.

   Expected result: HTTP 401 `Token is missing`. This proves authentication is enforced.

2. Send `GET /api/transactions?account_number=<your own account>` with a valid token and record the response.

   Expected result: HTTP 200 with your own ledger.

3. Send the same request with a different customer's account number, keeping the same token.

   Expected result: HTTP 200 with that customer's ledger. This is the authorization failure.

4. Repeat step 3 in the opposite direction using the second account's token, to confirm the defect is not specific to one account.

No cleanup is required.

Business Impact

Every customer can read every other customer's payment history from inside the application. Because a valid session can be created for free through **VULN-02**, this is effectively open to anyone. The exposed data shows who each customer pays, how much and how often, which supports fraud, social engineering and privacy harm. A single script could harvest the entire bank's transaction history in minutes.

Remediation

1. Remove the `account_number` query parameter and derive the account from the signed-in session.
2. If callers must be able to name an account, verify ownership against the verified session subject before returning any data, and return HTTP 403 otherwise.
3. Apply the same ownership check to every endpoint that takes a resource identifier from the caller. **VULN-14** and **VULN-20** are the same defect on the card endpoints.
4. Verify the fix by repeating step 3 of the reproduction. The request must return HTTP 403 or an empty result, while step 2 continues to work.

VULN-14

Full Card Numbers, Security Codes and Expiry Dates Are Returned and Stored Unprotected

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/api/virtual-cards, /api/virtual-cards/create and /api/virtual-cards/{card_id}/transactions

CVSS V4.0 BASE SCORE:

7.1 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:H/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

A logged-in caller receives complete unmasked card numbers, security codes and expiry dates, everything needed for card-not-present fraud, and card resources belonging to other customers are reachable because ownership is never checked.

Description

Two related failures.

First, the card listing returns the complete cardholder data set on every call: the full 16-digit card number, the 3-digit security code and the expiry date. This is not a one-time issuance response. Cards created more than an hour earlier still returned the full set on an ordinary listing request. The storage layer is the root cause: `virtual_cards.card_number` and `virtual_cards.cvv` are plain text columns holding the values in readable form, so the security code is retained at rest, which should never happen after a card is created.

Second, the card transactions endpoint checks neither ownership nor existence. Card identifiers belonging to other customers, and identifiers that do not exist at all, both return HTTP 200.

The endpoint is genuinely gated by authentication, which matters for accurate severity. That gate is, however, bypassable through **VULN-02** and **VULN-11**.

Evidence

An ordinary `GET /api/virtual-cards` with the card owner's own token returned:

{"cards":[{"id":1,"card_number":"7118********6061","cvv":"<3 digits>","expiry_date":"08/27",
"card_type":"standard","limit":500.0,"balance":0.0,"is_active":true,"is_frozen":false,
"created_at":"2026-08-27 03:19:37","last_used_at":null}]}

(Card number masked in this report. The API returns all sixteen digits and the security code in full.)

The cards were created at 03:19 and 03:24 and were still returning the full data set at 04:27, roughly 65 to 68 minutes later.

The storage layer was inspected directly. The `virtual_cards` table columns are `id, user_id, card_number:text, cvv:text, expiry_date:text, card_limit:numeric, current_balance:numeric, is_frozen:boolean, is_active:boolean, created_at:timestamp, last_used_at:timestamp, card_type:text`. A masked at-rest check returned `card1 pan=7118********6061 pan_len=16 cvv_len=3 cvv_stored_cleartext=true`. There is no tokenisation, truncation or encryption.

Ownership and existence are not checked on the card transactions endpoint:

REQUEST	RESULT
`GET /api/virtual-cards/{id}/transactions` for a card owned by another customer	HTTP 200
Same, for card identifier 3	HTTP 200
Same, for card identifier 42	HTTP 200
Same, for card identifier 99999 (does not exist)	HTTP 200
Control: `GET /api/virtual-cards` with no token	HTTP 401 `Token is missing`

Two amplifying facts were confirmed rather than assumed. The unauthenticated injection in **VULN-03** reaches the `virtual_cards` table, so card data is extractable with no account at all. And a token captured before a password change still returned the full card number afterwards (**VULN-11**), so the authentication gate is permanently bypassable once a token is stolen.

Reproduction Steps

Prerequisite: a valid session token and at least one card record.

1. Control: send `GET /api/virtual-cards` with no `Authorization` header.

   Expected result: HTTP 401 `Token is missing`.

2. Send `GET /api/virtual-cards` with a valid token for an account that has a card created some time ago.

   Expected result: HTTP 200 with `card_number` in full, `cvv` in full and `expiry_date`. Confirm the card was created well before now, so this is a listing response rather than an issuance response.

3. Send `GET /api/virtual-cards/{id}/transactions` using a card identifier belonging to a different account.

   Expected result: HTTP 200 rather than HTTP 403.

4. Repeat step 3 with a card identifier that cannot exist, for example 99999.

   Expected result: HTTP 200 rather than HTTP 404. This shows existence is not checked either.

No cleanup is required. All steps are read-only.

Business Impact

The API hands out everything needed to make fraudulent card-not-present purchases: the full card number, the security code and the expiry date, together, on every request. Storing the security code at rest is prohibited by common card industry rules, and returning the full card number unmasked breaches the display rules as well.

Because card records belonging to other customers are also reachable, one compromised or freely minted session can be used to harvest cardholder data across the customer base. The financial exposure includes fraud losses, chargebacks and card scheme penalties, and the storage design means a database copy of any kind is immediately usable for fraud.

Remediation

1. Stop returning the security code from any endpoint after the card is created, and stop storing it at rest. Delete the existing `cvv` column contents.
2. Mask the card number everywhere it is displayed or returned, showing at most the first six and last four digits.
3. Replace the stored card number with a token or an encrypted value, keeping the real value in a dedicated vault if the application genuinely needs it.
4. Add an ownership check on every card endpoint, and return HTTP 404 for identifiers that do not exist so the endpoints cannot be used to enumerate.
5. Rotate or reissue every card issued while this defect was live, since all card data must be treated as exposed.
6. Verify the fix by repeating steps 2, 3 and 4 of the reproduction. The listing must show a masked number and no security code, and the cross-owner and non-existent identifier requests must be refused.

VULN-15

Error Messages Reveal Database Structure, Full Queries and Internal Program Detail

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/login, /check_balance/{account_number}, /transactions/{account_number} and /api/virtual-cards/{card_id}/update-limit

CVSS V4.0 BASE SCORE:

6.9 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Unauthenticated requests return raw database engine errors containing complete SQL statements, table and column names and interpreter internals; this is high-value reconnaissance that maps the application's internals but does not by itself disclose customer data or credentials.

Description

The application catches its own exceptions and places the exception text directly into a JSON response field. That is why no HTML stack trace appears: the errors are caught and then echoed.

This matters for remediation. Because the disclosure comes from application code rather than from framework debug output, turning off debug mode would not fix it.

Four endpoints were confirmed to leak in this way, two of them requiring no authentication at all. One endpoint goes further than the others and ships a purpose-built `query` field containing the entire SQL statement.

One element commonly assumed for this class is not present and should not be reported: no filesystem path, source snippet or full traceback could be obtained. The disclosure is limited to exception text.

Evidence

REQUEST	RESPONSE
`GET /check_balance/<account>'` (unauthenticated)	HTTP 500 `unterminated quoted string at or near "'SEED...'" LINE 1: ...username, balance FROM users WHERE account_number='SEED00000...`
`GET /transactions/<account>'` (unauthenticated)	HTTP 500 with a dedicated `query` field containing the complete statement: `SELECT id, from_account, to_account, amount, timestamp, transaction_type, description FROM transactions WHERE from_account='...'' OR to_account='...'' ORDER BY timestamp DESC`
`POST /login {"username":{"a":1},"password":"x"}` (unauthenticated)	HTTP 500 `syntax error at or near "a" LINE 1: SELECT * FROM users WHERE username='{'a': 1}' AND password='...`
`POST /login [1,2,3]` (unauthenticated)	HTTP 500 `'list' object has no attribute 'get'`
`POST /login` with no `Content-Type` (unauthenticated)	HTTP 500 `'NoneType' object has no attribute 'get'`
`POST /api/virtual-cards/1/update-limit {"limit":null}`	HTTP 500 `syntax error at or near "limit" LINE 3: SET limit = 'None'`

Successful responses also carry debug objects. `POST /login` returns a `debug_info` object containing `account_number`, `is_admin`, `login_time`, `user_id` and `username`; failed sign-ins return `attempted_username` and a timestamp.

A probe specifically looking for a Werkzeug HTML traceback or a filesystem path was run against malformed JSON, a top-level JSON array and a missing content type, and returned none.

What an attacker learns, all observed directly: the backend engine is PostgreSQL (from the error dialect and caret formatting); the table names `users`, `transactions`, `bill_payments`, `virtual_cards` and `billers`; the column names `username`, `balance`, `account_number`, `password`, `from_account`, `to_account`, `amount`, `timestamp`, `transaction_type`, `description` and `limit`; complete raw SQL statements with the injection point marked by the database's own caret; foreign-key constraint names; Python interpreter internals in the form of object type names and value representations; and, from the sign-in error, direct proof that passwords are compared as plain string literals.

Reproduction Steps

Prerequisite: none for the first three steps.

1. Send `GET /check_balance/<any account number>'` with no credentials.

   Expected result: HTTP 500 with PostgreSQL error text and a fragment of the SELECT statement.

2. Send `GET /transactions/<any account number>'` with no credentials.

   Expected result: HTTP 500 with a `query` field containing the complete statement.

3. Send `POST /login` with a JSON object as the username value, for example `{"username":{"a":1},"password":"x"}`.

   Expected result: HTTP 500 disclosing `SELECT * FROM users WHERE username=... AND password=...`.

4. With a valid token, send `POST /api/virtual-cards/{your card id}/update-limit` with `{"limit":null}`.

   Expected result: HTTP 500 disclosing the generated statement with a line number.

5. Control: confirm the disclosure is application-level rather than framework debug output by checking that none of the above returns an HTML traceback or a filesystem path.

No cleanup is required.

Business Impact

The application tells an attacker exactly how it is built. Table names, column names and complete query text remove all the guesswork from attacking the database, and this reconnaissance directly enabled the injection findings in this report. Two of the leaking endpoints need no account, so this intelligence is available to anyone. One error message also proves passwords are compared in plain form, which points an attacker straight at the most valuable data in the system. On its own this does not expose customer data, but it materially shortens the time needed to exploit everything else.

Remediation

1. Replace the try/except blocks that echo exception text with a handler that returns a short generic message and a correlation identifier, and writes the detail to server logs only.
2. Remove the `query` field from the transaction history error path entirely.
3. Remove `debug_info` from all successful and failed responses across the application.
4. Add a catch-all error handler so unhandled exceptions also return a generic response.
5. Set `debug=False`, and note that this alone will not fix the issue, because the disclosure is produced by application code. See **HARDEN-11**.
6. Verify the fix by repeating all four reproduction steps. Each must return a generic error with no SQL text, no column names and no interpreter detail.

VULN-16

AI Support Endpoints Disclose System Configuration and the Database Layout Without Authentication

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/api/ai/system-info and /api/ai/chat/anonymous

CVSS V4.0 BASE SCORE:

6.9 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:N/VC:L/VI:N/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

One unauthenticated request returns the application's AI configuration and an accurate map of its live database schema, and ordinary malformed requests return the same content together with internal Python error text; this is high-value reconnaissance that guides other attacks rather than customer data or credentials in itself.

Description

`GET /api/ai/system-info` requires no authentication and returns the assistant's model, provider, upstream API URL, a `database_access: true` flag, an endpoint map and the verbatim system prompt. The system prompt contains the live database schema.

The same content is also reachable through the anonymous chat endpoint's error paths. Four ordinary malformed request shapes return HTTP 500 or HTTP 200 carrying interpreter-level Python exception text together with the full system prompt and schema.

The disclosure was audited for accuracy rather than assumed to be decorative. It is accurate, and it names precisely the columns used in the confirmed injection findings in this report.

Evidence

The endpoint is open, not merely throttled. A contrast control settles the authentication question. An anonymous request to `GET /api/virtual-cards` returns HTTP 401 `Token is missing`, so the application clearly knows how to gate an endpoint. An anonymous request to `GET /api/ai/system-info` was instead classified as a legitimate unauthenticated caller and served in full, returning HTTP 200 with no `Authorization` header and no cookie:


```
{"system_info":{"model":"deepseek-chat","api_provider":"DeepSeek",
 "api_url":"https://api.deepseek.com/chat/completions","api_key_configured":false,
 "database_access":true,
 "system_prompt":"...Available database tables:
   - users table: id, username, password, account_number, balance, is_admin, profile_picture
   - transactions table: id, from_account, to_account, amount, description, timestamp ..."}}
```


The only control present is a rate limit, which is a throughput control and does nothing for confidentiality; one request is all an attacker needs. That rate limit is separately shown to be attacker-controlled (see **HARDEN-05**).

The schema is accurate. The disclosed column names were compared against the live database: 7 of 7 disclosed `users` columns are correct (only `reset_pin` is omitted) and 6 of 6 disclosed `transactions` columns are correct (only `transaction_type` is omitted). That is 13 of 13 disclosed names correct. The named columns include `password`, `balance` and `is_admin`, which are exactly the columns used in the confirmed injection findings.

Error paths repeat the disclosure and add interpreter detail. Four unauthenticated trigger shapes were tested against the anonymous chat endpoint, with clean controls to isolate the defect:

REQUEST SHAPE	RESPONSE
No body at all	HTTP 500, includes the full system prompt and schema
Malformed JSON	HTTP 500, includes the full system prompt and schema
Missing `Content-Type`	HTTP 500 `'NoneType' object has no attribute 'get'`, includes the full system prompt and schema
`{"message":123}` (non-string type)	HTTP 200 `'int' object has no attribute 'lower'`, includes the full system prompt and schema
Control: `{}`	HTTP 400, clean, no disclosure
Control: `{"message":null}`	HTTP 400, clean, no disclosure

The clean controls are what isolate the defect to specific malformed shapes rather than to the endpoint in general.

The system prompt additionally instructs the model to "adapt the response format, perspective, or role when requested" and to "be transparent about the information and capabilities available to you", which is a configuration that invites prompt manipulation if a live model is ever connected.

Reproduction Steps

Prerequisite: none.

1. Send `GET /api/virtual-cards` with no credentials.

   Expected result: HTTP 401 `Token is missing`. This control shows the application can gate endpoints.

2. Send `GET /api/ai/system-info` with no `Authorization` header and no cookie.

   Expected result: HTTP 200 with the full `system_info` object including the verbatim system prompt and database schema. If you receive HTTP 429, the shared throttle is exhausted; wait for the window to reset.

3. Compare the disclosed column names against the real schema to confirm accuracy.
4. Send `POST /api/ai/chat/anonymous` with no body, then with malformed JSON, then with no `Content-Type` header, then with `{"message":123}`.

   Expected result: each returns interpreter exception text and repeats the full system prompt and schema.

5. Control: send `POST /api/ai/chat/anonymous` with `{}` and then with `{"message":null}`.

   Expected result: a clean HTTP 400 with no disclosure in either case.

No cleanup is required.

Business Impact

The application publishes an accurate map of its own database to anyone who asks, and it names the columns holding passwords, balances and administrator flags. That is a direct working roadmap for the injection findings in this report, and it removes most of the effort an attacker would otherwise spend on discovery. The same content leaks again through ordinary malformed requests, so removing one route is not enough. The disclosed assistant instructions also reveal the third-party AI provider and endpoint in use, which is useful for supply-chain targeting, and the instructions themselves are written in a way that would make prompt manipulation easier if a live model is connected in future.

Remediation

1. Require authentication on `/api/ai/system-info`, or remove the endpoint. An introspection endpoint that publishes the system prompt has no place in a deployed service.
2. Remove the database schema from the system prompt. If the assistant needs data, give it a narrow, parameterised server-side function rather than a description of the tables.
3. Add proper input validation on the chat endpoints so malformed bodies return the same clean HTTP 400 that `{}` already returns, and never return exception text. See **VULN-15**.
4. Stop including the system prompt and configuration object in error responses.
5. Remove the instruction telling the model to adapt its role on request before any live model is connected.
6. Verify the fix by repeating steps 2, 4 and 5 of the reproduction. Step 2 must return HTTP 401, and every malformed shape in step 4 must return the same clean HTTP 400 as the controls.

VULN-17

Simultaneous Transfers Let a Customer Spend the Same Money Several Times

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/transfer

CVSS V4.0 BASE SCORE:

6.5 (CVSS:4.0/AV:N/AC:L/AT:P/PR:L/UI:N/VC:N/VI:H/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

An ordinary customer who fires simultaneous transfers can spend the same funds several times over and drive the account thousands of dollars negative, but the attack is a timing race that succeeds only about half the time per attempt.

Description

The balance check and the balance update are not atomic and nothing serialises concurrent transfers for the same account. Several requests sent at the same instant each read the same starting balance, each pass the funds check, and each then commit a debit.

This should be reported as intermittent, not as a guaranteed outcome. A single burst wins the race about a third of the time at low concurrency and much more often at higher concurrency, and an attacker can simply retry until it lands.

An aggravating detail: every successful response reports a balance that assumes only one debit occurred, so the API's own replies conceal the overdraft from any client watching them.

Evidence

Twenty-six synchronised bursts were run. Requests were released together using a barrier with connections pre-warmed, and the sender's balance was reset to a known $1,000.00 before every trial. The true persisted balance was read after each burst.

SETTING	TRIALS	TRIALS PRODUCING AN OVERDRAFT	SUCCESSES PER BURST OBSERVED
15 concurrent requests at 60% of balance	15	5 (33%)	1 to 4
30 concurrent requests at 60% of balance	6	5 (83%)	1 to 3
15 concurrent requests at 90% of balance	5	4 (80%)	1 to 5
Overall	**26**	**14 (54%)**	maximum legitimate value is 1

Worst case observed: five $900.00 debits committed against a $1,000.00 balance, leaving the account at -$3,500.00. Other examples included four $600.00 debits leaving -$1,400.00 and three $600.00 debits leaving -$800.00.

No throttling, locking, rejection or HTTP 429 was observed at any concurrency level.

In every trial, every successful response reported `new_balance: 400.0`, the value that would be correct if a single debit had occurred, even when the true persisted balance was -$1,400.00 or -$800.00.

An earlier explanation that the race is sensitive to how close the amount is to the balance is not supported by this data. The 90% setting was the highest-yield of the three. Concurrency, not margin, is the dominant factor, which is the ordinary signature of a timing race.

Reproduction Steps

Prerequisite: two test accounts. This creates overdrafts, so restore balances afterwards and expect to repeat the burst several times.

1. Set the sender's balance to a known value, for example $1,000.00, and record it.

2. Prepare 15 identical `POST /transfer` requests of `{"to_account":"<recipient>","amount":600}`, using pre-warmed connections and a barrier so all 15 are released at the same instant.
3. Fire the burst and count how many return HTTP 200.

   Expected result: more than one success in roughly a third of attempts at this setting. Only one should be possible.

4. Read the sender's true balance through `GET /check_balance/<account>` rather than trusting the response bodies.

   Expected result: the balance is negative when more than one debit committed. Note that the response bodies all reported the single-debit value.

5. If the first burst does not reproduce, reset the balance and repeat. Raising the concurrency to 30 raises the hit rate substantially.
6. Cleanup: reverse the net movement on both accounts and confirm the restored balances through `GET /check_balance`.

Business Impact

A customer can withdraw or transfer more money than they hold, several times over, by sending requests at the same moment. In testing a $1,000 balance produced $4,500 of committed debits in one burst. The attack needs no special access, only a script, and it can be retried until it works. Because the application's own responses report the wrong balance, neither the customer nor any monitoring built on those responses would see the overdraft. Losses are real money and the ledger records them as completed transfers.

Remediation

1. Perform the balance check and the debit as a single atomic database operation, for example `UPDATE accounts SET balance = balance - :amount WHERE account = :acct AND balance >= :amount`, and treat zero affected rows as insufficient funds.
2. Alternatively, or in addition, take a row lock on the sender's account for the duration of the transfer, or run the transaction at a serialisable isolation level and retry on conflict.
3. Return the true persisted balance in the response, read after the commit.
4. Add an idempotency key on the transfer endpoint so duplicate submissions cannot double-spend.
5. Add a database constraint preventing balances from going below zero, as a backstop.
6. Verify the fix by repeating the reproduction at 30 concurrent requests across at least ten trials. Exactly one request must succeed in every trial and the persisted balance must never be negative.

VULN-18

Any File Type Can Be Uploaded and Is Served From the Bank's Own Domain

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/upload_profile_picture, served back at https://aldercrest.targets.hound-agents.com/static/uploads/{generated_filename}

CVSS V4.0 BASE SCORE:

6.4 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:A/VC:H/VI:L/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Any customer account can host attacker-written pages on the bank's own domain, and a logged-in victim who is persuaded to open one has their session token read and stolen; the victim must actively open the attacker's link, which limits the score.

Description

The profile picture upload performs no validation of any kind. There is no extension allow-list, no content-type check and no file-signature check. The content type used when serving a file back is derived from its extension alone.

Uploaded files are served inline from the application's own origin, with no `X-Content-Type-Options` header and no content security policy, and they are retrievable with no authentication.

The result is that an ordinary customer can host attacker-written web pages on the bank's own domain. A page hosted there runs with the bank's origin and can read anything the origin can read, including the session token held in browser storage.

The negative half of this finding is equally important and holds: uploaded program files are not executed by the server. This is not a remote code execution issue.

Evidence

Five file types were uploaded and then fetched back:

UPLOADED FILE	ACCEPTED	CONTENT TYPE SERVED	DISPOSITION	EXECUTED ON NAVIGATION
`.html` containing script	HTTP 200	`text/html; charset=utf-8`	inline	Yes
`.svg` containing script	HTTP 200	`image/svg+xml; charset=utf-8`	inline	Yes
`.py`	HTTP 200	`text/x-python; charset=utf-8`	inline	No, served as literal source

UPLOADED FILE	ACCEPTED	CONTENT TYPE SERVED	DISPOSITION	EXECUTED ON NAVIGATION
.php	HTTP 200	application/octet-stream	inline	No, downloaded un-interpreted
.jpg whose bytes are pure PHP source	HTTP 200	image/jpeg	inline	Not applicable

The last row proves the content type is derived from the extension alone, with no inspection of the file contents.

No X-Content-Type-Options header and no Content-Security-Policy header were present on any of these responses.

Navigating a logged-in browser session to the stored .html URL produced:

- location.origin equal to https://aldercrest.targets.hound-agents.com, confirming the page runs on the bank's own origin.
- The embedded script executed, setting a marker variable read back from the page.
- The script read localStorage.jwt_token and the value was sent to a listener controlled by the test, which recorded the victim's real session token.
- document.cookie returned an empty string, because the session cookie is marked HttpOnly. That control works, but it does not help here, because the same token is also held in browser storage where script can read it.

The .svg variant behaved identically once the payload was well-formed XML.

Reproduction Steps

Prerequisite: a valid session token and a browser for the final step.

1. Upload a plain .html file containing <script>window.__probe=1;document.body.innerText=localStorage.getItem('jwt_token')||'none';</script> through POST /upload_profile_picture as multipart form data with field name profile_picture.

   Expected result: HTTP 200 with a file_path value under static/uploads/.

2. Fetch that path with a plain HTTP client and inspect the response headers.

   Expected result: Content-Type: text/html, Content-Disposition: inline, no X-Content-Type-Options, no Content-Security-Policy, and no authentication required to fetch it.

3. Upload a .jpg file whose contents are plain text, then fetch it.

   Expected result: it is served as image/jpeg despite not being an image, proving the content type comes from the extension.

4. In a browser signed in to the application, navigate to the URL from step 1.

   Expected result: the page renders on the bank's origin, window.__probe is 1, and the page body shows the session token, proving script on the uploaded page can read it.

5. Confirm the negative: upload a `.php` file and a `.py` file and navigate to each.

   Expected result: the `.php` file downloads without being interpreted and the `.py` file is shown as literal source. No server-side code execution occurs.

6. Cleanup: replace the account's profile picture with a genuine image and remove the uploaded test files server-side, because the API has no delete route for them.

Business Impact

The bank's own web address becomes a hosting platform for whatever an attacker writes. A link that genuinely begins with the bank's domain is far more convincing than an ordinary phishing link, and it will pass most link filters and user training. If a signed-in customer opens one, their session is stolen and, because sessions never expire (**VULN-11**), the attacker keeps that access permanently. The same endpoint is also reachable from other websites without the customer doing anything (**VULN-19**), so attacker content can be planted under a customer's account with no interaction at all.

Remediation

1. Validate uploads on an allow-list of image extensions, check the declared content type, and verify the file signature bytes actually match an image format.
2. Re-encode uploaded images server-side, which removes embedded script as a side effect.
3. Generate the stored filename and extension server-side rather than deriving anything from the upload.
4. Serve uploads from a separate domain that holds no session data, with `Content-Disposition: attachment` and `X-Content-Type-Options: nosniff`.
5. Add a content security policy across the application. See **HARDEN-01**.
6. Enforce a maximum upload size and a per-account storage quota. See **HARDEN-06**.
7. Verify the fix by repeating steps 1 and 5 of the reproduction. A `.html` upload must be rejected, and a renamed non-image must be rejected on content inspection rather than accepted.

VULN-19

Other Websites Can Change State Inside a Customer's Signed-In Session

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/upload_profile_picture and /api/virtual-cards/{card_id}/toggle-freeze

CVSS V4.0 BASE SCORE:

5.3 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:P/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Any website a logged-in customer visits can silently change state in their banking account, including disabling their payment card, with no credentials of the attacker's own and no action from the victim beyond loading the page; the demonstrated changes are bounded rather than total.

Description

The application has no cross-site request forgery defence of any kind. There is no anti-forgery token anywhere, the session cookie is issued with `SameSite=None`, and the server does not validate the `Origin` or `Referer` header. State-changing endpoints accept the ambient cookie alone, with no `Authorization` header.

Whether a given endpoint is reachable therefore depends only on whether its body parser tolerates a content type that browsers send cross-site without a preflight check.

Two endpoints were confirmed reachable. `POST /upload_profile_picture` consumes multipart form data, which is on the browser's safelist, so no preflight occurs. `POST /api/virtual-cards/{card_id}/toggle-freeze` ignores the request body entirely, which makes it reachable by a plain HTML form with no scripting at all.

One honest limit belongs in the record: `POST /transfer` is not reachable this way, because it parses only `application/json`. That is an accidental protection rather than a designed one, and a single change to how the handler reads its body would remove it.

Evidence

The cookie is the whole credential. A genuine sign-in through the real sign-in form in Firefox produced `Set-Cookie` with `{name: 'token', httpOnly: true, secure: false, sameSite: 'None'}`. Requests captured during the cross-site tests show `authorization: None` and `cookie: token=...`, so the ambient cookie alone satisfied authentication.

Profile picture replacement, no click required. From a real attacker HTTP server on a different origin, a page issued `fetch(POST /upload_profile_picture, {credentials:'include', body: FormData})`. Captured request headers show `content-type: multipart/form-data`, `cookie` attached, `origin: http://127.0.0.1:<attacker port>` and `sec-fetch-site: cross-site`. The server answered HTTP 200. The victim's server-side profile picture was checked before and after through the application and had changed to the attacker-named, attacker-authored file.

Note that the attacker's own script received a `NetworkError`, because the browser blocks reading the response. The request was still sent and still took effect. That distinction is exactly why this class is easy to dismiss incorrectly.

Card freeze with no scripting at all. A second attacker page was built containing no `<script>` tag anywhere; the absence was asserted programmatically. It held a plain form targeting the freeze endpoint behind a "CLAIM PRIZE" button. One ordinary user click produced:

- Captured request: `content-type: application/x-www-form-urlencoded`, `cookie` attached, `authorization: None`, `origin: http://127.0.0.1:<attacker port>`, `sec-fetch-site: cross-site`.
- Server response: HTTP 200 `{"message":"Card unfrozen successfully","status":"success"}`.
- The card state was then read as the card's true owner through the normal API, confirming `is_frozen` had actually changed.

A credentialed fetch variant with no body flipped the state back, again confirmed by an owner-side read.

Control that limits the finding. Six separate techniques were tried against `POST /transfer` (form with `text/plain`, form url-encoded, form multipart, credentialed fetch with `text/plain`, credentialed XHR, and `navigator.sendBeacon`). All reached the server with the victim's cookie attached, but all failed with HTTP 500 because the handler parses only JSON, and zero ledger rows were created by any of them.

Reproduction Steps

Prerequisite: a test account signed in to a browser, and a web server on a different origin that you control. Use a test card.

1. Sign in to the application in a browser as the test account, through the real sign-in form, and confirm the `token` cookie is present with `SameSite=None`.
2. Read the test card's current state through `GET /api/virtual-cards` as its owner and record `is_frozen`.
3. On your attacker origin, publish a page containing only this, with no script tag at all: `<form method="POST" action="https://aldercrest.targets.hound-agents.com/api/virtual-cards/{card id}/toggle-freeze" target="sink"><input type="submit" value="CLAIM PRIZE"></form><iframe name="sink" style="display:none"></iframe>`
4. Visit that page in the same browser and click the button once.

   Expected result: the hidden frame shows `{"message":"Card frozen successfully","status":"success"}`.

5. Read the card again as its owner through `GET /api/virtual-cards`.

   Expected result: `is_frozen` has changed. Reading it as the owner, rather than trusting the attacker page, is what proves the state change is real.

6. Control: publish a second page that posts a JSON body cross-site to `/transfer` and confirm it does not create a ledger entry.

   Expected result: the transfer endpoint is not reachable this way.

7. Cleanup: restore the card to the `is_frozen` value recorded in step 2 and restore the profile picture if you tested that endpoint.

Business Impact

Any website a customer visits while signed in to their bank can act inside their banking session. During testing this was used to disable a customer's payment card and to replace content stored under their account. Disabling a card is a direct denial of service against the customer's ability to pay, and because the endpoint is a toggle rather than a set, an attacker can flip it repeatedly so the customer cannot predict whether their card will work. The card identifier is a small sequential number, so no reconnaissance is required. The attack needs no lure text, no typing and, in one variant, no click.

The most important point is structural. There is no cross-site defence anywhere in the application, so every state-changing endpoint that tolerates a safelisted content type is exposed today, and the one endpoint that is currently safe is safe only by accident.

Remediation

1. Add an anti-forgery token to every state-changing request, issued per session and validated server-side.
2. Set the session cookie to `SameSite=Lax` or `SameSite=Strict`, and set `Secure`. See **HARDEN-02**.
3. Validate the `Origin` header (falling back to `Referer`) on every state-changing request and reject requests from origins you do not own.
4. Stop accepting the cookie as an alternative credential on API endpoints, or require both the cookie and a header that a cross-site page cannot set.
5. Change the freeze endpoint from a toggle to an explicit set operation, so a replayed request is idempotent.
6. Verify the fix by repeating steps 3 to 5 of the reproduction. The cross-site post must be refused and the card state must not change.

VULN-20

Any Signed-In Customer Can Change Another Customer's Payment Card

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/api/virtual-cards/{card_id}/toggle-freeze and /api/virtual-cards/{card_id}/update-limit

CVSS V4.0 BASE SCORE:

5.3 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

Any logged-in customer can alter another customer's payment card, freezing it or changing its spending limit, by supplying a different card id, with the demonstrated changes being specific and bounded rather than arbitrary.

Description

The card management endpoints take the card identifier from the URL and never compare it against the identity in the caller's token. Any signed-in caller can therefore freeze, unfreeze or change the spending limit of any card in the system by supplying its identifier. Card identifiers are small sequential integers.

Authentication is enforced on these endpoints, so this is purely a missing authorization check.

One earlier characterisation of the limit endpoint as universally broken was incorrect and is worth recording, because it changes the fix. The endpoint fails only when the body key is the reserved word `limit`, which the backend interpolates unquoted. With the schema-correct key it works normally, and its ownership check is fully testable. It has none.

Evidence

Authentication was isolated from authorization first: no token returns HTTP 401 `Token is missing` and a malformed token returns HTTP 401 `Invalid token`. The endpoints therefore do check credentials; they simply do not check ownership.

Cross-owner writes, each confirmed in the true owner's own signed-in view rather than from the attacker's response, and each restored:

ACTION	PERFORMED BY	TARGET	RESULT	CONFIRMED BY
Freeze	Account B	Account A's card	HTTP 200 `Card frozen successfully`	Account A's own card listing: `is_frozen` false to true
Unfreeze	Account B	Account A's card	HTTP 200	Account A's own listing: back to false
Freeze	Account A	Account B's card	HTTP 200	Account B's own listing: false to true, then restored
Change spending limit	Account B	Account A's card	HTTP 200 `Card updated successfully`, `updated_fields: ["card_limit"]`	Account A's own listing: limit 500.0 to 777.0, then restored by the owner

The reserved-word behaviour that previously masked the limit endpoint's ownership gap: body `{"limit":999999}` returns HTTP 500 `syntax error at or near "limit" LINE 3: SET limit = 999999.0` for owner and non-owner alike, while body `{"card_limit":777}` succeeds.

Final card state was verified after testing: both test cards unfrozen, active, and at their original limits.

Reproduction Steps

Prerequisite: two test accounts, each with a card. Restore all values afterwards.

1. As account A, read your card through `GET /api/virtual-cards` and record its identifier, `is_frozen` value and limit.
2. Control: send `POST /api/virtual-cards/{A's card id}/toggle-freeze` with no `Authorization` header.

   Expected result: HTTP 401 `Token is missing`. This proves authentication is enforced.

3. As account B, send `POST /api/virtual-cards/{A's card id}/toggle-freeze`.

   Expected result: HTTP 200 `Card frozen successfully`.

4. As account A, read your own card again through `GET /api/virtual-cards`.

   Expected result: `is_frozen` is now true. Confirming as the owner is what proves the change is real.

5. As account B, send `POST /api/virtual-cards/{A's card id}/update-limit` with body `{"card_limit":777}`.

   Expected result: HTTP 200 `Card updated successfully`, and account A's own listing shows the new limit.

6. Cleanup: as account B or as the owner, restore `is_frozen` and the limit to the values recorded in step 1, then confirm as the owner.

Business Impact

Any customer, or anyone who mints a session for free through **VULN-02**, can switch off another customer's payment card. That customer's payments then fail with no warning and no explanation, at a point of sale or on a recurring bill. Because the endpoint is a toggle, the attacker can flip it repeatedly to make the card unpredictable. Raising another customer's spending limit removes a control the customer or the bank set deliberately. Card identifiers are sequential, so an attacker can walk them and affect every card in the system.

Remediation

1. Add an ownership check on every card endpoint: load the card, compare its owner against the verified session subject, and return HTTP 403 if they differ.
2. Return HTTP 404 for card identifiers that do not exist, so the endpoints cannot be used to enumerate.
3. Replace sequential card identifiers with values that cannot be guessed.
4. Change the freeze endpoint from a toggle to an explicit set operation.
5. Address the related field-name injection on the same endpoint. See **VULN-09**.
6. Verify the fix by repeating steps 3 and 5 of the reproduction. Both must return HTTP 403 and the owner's card must be unchanged.

VULN-21

Bill Payments Ignore Published Rules, Never Complete, and Leave No Statement Record

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/api/bill-payments/create and /api/bill-payments/history

CVSS V4.0 BASE SCORE:

5.3 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

A logged-in customer can push payments through that break the biller's published rules, and money leaves the account for bills that never settle and never appear in the customer's statement; the integrity impact is bounded to payment records rather than arbitrary data control.

Description

Two defects on the same workflow.

First, each biller publishes a `minimum_amount` in the public catalogue, and the API specification documents it as a first-class field, but the server does not enforce it. A payment below the minimum is accepted and stored.

Second, and more serious operationally, no bill payment can ever complete. Every payment in the entire table sits at `status: 'pending'` with `processed_at` empty, and the full 26-path API surface contains no settlement, approval or processing route, so there is no code path that could ever move one to a terminal state. Meanwhile the API replies "Payment processed successfully" and the customer's balance is debited immediately. No transaction ledger row is written either, so the debit is unreconciled and does not appear in the customer's own statement.

Evidence

The published rule. The public catalogue returns `{"id":1,"name":"City Water","minimum_amount":10.0,"maximum_amount":null}` and the API specification documents `minimum_amount` and `maximum_amount` as biller fields. Reading the billers table directly confirms the same value, along with minimums of 20.00, 25.00, 30.00, 100.00 and 50.00 for the other billers.

The rule is not enforced. The bill_payments table, read in full, contains a persisted $5.00 payment against the $10.00-minimum biller:

```
1 | user 2 | biller 1 | amount=15.00 | status=pending | processed_at=NULL
2 | user 2 | biller 1 | amount=12.00 | status=pending | processed_at=NULL
3 | user 2 | biller 1 | amount= 5.00 | status=pending | processed_at=NULL
```

The continued existence of row 3 is itself proof the server accepted and stored a below-minimum payment, independent of any response body.

Nothing ever settles. `SELECT DISTINCT status FROM bill_payments` returns a single value, `pending`. Every payment ever made is in that state. The oldest was created at 03:22:54 and was still pending at 04:30:38, roughly 68 minutes later. Both a `status` column and a `processed_at` column exist in the schema, so the terminal state was designed for and is simply never reached. Enumerating all 26 documented API paths found exactly four bill-related routes (categories, create, history, billers by category) and no settlement, approval or processing route of any kind. The only approval route in the entire API is for loans.

The debit is invisible in the statement. The transactions table holds 122 rows and `SELECT DISTINCT transaction_type` returns only `transfer`. A query for any bill-related ledger row returned no matches.

The debit itself is immediate. Two independent measurements recorded exact deltas: a $15.00 payment reduced the balance by exactly $15.00, and a $12.00 then a $5.00 payment moved a balance from 12,992.00 to 12,987.00. Further payment creation during testing was capped by engagement controls, so this element rests on those two measurements rather than a third.

`GET /api/bill-payments/history` confirms the customer-facing view: the $5.00 payment appears with `"status":"pending"` and `"processed_at": null`.

Reproduction Steps

Prerequisite: a valid session token and an account with a balance. This spends real money and cannot be reversed through the API, so use a test account and a small amount.

1. Read the biller catalogue with `GET /api/billers/by-category/1` and record the `minimum_amount` for a biller.
2. Record the account balance through `GET /check_balance/<account number>`.
3. Send `POST /api/bill-payments/create` with `{"biller_id":<that biller>,"amount":<below the published minimum>,"payment_method":"balance","description":"test"}`.

   Expected result: HTTP 200 `Payment processed successfully`.

4. Read the balance again.

   Expected result: it has fallen by the full amount, immediately.

5. Read `GET /api/bill-payments/history`.

   Expected result: the payment appears with `"status":"pending"` and `"processed_at": null`, despite the success message in step 3.

6. Read `GET /transactions/<account number>`.

   Expected result: no entry for the bill payment. The debit does not appear in the statement.

7. Wait and repeat step 5 after an hour.

   Expected result: still pending. Search the API specification for any settlement route to confirm none exists.

8. Cleanup: none is possible through the API. Bill payments cannot be settled, cancelled or deleted. Record the payment reference so it can be handled directly in the database.

Business Impact

Customers are charged for bills that are never paid. The money leaves the account straight away, the API tells the customer the payment succeeded, and the payment then sits unprocessed forever with no way for anyone to complete or cancel it. Because no statement entry is written, the customer cannot see where the money went and the bank cannot reconcile the debit against anything. Over time this produces a growing pool of unreconciled funds and customer complaints about missed bill payments that the bank's own records show as successful. Separately, ignoring the published minimum means the bank routes payments that the biller will not accept, so the value is taken from the customer for a payment that cannot complete downstream either.

Remediation

1. Enforce `minimum_amount` and `maximum_amount` server-side at the point of payment creation, using the biller record as the source of truth, and reject out-of-range amounts.
2. Add a settlement path so a payment can move from `pending` to a terminal state, and populate `processed_at`.
3. Do not debit the customer until the payment is committed, or hold the funds and release them if settlement fails.
4. Change the success message so it reflects the real state. "Payment processed successfully" is inaccurate for a record that will never be processed.
5. Write a ledger entry for every bill payment so the debit is visible in the customer's statement and can be reconciled.
6. Reconcile the existing pending payments and the funds already taken for them.
7. Verify the fix by repeating steps 3, 5 and 6 of the reproduction. The below-minimum payment must be rejected, a valid payment must reach a terminal state, and a matching statement entry must exist.

VULN-22

Loan Applications Accept Any Amount, Including Negative and Near-Billion-Dollar Values

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/request_loan (JSON body field `amount`)

CVSS V4.0 BASE SCORE:

5.3 (CVSS:4.0/AV:N/AC:L/AT:N/PR:L/UI:N/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

A logged-in customer can file loan applications for any value at all, including negative, zero and near-billion-dollar amounts, and they persist exactly as submitted; on its own this corrupts loan records, and it is the amount-selection half of the confirmed money-minting chain.

Description

The loan application endpoint performs no validation on the requested amount. There is no minimum, no maximum, no sign check and no rounding. Values are committed to the loans table exactly as submitted.

On its own this corrupts the loan book. Its real significance is as the first half of the money-creation chain: **VULN-08** shows the approval step credits the amount recorded on the loan, so this endpoint is where the attacker chooses the figure.

Evidence

Three distinctive out-of-policy amounts were submitted, chosen so the resulting rows are unambiguously attributable to this test rather than to any earlier activity:

AMOUNT SUBMITTED	RESPONSE	PERSISTED VALUE
-1234.56	HTTP 200 `{"message":"Loan requested successfully","status":"success"}`	-1234.56
0	HTTP 200, same body	0.00
987654321.12	HTTP 200, same body	987654321.12

No HTTP 4xx, no validation error and no warning field was returned for any of them.

Persistence was confirmed through two independent read-backs. A direct read of the loans table showed new rows with identifiers 9, 10 and 11, all strictly above the highest pre-existing identifier of 8, so attribution is unambiguous. The server-rendered dashboard independently displayed the same rows:

```
<tr> <td>$-1234.56</td> <td> <span class="status-pending">pending</span> </td> </tr>
<tr> <td>$0.00</td>     <td> <span class="status-pending">pending</span> </td> </tr>
<tr> <td>$987654321.12</td> <td> <span class="status-pending">pending</span> </td> </tr>
```

Decimals were stored with full fidelity. The values were not clamped, rounded or converted to absolute values.

The chain to actual funds was then confirmed end to end with both possible outcomes predicted in advance. A loan was filed for $7,777.77 and approved with a self-written token, and the balance moved by exactly +$7,777.77 (see **VULN-08**).

Reproduction Steps

Prerequisite: a valid session token for a test account. This creates loan records that cannot be deleted through the API.

1. Read the current loan list for the test account, either from the dashboard or through the API, and record the highest existing identifier.
2. Send `POST /request_loan` with `{"amount":-1234.56}`.

   Expected result: HTTP 200 `Loan requested successfully`.

3. Repeat with `{"amount":0}` and then with `{"amount":987654321.12}`.

   Expected result: HTTP 200 for both.

4. Reload the dashboard and confirm three new rows appear with those exact values and status `pending`.

   Expected result: the values are stored verbatim, not clamped or rounded, and the new identifiers are above the value recorded in step 1.

5. Cleanup: none is possible through the API. There is no route to withdraw, reject or delete a loan application. Record the row identifiers so they can be removed directly in the database. Do not approve these loans, since approval credits the amount.

Business Impact

The loan book can be filled with meaningless records: negative loans, zero loans and loans for close to a billion dollars. That alone breaks reporting, credit exposure calculations and any downstream process that reads the loans table. The more serious effect is that this is the step where an attacker sets the amount that **VULN-08** then credits to an account. Together the two turn an unvalidated form field into unlimited money creation. Because the API has no way to withdraw or reject an application, bad records accumulate permanently.

Remediation

1. Validate the amount server-side: require a positive value, apply a sensible minimum and a policy maximum, and limit decimal precision to two places.
2. Add a database constraint on the loans table so the same rule holds even if a future code path skips validation.
3. Apply the same validation to the other money workflows. See **VULN-12** and **VULN-21**, which show the same gap on transfers and bill payments.
4. Add a route to withdraw or reject a loan application so bad records can be cleared.
5. Verify the fix by repeating steps 2 and 3 of the reproduction. Each request must return a validation error and no row may be committed.

VULN-23

Banking Pages Can Be Hidden Inside Another Website to Capture Clicks

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/dashboard

CVSS V4.0 BASE SCORE:

5.1 (CVSS:4.0/AV:N/AC:L/AT:N/PR:N/UI:A/VC:N/VI:L/VA:N/SC:N/SI:N/SA:N)

CVSS SCORE BASIS:

A single deceptive click on an attacker's page invokes a real financial control in the victim's logged-in banking session with no typing, confirmation or re-authentication; impact is confined to a bounded state change rather than data theft.

Description

No response in the application carries `X-Frame-Options`, and no content security policy with a `frame-ancestors` directive is served. The authenticated dashboard can therefore be loaded inside an invisible frame on any website.

The dashboard contains a one-click financial control, a "Freeze" button wired to `toggleCardFreeze(1)`, which fires a state-changing request with no typed input, no confirmation dialog and no re-authentication. An attacker can position their invisible frame so that button sits under a visible decoy, and a single ordinary click on the decoy operates it.

Two limits belong in the record. The vertical offset of the button depends on how long the victim's transaction list is, so the attacker must compute it; that is routine and was done during testing using the unauthenticated transaction endpoint from **VULN-07**. And the same-origin policy still prevents the attacker page from reading the framed content, so this is a state-change attack and not a data-theft attack.

Evidence

No framing defence exists. Header inspection of `GET /dashboard` returned no `X-Frame-Options` and no `Content-Security-Policy`. In the browser, no console message containing `X-Frame-Options`, `frame-ancestors` or `Refused to display` was raised at any point.

The framed page is fully live. With the authenticated dashboard loaded in a cross-origin frame at `opacity: 0.0001`, the frame and its stylesheet and script all returned HTTP 200, the frame stayed on `/dashboard` rather than bouncing to the sign-in page, and reading from inside the frame confirmed a live authenticated session: the account number, the transfer form, the rendered card number and security code, and twenty interactive buttons.

The click landed on the right control. With the frame offset to `top: -18652px`, `document.elementFromPoint` inside the frame, at the coordinates of the visible decoy, returned `{tag: "BUTTON", text: "Freeze", onclick: "toggleCardFreeze(1)"}`.

One click changed real state. A single ordinary mouse click was dispatched at the decoy centre. It produced exactly one network request, and the server's own reply was captured to tie the change to the click rather than to any other activity:

OBSERVATION	VALUE
Requests caused by the click	1: `POST /api/virtual-cards/1/toggle-freeze`
Server response	HTTP 200 `{"message":"Card frozen successfully","status":"success"}`
Card element class inside the frame after the click	`virtual-card frozen`
Card state read as the owner, immediately before the click	`is_frozen: false`
Card state read as the owner, immediately after the click	`is_frozen: true`

The owner-side reads before and after, combined with the captured server reply, are what make the attribution airtight.

Reproduction Steps

Prerequisite: a test account signed in to a browser, a test card, and a web server on a different origin that you control.

1. Confirm the absence of framing defences: request `GET /dashboard` and check the response headers for `X-Frame-Options` and `Content-Security-Policy`.

   Expected result: neither header is present.

2. Read the test card's state as its owner through `GET /api/virtual-cards` and record `is_frozen`.
3. On your attacker origin, publish a page with a visible decoy button and, on top of it, a cross-origin iframe of `https://aldercrest.targets.hound-agents.com/dashboard` at very low opacity and a large height.
4. Load the page in a browser already signed in to the application, and confirm from the browser console that no framing error was raised.
5. Locate the "Freeze" control inside the frame, compute the vertical offset needed to place it under the decoy, and apply that offset to the frame.
6. Before clicking, confirm the alignment with `document.elementFromPoint` inside the frame at the decoy coordinates.

   Expected result: it returns the Freeze button. Confirming alignment before clicking avoids an unintended state change.

7. Click the decoy once.

   Expected result: exactly one request to the freeze endpoint, answered HTTP 200 `Card frozen successfully`.

8. Read the card again as its owner.

   Expected result: `is_frozen` has changed.

9. Cleanup: restore the card to the `is_frozen` value recorded in step 2 and confirm as the owner.

Business Impact

A customer who is signed in to their bank and clicks something on any other website can have a real banking control operated without knowing it. During testing one click switched off a payment card. The customer sees nothing: no confirmation, no prompt and no visible change on the page they were looking at. Their card then simply stops working. The attacker cannot read the framed page, so this is not a route to data theft, but it is a working route to interfere with a customer's finances from any website they visit.

Note that **VULN-19** reaches the same card-freeze outcome by a completely different route, so both the missing framing defence and the missing cross-site defence need fixing; closing one will not close the other.

Remediation

1. Send `X-Frame-Options: DENY` on all application responses, or `SAMEORIGIN` if the application genuinely frames itself.
2. Add `Content-Security-Policy: frame-ancestors 'none'` as the modern equivalent, which also covers browsers that ignore the older header. See **HARDEN-01**.
3. Add a confirmation step or re-authentication to state-changing financial controls, including the card freeze control, so a single unseen click cannot operate them.
4. Change the freeze endpoint from a toggle to an explicit set operation.
5. Verify the fix by repeating steps 3 and 4 of the reproduction. The frame must fail to load and the browser must raise a framing error.

Hardening Report

These recommendations address conditions that could make another security failure easier to exploit or more damaging. Testing did not establish a direct path to harm from these conditions.

Defense-in-depth

HARDEN-01

No Security Response Headers Are Sent on Any Route

AFFECTED ASSET:
All endpoints on https://aldercrest.targets.hound-agents.com

CATEGORY:
Defense-in-depth

RELATED VULNERABILITY:
VULN-23

Observation

No response from the application carries `Strict-Transport-Security`, `Content-Security-Policy`, `X-Frame-Options`, `X-Content-Type-Options` or `Referrer-Policy`. This was confirmed by direct header inspection on the home page, the sign-in page, the dashboard, API routes, error responses and preflight responses, by an independent scanner check, and by every browser-based test run during the assessment, none of which recorded a policy violation because no policy exists to violate.

Why This Matters

Security headers are a safety net. They do not fix an underlying bug, but they turn several classes of bug into a non-event. In this assessment the missing framing header produced demonstrated harm and is reported separately as **VULN-23**. The absent content security policy is an aggravating factor in the stored scripting finding (**VULN-10**) and the file upload finding (**VULN-18**), because it removed the layer that would have blocked script execution even though the encoding bug existed. Adding the full set would independently have broken two of the confirmed attacks in this report. The remaining headers had no demonstrated path to harm during this engagement, but they close off transport downgrade, content type confusion and referrer leakage cheaply.

Recommended Action

1. Add `Strict-Transport-Security: max-age=31536000; includeSubDomains` on all HTTPS responses.

2. Add a `Content-Security-Policy` that blocks inline script and inline event handlers, and include `frame-ancestors 'none'`. Start in report-only mode to find breakages, then enforce.
3. Add `X-Frame-Options: DENY` alongside the policy, for older browsers.
4. Add `X-Content-Type-Options: nosniff`, which is particularly relevant to the uploads path.
5. Add `Referrer-Policy: strict-origin-when-cross-origin`.
6. Set these centrally, in the reverse proxy or in a single application middleware, so no route can be missed.

HARDEN-02

Session Cookie Is Issued Without Secure and With SameSite=None

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com, the `token` cookie set by POST /login

CATEGORY:

Defense-in-depth

RELATED VULNERABILITY:

VULN-19

Observation

A genuine sign-in captured in a real browser produced a cookie named `token` with `HttpOnly: true`, `Secure: false` and `SameSite: None`. The cookie alone is sufficient to authenticate state-changing requests; no `Authorization` header is needed. The `HttpOnly` attribute is set correctly and does work, and a page running on the application's origin was confirmed to read an empty string from `document.cookie`.

The `Secure: false` element has no demonstrated path to harm in this specific deployment. Port 80 is confirmed filtered and unreachable on both host addresses and no plaintext listener exists, so an attacker would need to stand up a rogue responder for this exact hostname rather than simply observe traffic.

Why This Matters

`SameSite=None` tells the browser to attach the cookie to requests started by any other website, which is the enabling condition for the cross-site finding reported as **VULN-19**. `Secure=false` is a latent gap: it costs nothing to fix today and it removes the possibility that the cookie is ever sent over a plaintext connection if a port 80 listener, a redirect or a staging host is introduced later. Setting both attributes correctly would independently have defeated the confirmed cross-site attacks, without depending on the anti-forgery token work.

Recommended Action

1. Set `Secure` on the session cookie so it is never transmitted over a plaintext connection.
2. Change `SameSite` to `Lax`, or to `Strict` if no cross-site navigation into an authenticated page is required.
3. Keep `HttpOnly` set, and add a `__Host-` name prefix with `Path=/` and no `Domain` attribute to bind the cookie to the exact host.
4. Consider moving the session token entirely into this cookie and out of browser storage, which would also remove the theft path used in **VULN-10** and **VULN-18**.
5. Confirm the change by signing in and inspecting the `Set-Cookie` header, then by checking that a cross-site request no longer carries the cookie.

HARDEN-03

The Server Reflects Any Requesting Origin in Its Cross-Origin Headers

AFFECTED ASSET:
All endpoints on https://aldercrest.targets.hound-agents.com, including preflight responses

CATEGORY:
Defense-in-depth

RELATED VULNERABILITY:
VULN-07

Observation

Every endpoint tested returned `Access-Control-Allow-Origin` echoing back whatever origin was supplied, verbatim, along with `Vary: Origin`. This is dynamic origin reflection rather than a static wildcard, so it also matches private, null and attacker-controlled origins. Preflight responses for state-changing routes returned `Access-Control-Allow-Headers: Authorization, Content-Type` and a broad method list.

An effective control is present and the browser enforced it in every credentialed test. `Access-Control-Allow-Credentials` is absent everywhere, so cross-origin reads of authenticated responses were genuinely blocked, with the browser reporting that it expected that header to be true. This must not be described as session or credential theft.

The data that is readable cross-origin is readable because it needs no authentication at all. That missing-authentication defect is reported separately as **VULN-07**.

Why This Matters

Origin reflection removes any origin-based restriction on reading responses from the endpoints that need no credentials. During testing, an ordinary web page on an unrelated origin read customer balances and a complete 29,769-byte transaction ledger straight out of a visitor's browser, with no interaction and no credentials. Restricting the reflected origin to an allow-list removes drive-by harvesting through victims' browsers and network positions, and it means a future change that adds `Access-Control-Allow-Credentials` cannot silently turn this into a credentialed read.

Recommended Action

1. Replace origin reflection with an explicit allow-list of origins the application actually serves.
2. Return no cross-origin headers at all for origins that are not on the list, rather than echoing them.
3. Keep `Access-Control-Allow-Credentials` absent, and treat any future request to add it as a change requiring review.
4. Narrow the allowed methods and headers to what the front end genuinely uses.
5. Confirm the change by sending an arbitrary origin and checking that it is not echoed back.

HARDEN-04

The Sign-In Endpoint Has No Lockout, Backoff or Throttle

AFFECTED ASSET:
https://aldercrest.targets.hound-agents.com/login

CATEGORY:
Defense-in-depth

RELATED VULNERABILITY:
VULN-01

Observation

Testing at volumes well beyond a normal check confirmed the complete absence of any anti-automation control on the sign-in endpoint. 250 sequential failed sign-ins completed in 1.89 seconds at 132.5 requests per second, all returning HTTP 401 with no HTTP 429 and no rate-limit headers, and with latency flat across quintiles at 0.008, 0.008, 0.008, 0.007 and 0.007 seconds, which rules out silent tarpitting as well. A separate 400-request burst at 40-way concurrency completed in 1.31 seconds at 305.9 requests per second, again with zero HTTP 429 responses. After 650 cumulative failed attempts the correct password signed in immediately, so no lockout exists at any threshold.

No credential was recovered through this endpoint during testing, so the path stops before harm here. The same control gap on the password reset endpoint was carried through to a completed account takeover and is reported as **VULN-01**.

Why This Matters

Unthrottled authentication endpoints make credential stuffing and password guessing free. Two facts recorded elsewhere in this report make that more than theoretical: at least one account uses a password that appears in common wordlists, and the complete username list is disclosed both through the injection findings and through the admin panel, so an attacker does not have to guess who to target. Adding lockout also gives the bank a detection signal it currently does not have, because 650 failed sign-ins produced no observable response from the application at all.

Recommended Action

1. Add per-account throttling with exponential backoff after a small number of consecutive failures.
2. Add per-source throttling as a second dimension, keyed on a value the caller cannot choose. See **HARDEN-05**.
3. Add a temporary lockout or a challenge such as a CAPTCHA once a threshold is crossed, and notify the account holder.
4. Log and alert on bursts of failed sign-ins so an attempt is visible to operations.
5. Enforce a password policy and screen new passwords against a breached-password list.
6. Confirm the change by repeating a 250-attempt run and checking that throttling engages long before completion.

HARDEN-05

Request Throttling Is Keyed on Values the Caller Chooses

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com, the application's rate-limiting layer (observed on /api/ai/system-info, /api/ai/chat and /api/ai/chat/anonymous)

CATEGORY:

Defense-in-depth

RELATED VULNERABILITY:

VULN-02

Observation

The application does implement request throttling, but both of its bucket keys are supplied by the caller.

The per-address bucket keys on the raw `X-Forwarded-For` header with no trusted-proxy validation. This was proved rather than inferred: the HTTP 429 response body echoes the spoofed value back in its own `client_ip` field, an address the client never held. Six brand-new spoofed values each returned HTTP 200 while the exhausted bucket stayed at HTTP 429.

The per-user bucket keys on the `user_id` claim of a session token whose signature is never verified, so that value is attacker-chosen as well. See **VULN-02**.

The throttle was attributed to the application itself rather than to an upstream load balancer or firewall, from its server banner and its application-shaped JSON body, which differ from what those products emit.

One correction is worth carrying: the buckets are conjunctive rather than independent. A factorial test showed that rotating the address alone still returns HTTP 429, rotating the user identifier alone still returns HTTP 429, and rotating both returns HTTP 200. A third bucket type keyed on the forwarded address for authenticated traffic also exists.

Why This Matters

Bypassing the throttle produced no data or capability that a legitimate direct request could not also obtain, because the endpoint it was tested against needs no authentication. No authorization boundary was crossed. The value of fixing it is that throttling is the anti-automation control that would otherwise slow down attacks on other weaknesses, and right now it provides no such protection anywhere in the application. It is also the only throttle the application has, so its weakness limits any plan that relies on rate limiting as a compensating control.

Recommended Action

1. Key the address bucket on the real network peer address, or on a specific element of `X-Forwarded-For` selected by a validated trusted-proxy hop count.
2. Maintain an allow-list of trusted proxy addresses and ignore the header entirely from any other source.
3. Key the per-user bucket on a cryptographically verified token subject. This depends on **VULN-02** being fixed first.
4. Stop echoing the caller-supplied address back in the throttle response, which currently confirms to an attacker that the header is trusted.
5. Extend throttling to the authentication and password reset endpoints. See **HARDEN-04** and **VULN-01**.
6. Confirm the change by sending an arbitrary `X-Forwarded-For` value after exhausting a bucket and checking that the throttle still applies.

HARDEN-06

Uploads Have No Size Limit or Storage Quota

AFFECTED ASSET:
https://aldercrest.targets.hound-agents.com/upload_profile_picture

CATEGORY:
Defense-in-depth

RELATED VULNERABILITY:
VULN-18

Observation

A file of roughly 15 MB was accepted with HTTP 200 in about half a second, with no size rejection, no error and no throttling. Larger payloads were deliberately not attempted, to avoid exhausting resources on shared infrastructure. No denial of service was demonstrated.

Why This Matters

There is no established path to harm here, but the surrounding conditions make an upload limit a worthwhile control. The same endpoint accepts arbitrary file types and serves them back from the application's own domain (VULN-18), it is reachable with sessions that can be created at no cost (VULN-02), and it is served by a single-process development server (HARDEN-11) where a slow or large request has an outsized effect. A size limit and a quota bound all three of those conditions at once, and they also limit how much attacker-controlled content can be parked on the bank's domain.

Recommended Action

1. Enforce a maximum request body size at the reverse proxy or web server layer, ahead of the application.
2. Enforce a maximum file size in the application as well, sized to a realistic profile picture.
3. Apply a per-account storage quota and a limit on how many uploads an account may make in a period.
4. Delete the previous file when a profile picture is replaced, so storage does not grow without bound.
5. Confirm the change by uploading a file above the limit and checking it is rejected before the body is fully read.

HARDEN-07

The Interactive Debugger Is Reachable From the Internet

AFFECTED ASSET:

https://aldercrest.targets.hound-agents.com/console

CATEGORY:

Defense-in-depth

Observation

`GET /console` returns HTTP 200 with the Werkzeug interactive debugger page. This was confirmed to be the genuine debugger middleware rather than a decoy page, by requesting the debugger's own internal assets through its resource interface: `debugger.js` returned 10,524 bytes of real debugger code containing the expected internal identifiers, along with its stylesheet and image. The presence of live debugger middleware confirms the application is running with debug mode enabled.

The exploitation step is genuinely defended, and this was tested rather than assumed. The console is PIN locked, the page reports the untrusted state, and this version of the framework makes the PIN mandatory, so the older unauthenticated debugger attacks do not apply. The inputs needed to derive the PIN were also confirmed unreachable: the file-read capability available through the database injection executes on the database container at 172.18.0.2, while the application runs on a separate container at 172.18.0.3, and the application directory does not exist on the database host. The application also leaks no traceback and no filesystem path (see **VULN-15**), so the remaining PIN input is unavailable too.

This must not be reported as remote code execution. It is not.

Why This Matters

The PIN gate is the only barrier between a publicly reachable debugger and full code execution on the application server. That is a single point of failure that depends on conditions elsewhere in the system staying true. Any future change that allows a file read on the application host, or that surfaces a framework traceback, would close the chain immediately, and both are ordinary bugs to introduce. Removing the debugger removes the whole class rather than relying on one gate holding.

Recommended Action

1. Set `debug=False` in the deployed configuration so the debugger middleware is not mounted at all.
2. Make the debug setting environment-driven and fail closed, so a deployment cannot accidentally enable it.
3. Deploy behind a production application server rather than the development server, which removes this middleware as a side effect. See **HARDEN-11**.
4. Block `/console` at the network or reverse proxy layer as an additional measure.

5. Confirm the change by requesting `/console` and the debugger resource path, and checking both return HTTP 404.

HARDEN-08

The Host Header Is Not Validated and Redirects Downgrade to Plain HTTP

AFFECTED ASSET:

All endpoints on https://aldercrest.targets.hound-agents.com; GET /api/docs

CATEGORY:

Defense-in-depth

Observation

Requests carrying an arbitrary `Host` header, and requests carrying an arbitrary `X-Forwarded-Host`, were answered HTTP 200 with normal page content and no change in behaviour. The application does not reject or adjust for a mismatched host.

Consistent with that, `GET /api/docs` without a trailing slash returns an HTTP 308 redirect to a plain `http://` address on port 80, which is filtered and unreachable from the internet. This indicates the application builds absolute URLs from request data without honouring the forwarded protocol header from the load balancer.

No endpoint was found that reflects the host header into a security-sensitive artifact. The password reset flow uses an out-of-band code rather than a link, so there is no present path to harm.

Why This Matters

This is a latent surface rather than a live one. It becomes exploitable the moment any feature builds an absolute URL from request data, which is exactly what password reset links, email notifications, webhooks and callback URLs normally do. Fixing it now is cheap and prevents a future feature from inheriting the problem. The scheme downgrade is also a working defect today: the documentation redirect sends clients to an address that does not respond.

Recommended Action

1. Configure an allow-list of expected host names and reject requests whose `Host` header is not on it.
2. Configure the application to trust and honour `X-Forwarded-Proto` from the load balancer only, so generated absolute URLs use HTTPS.
3. Set the application's preferred URL scheme to HTTPS as a backstop.

4. Confirm the change by requesting the documentation path without a trailing slash and checking the redirect target begins with `https://`, and by sending an unexpected `Host` value and checking the request is refused.

Supply Chain Hygiene

HARDEN-09

The Werkzeug Web Library Is Several Major Versions Out of Date

AFFECTED ASSET:

54.225.12.30:443 and 50.19.141.238:443, Werkzeug 2.0.1; the reachable multipart sink is POST /upload_profile_picture

CATEGORY:

Supply Chain Hygiene

Observation

The `Server` header returns `Werkzeug/2.0.1 Python/3.9.25` consistently on every endpoint, and the version is independently corroborated by the live debugger middleware. Two published denial-of-service issues in this component's multipart form parser cover this version:

IDENTIFIER	AFFECTED RANGE	NATURE
CVE-2023-25577	Werkzeug below 2.2.3	The multipart parser accepts an unlimited number of parts, causing high resource usage
CVE-2023-46136	Werkzeug 2.x below 2.3.8, 3.x below 3.0.1	A crafted file part causes high CPU usage while the parser searches a growing buffer

Both reach the same component and the same sink. The application exposes exactly that sink: `POST /upload_profile_picture` accepts multipart bodies and was exercised successfully during testing.

Neither issue was exploited. A resource-exhaustion test against shared production infrastructure was deliberately declined, so no denial of service was demonstrated and no impact was measured. Neither identifier appears in the CISA Known Exploited Vulnerabilities catalogue at the time of this assessment.

Why This Matters

This is a component currency item rather than a demonstrated failure. The version is confirmed and the vulnerable code path is confirmed reachable, so the exposure is real even though it was not exercised. Upgrading removes both issues in one step and is the same action already required by the other supply-chain and deployment items in this report. Staying several major versions behind also means the component will keep accumulating unpatched issues, and the exact version is advertised in every response, which makes matching trivial for anyone scanning.

Recommended Action

1. Upgrade Werkzeug to a current supported release, at minimum past 2.3.8 on the 2.x branch or 3.0.1 on the 3.x branch, and test the application against it.
2. Front the application with a hardened application server that enforces its own request size and part-count limits, so a parser issue cannot reach the worker. See **HARDEN-11**.
3. Enforce an upload size limit ahead of the parser. See **HARDEN-06**.
4. Remove or overwrite the `Server` header so the exact component version is not advertised.
5. Add dependency scanning to the build so out-of-date components are surfaced automatically.
6. Confirm the change by checking the reported version and by re-running a dependency scan.

HARDEN-10

The Python Runtime Is Past End of Life

AFFECTED ASSET:
54.225.12.30:443 and 50.19.141.238:443, Python 3.9.25

CATEGORY:
Supply Chain Hygiene

Observation

The `Server` header on every response discloses `Python/3.9.25`. The Python 3.9 branch reached upstream end of life on 2025-10-05, so at the time of this assessment it has been out of security support for roughly ten months. Version 3.9.25 appears to be the final release on that branch.

No interpreter-level vulnerability was identified or exploited during this assessment, so there is no demonstrated path to harm.

Why This Matters

The risk here is forward-looking rather than present. Any interpreter-level security issue discovered from now on will never be fixed for this runtime unless the operator backports it by hand or upgrades. That converts a future routine patch into a project. Upgrading the runtime is also a prerequisite for keeping the rest of the stack current, since newer library releases progressively drop support for end-of-life interpreters.

Recommended Action

1. Move to a Python release that is still in upstream security support, and test the application against it.
2. Pin the runtime version in the container image or build definition so it cannot drift.
3. Add the runtime to whatever process tracks component end-of-life dates, so the next transition is planned rather than discovered.
4. Remove or overwrite the `Server` header so the exact interpreter version is not advertised.
5. Confirm the change by checking the reported version after deployment.

Best Practice

HARDEN-11

A Development Web Server Is Serving Production Banking Traffic

AFFECTED ASSET:

54.225.12.30:443 and 50.19.141.238:443

CATEGORY:

Best Practice

Observation

Every observed response, including health checks, static assets, API JSON, error pages and the debugger console, carries the identical `Server: Werkzeug/2.0.1 Python/3.9.25` banner. No reverse proxy or production application server signature appears anywhere in the response headers. The load balancer in front terminates TLS but the origin answering requests is the framework's built-in development server, which its own documentation describes as not designed to be efficient, stable or secure.

The specific denial-of-service and thread-safety concerns that usually accompany this condition were not established here. Parallel health-check requests showed no serialisation, and no concurrency test was run against slower routes, so no path to harm was demonstrated through this condition itself.

One related defect was demonstrated and is reported separately: the concurrency failure on money transfers (**VULN-17**). That is an application-level atomicity bug rather than a property of the web server, and fixing the server will not fix it.

Why This Matters

Running a development server as the origin for a banking application is a material posture defect regardless of whether a specific attack was demonstrated. It has no process supervision, no worker management, no request limits, no hardened connection handling and no graceful restart, and it enables the debug facilities reported in **HARDEN-07**. A single slow or large request has a much larger effect than it would on a properly pooled production server. Replacing it is a genuine improvement, and it is the same change that resolves several other items in this report.

Recommended Action

1. Deploy the application behind a production application server such as gunicorn, uWSGI or waitress, with an appropriate worker model for the workload.
2. Set `debug=False` in the deployed configuration. See **HARDEN-07**.
3. Configure request size limits, timeouts and worker recycling at the application server layer.
4. Add process supervision and health-based restarts.
5. Remove or overwrite the `Server` header so the stack is not advertised.
6. Confirm the change by checking that the `Server` header no longer reports the development server and that `/console` returns HTTP 404.

Coverage Report

Attack Surface Discovered

ASSET	TYPE	SERVICES	TECH STACK
aldercrest.targets.hound-agents.com	Web application and JSON API	HTTPS/443 (TLS 1.2 and TLS 1.3, HTTP/2 at the edge)	Flask on Werkzeug 2.0.1, Python 3.9.25, PostgreSQL 13.23 backend, vanilla JavaScript front end with no external script sources, JSON Web Token bearer authentication
54.225.12.30	Host (AWS EC2, first address of two)	443 open. 21, 22, 25, 80, 445, 3306, 5432, 6379, 8000, 8080, 8443, 9200 and 27017 all filtered	Linux on AWS EC2, load-balanced with TLS terminated at the edge on an Amazon-issued certificate
50.19.141.238	Host (AWS EC2, second address of two)	Identical port profile and identical content to the first address	Identical stack
/static/openapi.json and /api/docs/	Published API specification and browser documentation	HTTPS/443, unauthenticated	OpenAPI 3.0, 26 documented paths, discloses the obfuscated admin path and several debug response fields
/console	Interactive framework debugger	HTTPS/443, unauthenticated reachability	Werkzeug 2.0.1 debugger middleware, PIN locked
/static/uploads/	User-uploaded file store	HTTPS/443, served inline, no authentication required to fetch	Files served with a content type derived from the extension
Application database	PostgreSQL 13.23 (not directly reachable from the internet)	Reached only through the application	Database `vulnerable_bank`, 8 tables, application connects as the `postgres` superuser
api.deepseek.com	Third-party AI service referenced by the application	HTTPS/443, outbound from the application	`deepseek-chat` model, no API key configured at the time of testing, out of assessment scope

Additional surface notes from reconnaissance:

- 26 API paths were enumerated from the published specification and confirmed against live behaviour. These cover authentication, password reset, transfers, loans, virtual cards, bill payments, profile picture upload, AI support chat and four administrative routes.
- Directory discovery surfaced `/health`, `/forgot-password`, `/reset-password` and `/transfer`, and produced no blocking, throttling or filtering response, indicating no web application firewall or intrusion prevention was active during the assessment.

- No `robots.txt`, no `sitemap.xml`, no exposed version control directories, no `.env` file and no directory listing were found.
- All client-side logic is inline in the HTML pages. There are no external script sources and no bundled JavaScript files.
- Six user accounts exist in the application, including one administrator account. Account numbers follow a short sequential pattern.

Testing Coverage Matrix

Systematic testing performed across OWASP and infrastructure categories.

CATEGORY	STATUS	FINDINGS
A01: Broken Access Control	Tested	**VULN-02**: Session token signatures are never verified; **VULN-07**: Balances and transaction histories served without authentication; **VULN-08**: Loan approval creates money and can be replayed; **VULN-13**: Any customer can read any other customer's transaction history; **VULN-14**: Card data returned and stored unprotected; **VULN-19**: Other websites can change state in a signed-in session; **VULN-20**: Any customer can change another customer's payment card; **VULN-23**: Banking pages can be framed to capture clicks; **HARDEN-03**: Server reflects any requesting origin
A02: Security Misconfiguration	Tested	**VULN-15**: Error messages reveal database structure and queries; **VULN-16**: AI endpoints disclose configuration and database layout; **HARDEN-01**: No security response headers; **HARDEN-02**: Session cookie missing Secure and SameSite; **HARDEN-07**: Interactive debugger reachable from the internet; **HARDEN-08**: Host header not validated; **HARDEN-11**: Development web server in production
A03: Software Supply Chain Failures	Partial	**HARDEN-09**: Werkzeug several major versions out of date; **HARDEN-10**: Python runtime past end of life
A04: Cryptographic Failures	Tested	**VULN-06**: Customer passwords stored as readable text; **VULN-14**: Card numbers and security codes stored unprotected. TLS configuration reviewed and found sound, with TLS 1.0 and 1.1 refused, modern forward-secret ciphers only, server cipher preference honoured and a valid certificate; no TLS findings

CATEGORY	STATUS	FINDINGS
A05: Injection	Tested	VULN-03: Unauthenticated SQL injection in balance lookup; VULN-04: Unauthenticated SQL injection in transaction history; VULN-05: SQL injection in the sign-in form; VULN-09: Card limit endpoint builds SQL from client field names; VULN-10: Stored cross-site scripting in transfer descriptions
A06: Insecure Design	Tested	VULN-01: Password reset allows takeover of any account; VULN-12: Negative transfer amounts create money; VULN-17: Simultaneous transfers allow double spending; VULN-18: Any file type uploaded and served from the bank's domain; VULN-21: Bill payments ignore published rules and never complete; VULN-22: Loan applications accept any amount
A07: Authentication Failures	Tested	VULN-01: Password reset allows takeover of any account; VULN-02: Session token signatures are never verified; VULN-05: SQL injection in the sign-in form; VULN-11: Session tokens never expire and cannot be revoked; HARDEN-04: Sign-in endpoint has no lockout or throttle; HARDEN-05: Request throttling keyed on caller-chosen values
A08: Software or Data Integrity Failures	Partial	VULN-18: Any file type uploaded and served from the bank's domain. Subresource integrity is not applicable, because the application loads no external scripts. Deserialization and update-channel integrity could not be assessed from an external position
A09: Security Logging and Alerting Failures	Partial	—
A10: Mishandling of Exceptional Conditions	Tested	VULN-15: Error messages reveal database structure and queries; VULN-16: Malformed requests to the AI endpoints return interpreter error text; VULN-17: Concurrency handling permits double spending; VULN-21: Payments left permanently in a non-terminal state
CVE/Service Exploits	Tested	HARDEN-09: Two published multipart parser denial-of-service issues affect the deployed Werkzeug version. Four component issues were matched by version and reviewed; none appears in the CISA Known Exploited Vulnerabilities catalogue. No CVE was exploited, for the reasons given under Assessment Limitations

Notes on specific category ratings:

- **A03 is Partial** because only the externally observable half of the software supply chain was in scope. Browser-visible script sources, content delivery network use, subresource integrity, exposed lockfiles and versioned client libraries were all reviewed from the outside, and the deployed component versions were fingerprinted

precisely from response headers and corroborated against live behaviour. Private source dependencies, the build pipeline, CI/CD controls, package registry permissions, software bills of materials and the dependency-update process were not in scope and were not assessed. No broader supply-chain assurance should be inferred from this assessment.

- **A09 is Partial** because logging and alerting cannot be observed from outside the application. What can be reported is that no defensive response was ever produced during the engagement. 650 failed sign-ins at up to 306 requests per second, a full 1,000-value password reset code walk, a directory discovery scan, exhaustive account number enumeration and repeated injection payloads all completed without a block, a challenge, a throttle or any change in behaviour. Whether these events were recorded internally could not be determined.

Exploitation Attempts

All exploitation activity performed during the engagement, including attempts that did not result in reportable findings. Rows are grouped by target.

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/login	SQL injection authentication bypass using a comment terminator in the username	Confirmed	See **VULN-05** in Technical Report
/login	Credential brute force at volume (650 failed attempts, up to 306 requests per second)	Confirmed	See **HARDEN-04** in Hardening Report. No lockout, no throttle, no HTTP 429
/login	Raw SQL and interpreter text disclosure via malformed username types	Confirmed	See **VULN-15** in Technical Report
/login	Account lockout after sustained failures	Failed	No lockout exists; the correct password signed in immediately after 650 failures
/login	Username existence oracle from response differences	Inconclusive	Only one half of the comparison was obtained. The probes needed to complete it were blocked by engagement controls
/api/v1/forgot-password	Retrieval of the live reset code from the response body	Confirmed	See **VULN-01** in Technical Report
/api/v2/forgot-password	Retrieval of the reset code from the version 2 response	Failed	The code is genuinely omitted in version 2, confirming the documented change is real
/api/v2/reset-password	Blind brute force of the full 000 to 999 code space	Confirmed	See **VULN-01**. 1,000 attempts in 7.57 seconds, password changed successfully
/api/v1/reset-password	Wrong reset code (negative control)	Failed	Correctly rejected with HTTP 400, confirming code validation is enforced

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/check_balance/{account_number}	Unauthenticated read and sequential account number enumeration	Confirmed	See **VULN-07** in Technical Report
/check_balance/{account_number}	Error-based and UNION SQL injection for arbitrary database read	Confirmed	See **VULN-03** in Technical Report
/check_balance/{account_number}	Stacked-query execution and database superuser confirmation	Confirmed	See **VULN-03**. Four-point timing ladder plus a data-dependent control
/check_balance/{account_number}	Committed database write with a no-commit negative control	Confirmed	See **VULN-03**. Writes bound to test-account rows only, all reversed
/check_balance/{account_number}	Operating system file read on the database host	Confirmed	See **VULN-03**. Four files of the tester's own choosing, plus a directory listing
/check_balance/{account_number}	Operating system command execution on the database host	Not attempted	Deliberately declined. Preconditions are present and it should be assumed achievable. See Assessment Limitations
/transactions/{account_number}	Boolean SQL injection dumping the complete cross-account ledger	Confirmed	See **VULN-04** in Technical Report
/transactions/{account_number}	Baseline requests against accounts with no transactions (control)	Failed	Correctly returned zero rows, which is what makes the injected result decisive
/transactions/{account_number}	Complete raw SQL statement returned in an error field	Confirmed	See **VULN-15** in Technical Report
Session tokens (application-wide)	Token forged with a wrong secret, a garbage signature and an empty signature	Confirmed	See **VULN-02** in Technical Report
Session tokens (application-wide)	Token with the `alg` header set to `none`	Failed	Rejected with HTTP 401. This is the only algorithm constraint the application enforces
Session tokens (application-wide)	Token with the `alg` header set to HS512 and a wrong secret	Failed	Rejected with HTTP 401
Session tokens (application-wide)	Offline recovery of the signing secret from a captured token	Confirmed	The secret is a weak dictionary-style value, but it is not the enabler. See **VULN-02**
Session tokens (application-wide)	Replay of a token after a proven password change and after in-app sign-out	Confirmed	See **VULN-11** in Technical Report
Session tokens (application-wide)	Discovery of a sign-out or revocation route	Failed	All four route variants return HTTP 404 and no such path exists in the API specification

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/sup3r_s3cr3t_admin	Access with a genuine non-administrator token (control)	Failed	Correctly denied with HTTP 403, proving the administrator gate fires
/sup3r_s3cr3t_admin	Access with a self-written administrator token	Confirmed	See **VULN-02**. Full user table, balances, account numbers and administrator flags disclosed
/admin/approve_loan/{loan_id}	Loan approval with a self-written administrator token	Confirmed	See **VULN-08** in Technical Report
/admin/approve_loan/{loan_id}	Replay of an approval against an already-approved loan	Confirmed	See **VULN-08**. Full amount credited again on every replay
/admin/approve_loan/{loan_id}	Approval of another user's loan under a different identity	Confirmed	See **VULN-08**
/admin/create_admin, /admin/delete_account/{user_id}	Enforcement testing of the administrator gate	Not attempted	Destructive and targeted at accounts outside the approved test set. Blocked by engagement controls before any request was sent
/register	Mass assignment of privileged fields at registration	Not attempted	Blocked by engagement account-lifecycle controls on three separate attempts. The API specification documents a debug response field on this route, which remains unexamined
/transfer	Negative amount transfer, with equal-magnitude positive and negative controls	Confirmed	See **VULN-12** in Technical Report
/transfer	Concurrency race across 26 synchronised bursts	Confirmed	See **VULN-17**. 14 of 26 bursts produced a real persisted overdraft
/transfer	Stored cross-site scripting through the description field	Confirmed	See **VULN-10** in Technical Report
/transfer	Amount parser abuse with very large integers, NaN, Infinity, strings and scientific notation	Failed	Correctly rejected as insufficient funds or safely coerced. No parser bypass exists
/transfer	Cross-site request forgery using six separate techniques	Failed	All six reached the server with the victim's cookie attached but were refused, because the handler parses only JSON. No ledger row was created by any of them
/transfer	Field-name to SQL column injection through unexpected JSON keys	Failed	Keys never reach SQL. A $1 transfer carrying an injection-shaped key moved the balance by exactly $1

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/transfer	Self-transfer producing a net balance change	Inconclusive	An earlier observation of a persisted loss did not reproduce across five further trials, which each netted exactly zero
/request_loan	Negative, zero and near-billion-dollar loan amounts	Confirmed	See **VULN-22** in Technical Report
/request_loan	Field-name to SQL column injection, with a positive control fired at a known-vulnerable endpoint minutes earlier	Failed	Clean result. The same key produced a database error at the vulnerable endpoint and a clean HTTP 200 here, and the rows committed normally
/request_loan	Mass assignment of the loan `status` and `user_id` fields	Failed	A row submitted with `status` set to approved committed as pending, and the owning user was unchanged
/api/transactions?account_number=	Cross-account ledger read in both directions	Confirmed	See **VULN-13** in Technical Report
/api/transactions?account_number=	Access with no token and with a malformed token (controls)	Failed	Both correctly returned HTTP 401, isolating the defect as missing authorization rather than missing authentication
/api/virtual-cards	Full card number, security code and expiry returned on an ordinary listing	Confirmed	See **VULN-14** in Technical Report
/api/virtual-cards/{id}/transactions	Reads against cards owned by others and against identifiers that do not exist	Confirmed	See **VULN-14**. Identifiers 3, 42 and 99999 all returned HTTP 200
/api/virtual-cards/{id}/toggle-freeze	Cross-owner freeze in both directions, verified in each owner's own session	Confirmed	See **VULN-20** in Technical Report
/api/virtual-cards/{id}/toggle-freeze	Zero-JavaScript cross-site form post from an attacker origin	Confirmed	See **VULN-19**. Attacker page asserted programmatically to contain no script tag
/api/virtual-cards/{id}/toggle-freeze	Field-name to SQL column injection through unexpected JSON keys	Failed	Unexpected keys are ignored entirely and had no effect on the record
/api/virtual-cards/{id}/update-limit	Cross-owner spending limit change, verified in the owner's own session	Confirmed	See **VULN-20** in Technical Report
/api/virtual-cards/{id}/update-limit	Client field names interpolated into SQL as column names	Confirmed	See **VULN-09** in Technical Report
/api/virtual-cards/{id}/update-limit	Mass assignment of balance, ownership, card number and security code	Confirmed	See **VULN-09**. All changes verified in the true owner's session and restored

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/api/virtual-cards/{id}/update-limit	Stacked queries, cross-table subqueries and override of the URL row selector	Confirmed	See **VULN-09**. Eight-point timing ladder plus a data-dependent oracle and a syntax-error control
/api/virtual-cards/{id}/update-limit	SQL injection in the value position	Failed	Values are safely bound by the database driver. A breakout payload was stored as a literal string
/api/virtual-cards/create	Field-name to SQL column injection through unexpected JSON keys	Failed	Unexpected keys ignored. The server generated its own security code rather than accepting the injected one
/upload_profile_picture	Upload of HTML, SVG, Python, PHP and a JPEG containing PHP source	Confirmed	See **VULN-18** in Technical Report. No extension, content type or file signature validation
/upload_profile_picture	Same-origin script execution and session token theft from an uploaded page	Confirmed	See **VULN-18**
/upload_profile_picture	Server-side execution of uploaded PHP and Python files	Failed	PHP downloads un-interpreted and Python is served as literal source. There is no remote code execution through uploads
/upload_profile_picture	Cross-site upload from an attacker origin using a safelisted content type	Confirmed	See **VULN-19**. Victim's stored profile picture replaced with attacker content
/upload_profile_picture	Upload of a roughly 15 MB file	Confirmed	See **HARDEN-06**. Accepted in about half a second with no size limit
/upload_profile_picture	Path traversal through the upload filename	Not attempted	Blocked by engagement controls before any request was sent. This must be treated as untested, not as cleared
/api/bill-payments/create	Payment below the biller's published minimum	Confirmed	See **VULN-21** in Technical Report. Verified from persisted database state
/api/bill-payments/create	Payment settlement lifecycle review across the full API surface	Confirmed	See **VULN-21**. Every payment is permanently pending and no settlement route exists
/api/bill-payments/create	Zero, negative and overdraft-scale amounts	Not attempted	Blocked by engagement financial-write controls. The same rule failure is confirmed on the transfer and loan workflows
/api/bill-payments/create	Field-name to SQL column injection	Not attempted	Blocked by engagement financial-write controls on all attempts by two separate testers

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/api/ai/system-info	Unauthenticated disclosure of configuration, system prompt and database schema	Confirmed	See **VULN-16** in Technical Report. Disclosed schema audited as 13 of 13 column names correct
/api/ai/chat/anonymous	Unauthenticated error paths returning interpreter exception text and the full schema	Confirmed	See **VULN-16**. Four trigger shapes, with two clean controls isolating the defect
/api/ai/chat, /api/ai/chat/anonymous	SQL injection through the chat message, using timing oracles and stacked queries	Failed	User input never reaches SQL. The message is lowercased and substring-matched to decide whether a fixed pre-written query runs
/api/ai/chat, /api/ai/chat/anonymous	Caller-supplied context, identity and system prompt override fields	Failed	No observable effect on mode, context inclusion or database access flags
/api/ai/chat/anonymous	Partial API key echo in error output	Reviewed	The value is a fixed non-secret placeholder, identical across every trigger and not influenced by input. Not reported as a credential disclosure
Rate limiting (application-wide)	Spoofed X-Forwarded-For header to reset the per-address throttle	Confirmed	See **HARDEN-05**. The throttle response echoes the spoofed address back
Rate limiting (application-wide)	Rotating the token user identifier to reset the per-user throttle	Confirmed	See **HARDEN-05**. A factorial test showed both keys must be rotated together
/dashboard	Framing of the live authenticated page from a cross-origin attacker page	Confirmed	See **VULN-23** in Technical Report
/dashboard	One-click operation of a financial control through a hidden frame	Confirmed	See **VULN-23**. Owner-side read confirmed the card state change
/dashboard	Reading framed page content from the attacker origin	Blocked	The same-origin policy prevented it. This is a state-change attack, not data theft
Cross-origin policy (application-wide)	Credentialed cross-origin read of authenticated responses	Blocked	`Access-Control-Allow-Credentials` is absent and the browser enforced it in every credentialed test
Cross-origin policy (application-wide)	Uncredentialed cross-origin read of unauthenticated customer data	Confirmed	See **HARDEN-03** and **VULN-07**. Two customer balances and a full ledger read from an attacker page
/console	Confirmation that the live debugger middleware is genuinely mounted	Confirmed	See **HARDEN-07**. Verified by retrieving the debugger's own internal assets

TARGET	TECHNIQUE	OUTCOME	RESULT DETAIL
/console	Derivation of the debugger PIN from the available file-read capability	Failed	The file read executes on the database host, which does not hold the application files. The chain is broken
/console	PIN brute force and code execution	Not attempted	Out of scope. Must not be reported as remote code execution
54.225.12.30, 50.19.141.238	Port scan of 14 common service ports on both addresses	Failed	Only 443 is open. All other scanned ports are filtered on both addresses
54.225.12.30, 50.19.141.238	TLS protocol and cipher review	Failed	No weakness found. TLS 1.0 and 1.1 are refused, only modern forward-secret ciphers are offered, and the certificate is valid
54.225.12.30, 50.19.141.238	Arbitrary Host and X-Forwarded-Host header injection	Confirmed	See **HARDEN-08**. Accepted without validation, and the documentation redirect downgrades to plain HTTP
54.225.12.30, 50.19.141.238	HTTP request smuggling against the load balancer and origin boundary	Not attempted	Declined as a traffic-integrity risk on shared infrastructure. Version and architecture match only, so this is untested rather than cleared
54.225.12.30, 50.19.141.238	Multipart parser denial of service	Not attempted	Declined as a resource-exhaustion test. See **HARDEN-09**
54.225.12.30, 50.19.141.238	Cloud metadata service access through a server-side request forgery primitive	Not attempted	No such primitive was found anywhere in the application, so there was nothing to point at the metadata service
54.225.12.30, 50.19.141.238	Nameless cookie parsing confusion in the deployed web library	Not attempted	Requires control of a sibling subdomain, and reconnaissance found none. Rated low upstream and immaterial in this deployment

APPENDIX

Methodology, Scope, and Limitations

SCOPE

aldercrest.targets.hound-agents.com (Aldercrest online banking web application and JSON API, hosted at 54.225.12.30 and 50.19.141.238, port 443)

TESTING WINDOW

2026-08-27T18:35:05Z to 2026-08-27T19:17:16Z

TEST PERSPECTIVE

External unauthenticated, plus authenticated with 2 customer-approved test accounts, both holding the standard customer role

METHODOLOGY

External reconnaissance, OWASP web testing, CVE review, access-control testing, controlled exploitation, and verification of findings

Assessment Limitations

Factors that constrained the scope or depth of testing.

Scope restrictions

- Only two customer-approved test accounts were available, both holding the standard customer role. No administrator credentials were provided, so administrator behaviour was tested only through the token-forgery path and through the one administrative endpoint that could be exercised safely.
- Accounts belonging to the target that were not part of the test account set were treated as read-only. Their data was read as authorised non-destructive proof of access, but no sign-in was attempted against them, no password was recovered for them, and nothing belonging to them was modified.
- Administrator account creation and account deletion (`/admin/create_admin` and `/admin/delete_account/{user_id}`) were never invoked, because they are destructive and would act on accounts outside the approved set. The API specification documents an intended refusal for non-administrators, but that behaviour is unverified. Given that the one administrative endpoint that was exercised fell to a forged token, these two should be assumed reachable by the same route until tested.
- Registration was never exercised. Three separate attempts to test whether the registration endpoint accepts privileged fields such as an administrator flag, a starting balance or a chosen account number were stopped by engagement account-lifecycle controls before any request left the testing environment. The API specification documents a debug response field on this route that would likely reveal the answer. This area has zero coverage.
- The third-party AI provider referenced by the application was out of scope and was not tested. Note that no API key was configured at the time of testing, so the assistant returns a fixed fallback response. If a live model is

connected later, the AI chat endpoints should be retested, because user input reaching a real model or a query builder would change the conclusions recorded here.

Deliberately declined tests

- Operating system command execution on the database host was not attempted. Every precondition for it is confirmed present: the application connects as a database superuser, multi-statement execution works, and file reads on that host already succeed. This should be treated as an assessed and unmitigated capability rather than a cleared negative, and it will remain so until the injection in **VULN-03** is fixed.
- No denial-of-service testing was performed. The two published multipart parser issues affecting the deployed web library were matched by version and the vulnerable code path was confirmed reachable, but no exhaustion payload was sent, so no impact was measured.
- HTTP request smuggling was not attempted, because sending deliberately ambiguous request framing through a shared load balancer risks affecting other traffic. The version and the architecture both match the published risk scenario, so this remains untested rather than cleared.
- The debugger PIN was not brute-forced and no code was executed through the debug console.

Blocked by engagement safety controls

The following tests were stopped by the engagement's own safety controls before any request reached the target. They produced no result in either direction and must be treated as untested rather than clean.

- Path traversal through the upload filename. An indirect observation suggests the server normalises filenames in a way that would also collapse traversal sequences, but that was never confirmed.
- Zero, negative, below-minimum and overdraft-scale amounts on the bill payment endpoint, and field-name injection on the same endpoint. The equivalent rule failures were confirmed on the transfer and loan workflows, so the underlying pattern is established, but this specific endpoint's edge-case behaviour is unproven.
- The comparison probes needed to establish a username enumeration oracle. This is largely academic here, because the complete user list is already disclosed through the confirmed injection findings and through the admin panel.

Areas where deeper testing is recommended in a follow-up engagement

- Registration, administrator account creation and account deletion, once a dedicated test identity and an appropriate change window are available.
- The bill payment endpoint's amount handling and field handling, once a higher financial-write budget is agreed.
- The AI chat feature, once a live model and a working API key are deployed.
- Internal supply-chain assurance: the build pipeline, CI/CD controls, package registry permissions, private dependencies, software bills of materials and the dependency-update process. None of these were in scope, and no supply-chain assurance beyond externally observable component versions should be inferred from this report.
- Logging, alerting and detection capability, which cannot be assessed from outside. Nothing the assessment did produced any observable defensive response, and it is worth confirming internally whether any of it was recorded.

- A retest after remediation. Several findings in this report interact, and fixing one without the others leaves the outcome reachable by a second route. In particular, **VULN-19** and **VULN-23** reach the same card-freeze impact independently, and **VULN-02** makes several findings that nominally require a sign-in reachable without one.