VAMPI API SECURITY TESTING

By
Peace Dennis

VAmPI API Security Testing

Introduction

This is a hands-on project on **VAmPI (Vulnerable API)**, a deliberately insecure API designed for learning and practicing API security testing.

The goal here isn't just to "find bugs," but to understand how real-world API vulnerabilities actually look, how they're exploited, and why they matter. Here is the step-by-step walkthrough I published on Medium.

Vulnerabilities Found:

Unauthenticated Issues

- Broken authentication logic
- Username and password enumeration
- Excessive data exposure via debug endpoints
- Mass Assignment
- SQL injection leading to full database disclosure
- Lack of Resource Limits & Rate Limiting

Authenticated Issues

- Broken Object Level Authorization (BOLA)
- Unauthorized Password Change
- JWT Authentication Bypass via Weak Signing Key
- ReDoS (Regular Expression Denial of Service)

All of these are mapped back to real-world API security failures.

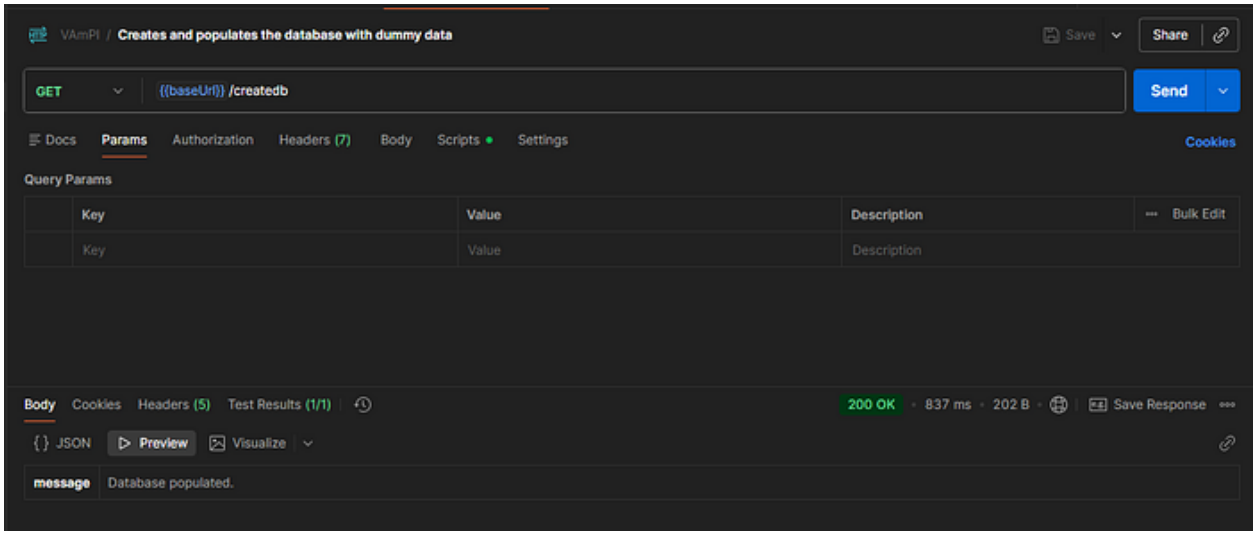
Tools Used:

- **Postman** for sending and organizing API requests
- **Burp Suite** for intercepting, modifying, and brute-forcing requests
- **sqlmap** for automated SQL injection exploitation
- **jwttool** for cracking and forging JWTs

Findings

Before we start looking for vulnerabilities, there's one quick setup step we need to take care of. We need to create and populate the database. If we skip this, the API will just throw internal server errors because there's no data to retrieve yet. To do this, we'll call the `GET /createdb` endpoint. This endpoint seeds the application with default users, books, and admin accounts, data we'll rely on later when we start exploiting vulnerabilities.

For this step, we'll use **Postman**.



Peace Dennis

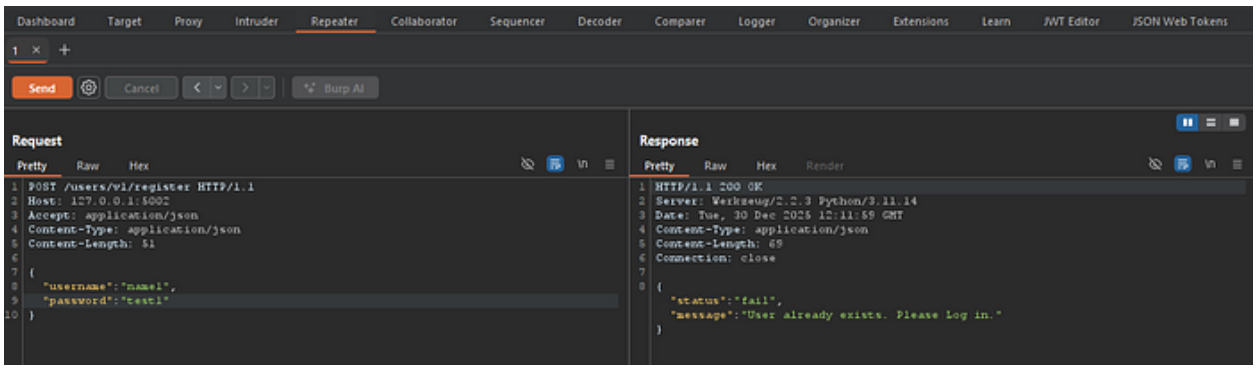
Database populated

Part 1: Unauthenticated Testing (Broken Authentication)

Now for the interesting part. The API does give us the option to register a new account using `/users/v1/register`, but instead of jumping straight to that, let's see what we can do **without logging in at all**. The goal here is to start with zero access and slowly work our way up the privilege chain. This approach helps us uncover **broken authentication** issues, situations where the API trusts users it shouldn't.

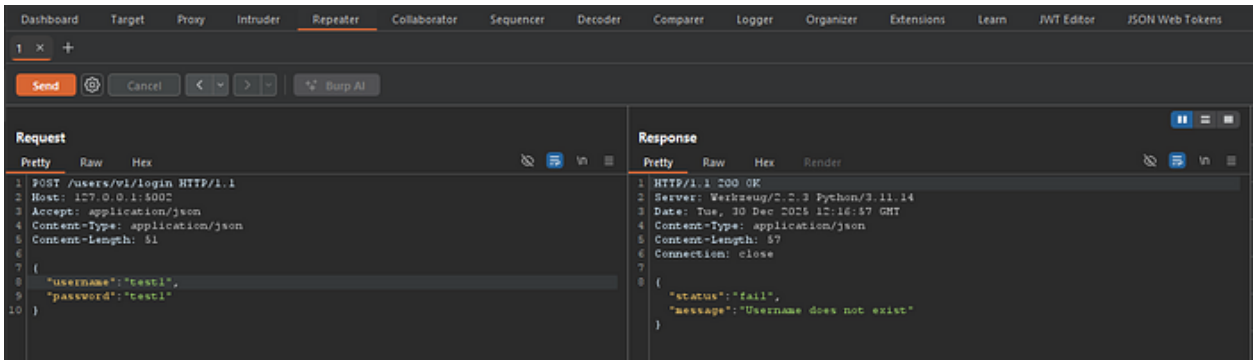
1. User Enumeration

Our first target is user enumeration. Let's see how easy it is to figure out which usernames actually exist in the system. One simple way is through the `/users/v1/register` endpoint. If you try to register with a username that already exists, the API tells you exactly that, "the user already exists." That might sound helpful, but it also gives attackers a clear signal that the username is valid.



User already exists

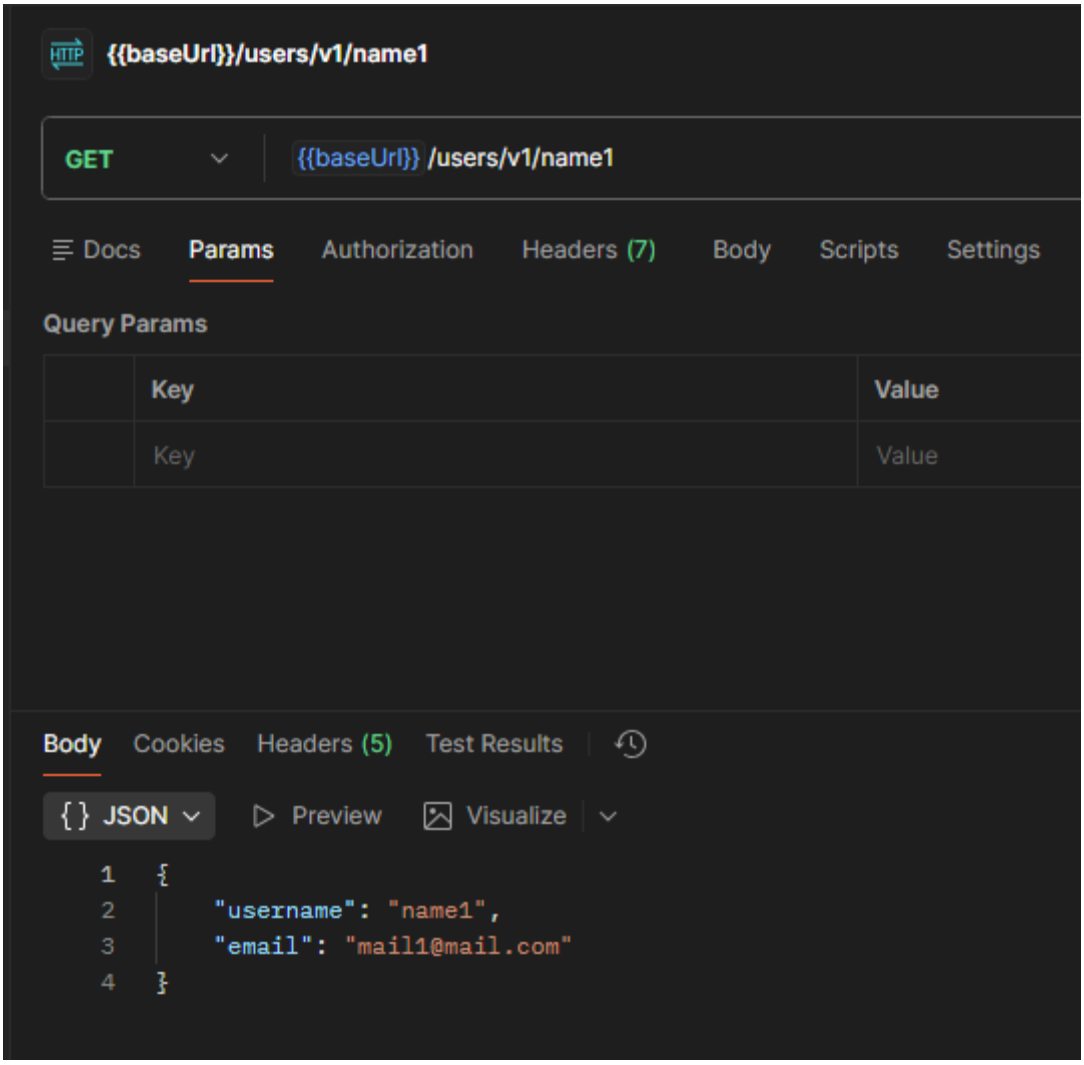
The same thing happens during login. If you attempt to log in with a username that doesn't exist, the API responds with **"Username does not exist."** That message alone is enough to confirm whether a user is real or not.



Username does not exist

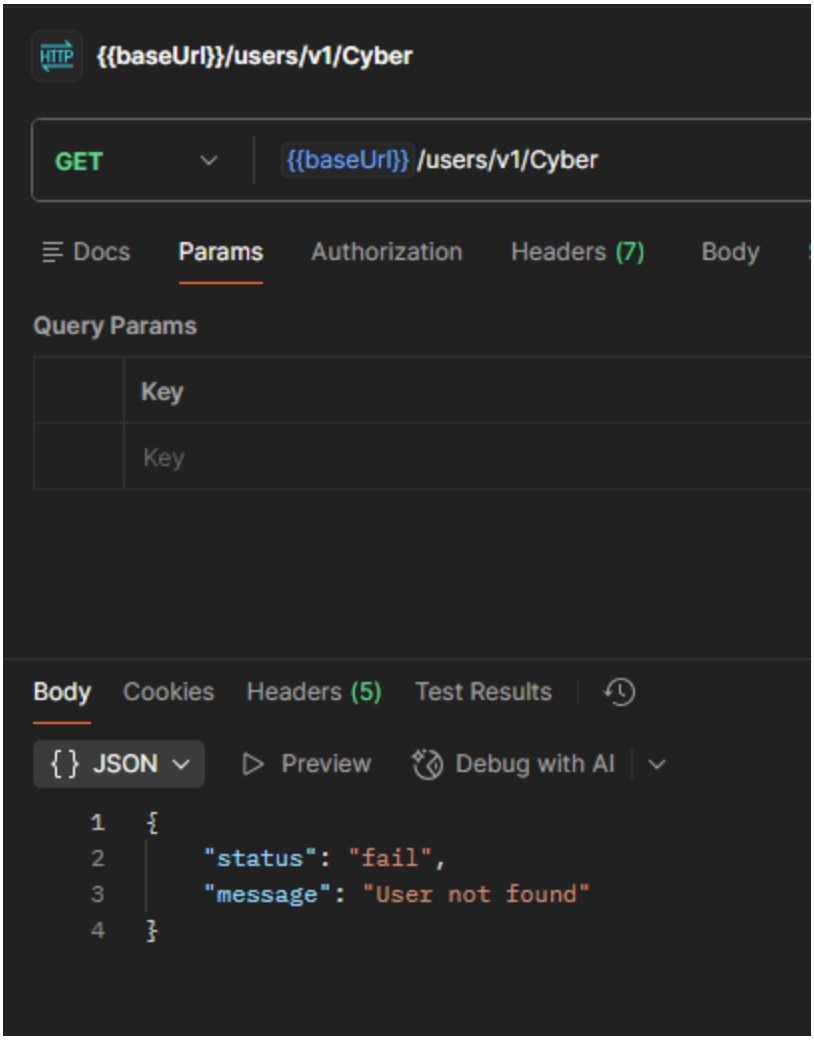
There's another option too. You can send a `GET` request to `/users/v1/{username}` for example, `GET /users/v1/name1` and see what comes back:

- If the user exists, the API returns that user's basic information.



Specific user details

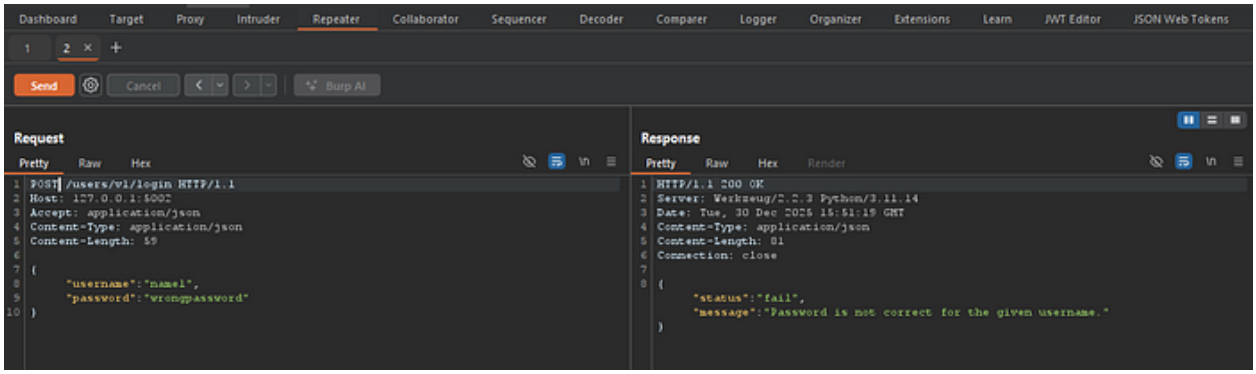
- If the user doesn't exist, you get a clear "user not found" response.



User not found

2. Password Enumeration

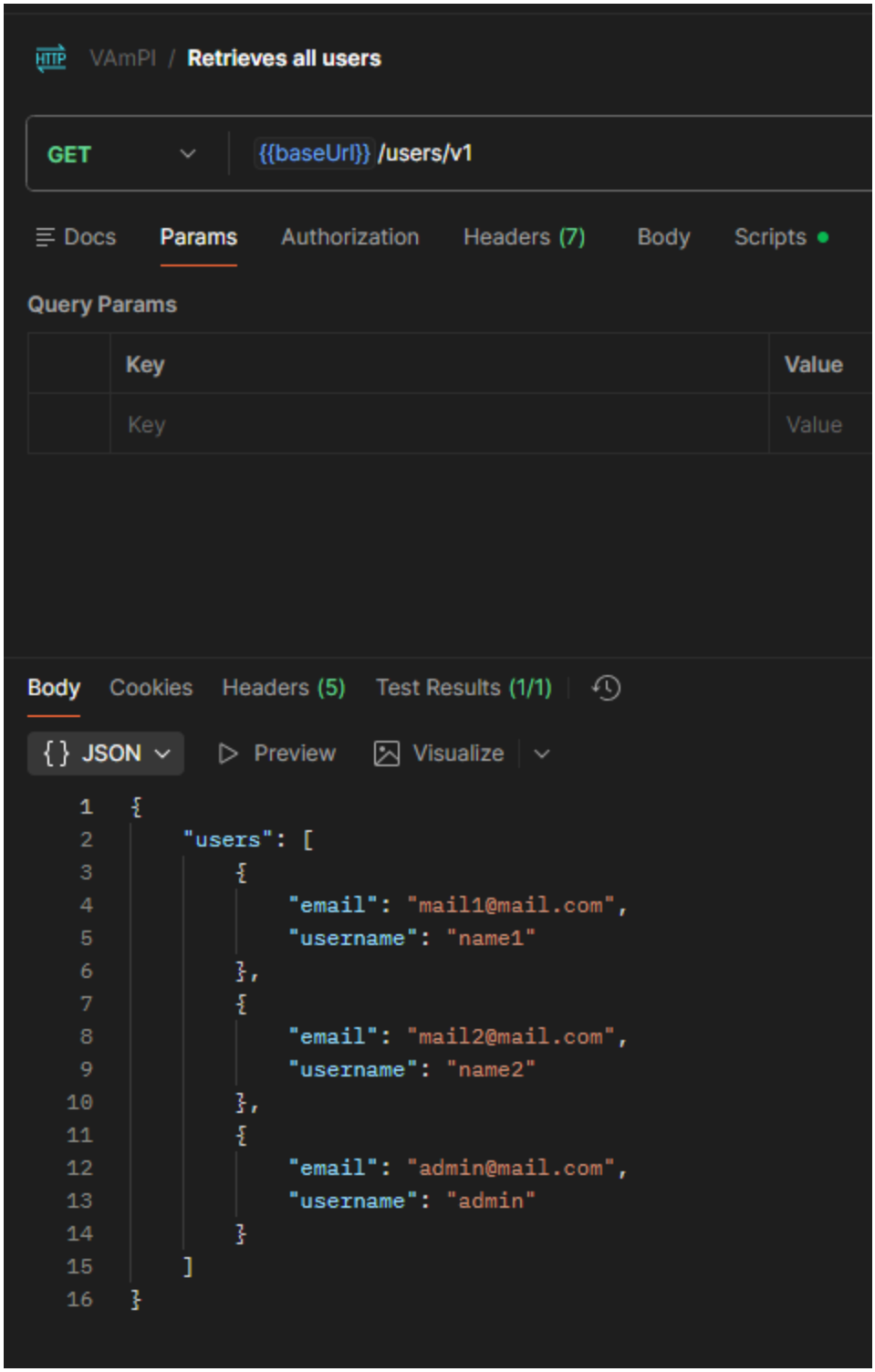
There's also a clear way to tell whether a password is correct or not, and that's a big problem. When you try to log in using a **valid username** but enter the **wrong password**, the API responds with a very specific message: **"Password is not correct for the given username."**



Password not correct for the given username

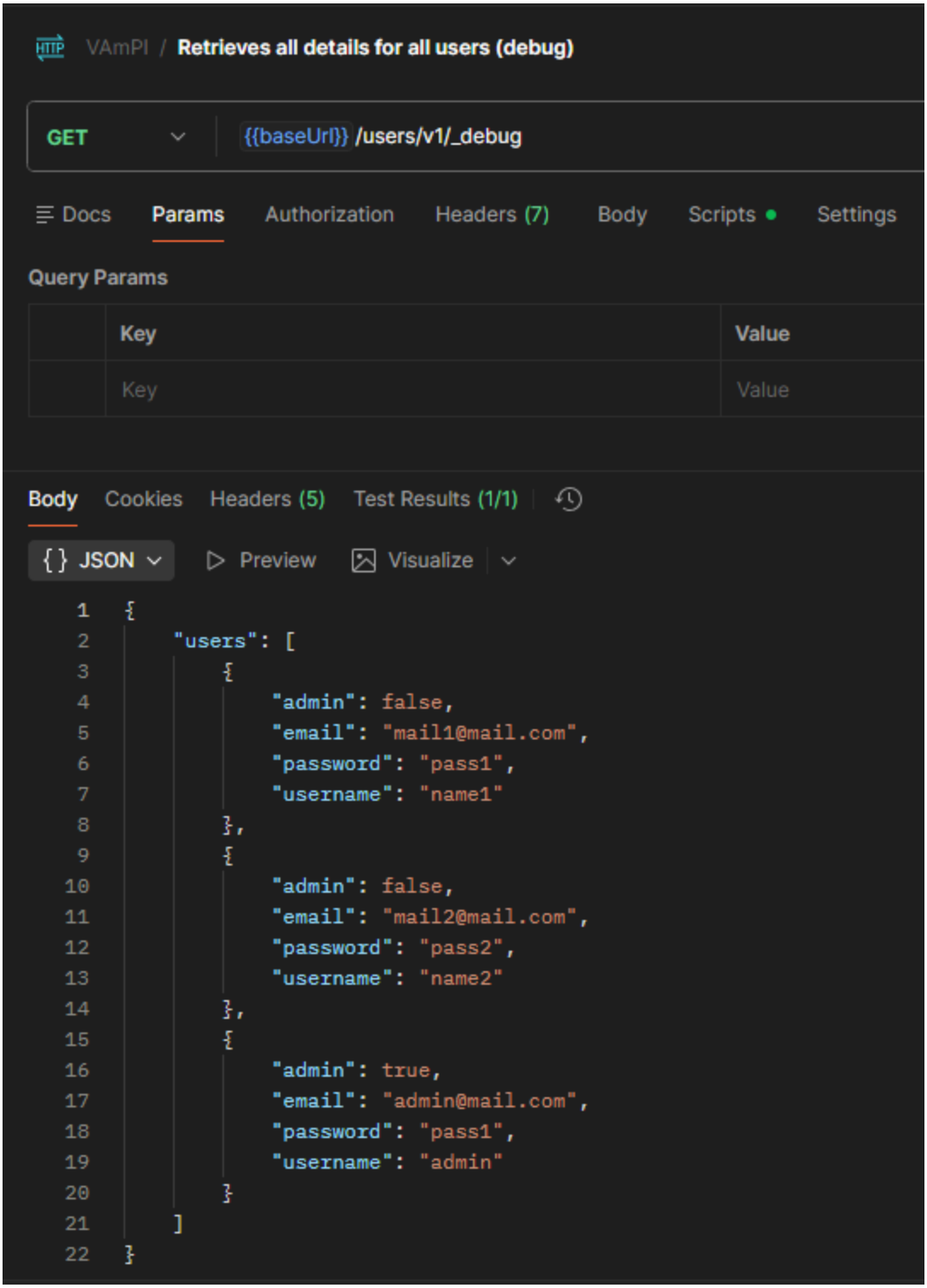
3. Excessive Data Exposure

By sending a request to `GET /users/v1`, we can retrieve the list of users that exist in the system. The problem? This endpoint doesn't require authentication. That means anyone, literally anyone, can query the API and see all registered users and their basic info. From an attacker's perspective, this is gold. Usernames are often the first stepping stone to more serious attacks like brute force attempts, password spraying, or privilege escalation.



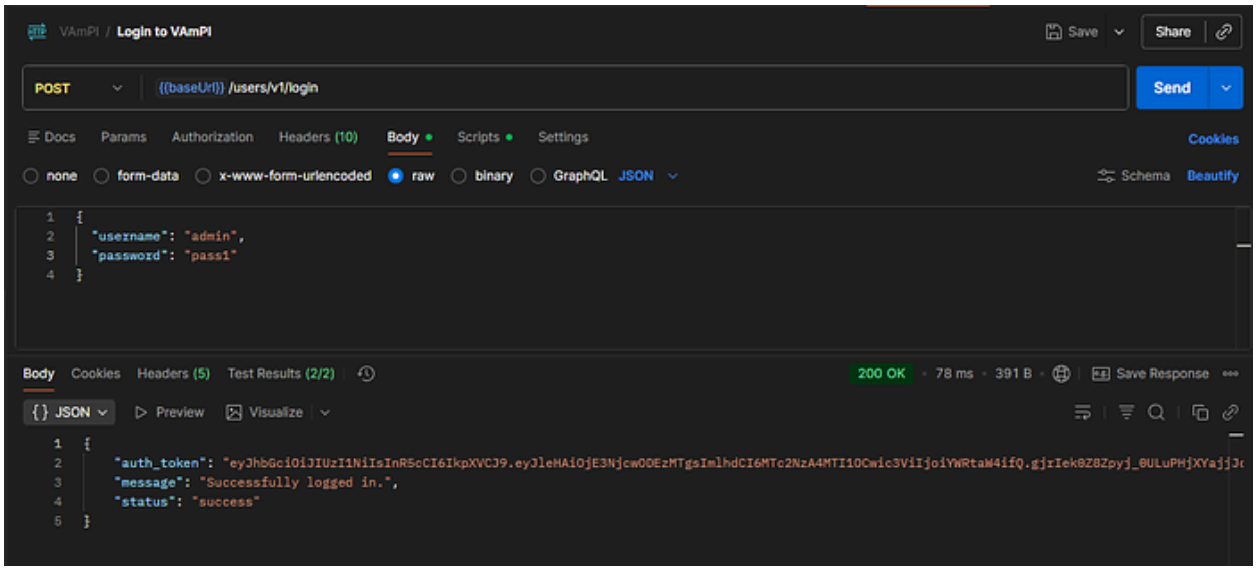
Available users

Things get even worse when we look at the `GET /users/v1/_debug` endpoint. Unlike the earlier endpoints that only returned limited user information, this one spills **everything,** full user records with far more detail than any regular user should ever see. The debug endpoint is usually meant for development or troubleshooting, but leaving it exposed in a production environment is dangerous.



Full user's details from the debug endpoint

To confirm that this data is actually real, and to understand the real-world impact of flaws like this, let's take it one step further. We'll try logging in using the admin credentials exposed by the debug endpoint. All we need to do is send a **POST** request to **/users/v1/login**, using the admin username and password we just discovered. If the vulnerability is as serious as it looks, this should give us admin access.



Successfully logged in using the admin details exposed at the debug endpoint

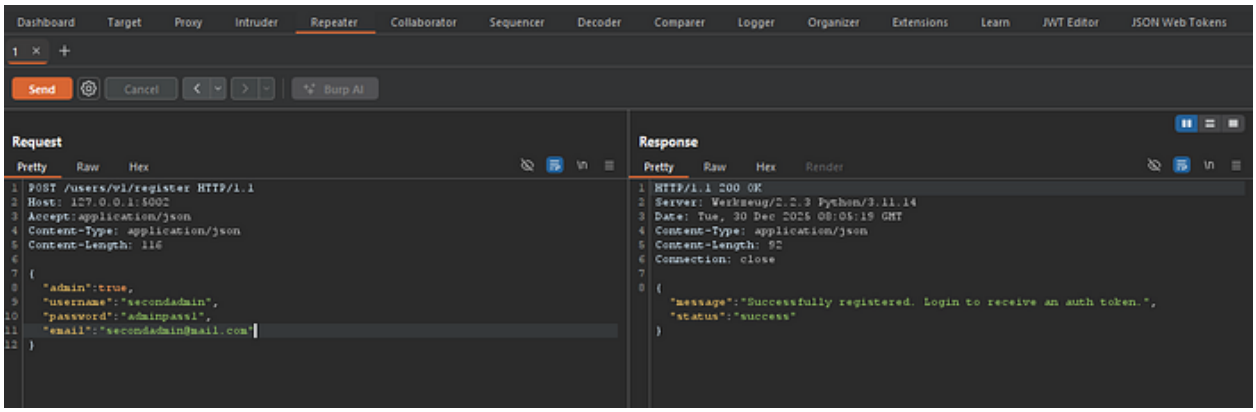
And sure enough, it works. We're able to log in successfully and received an `auth_token`.

One quick thing to note: this isn't a website where you log in once and stay logged in. In our case here, authentication usually works through tokens. Once you log in, the API gives you a token, and you have to include that token with every request to endpoints that require authentication. But since we're still exploring what's possible **without** proper authentication, we won't need a token now.

4. Mass Assignment — Creating an Admin User

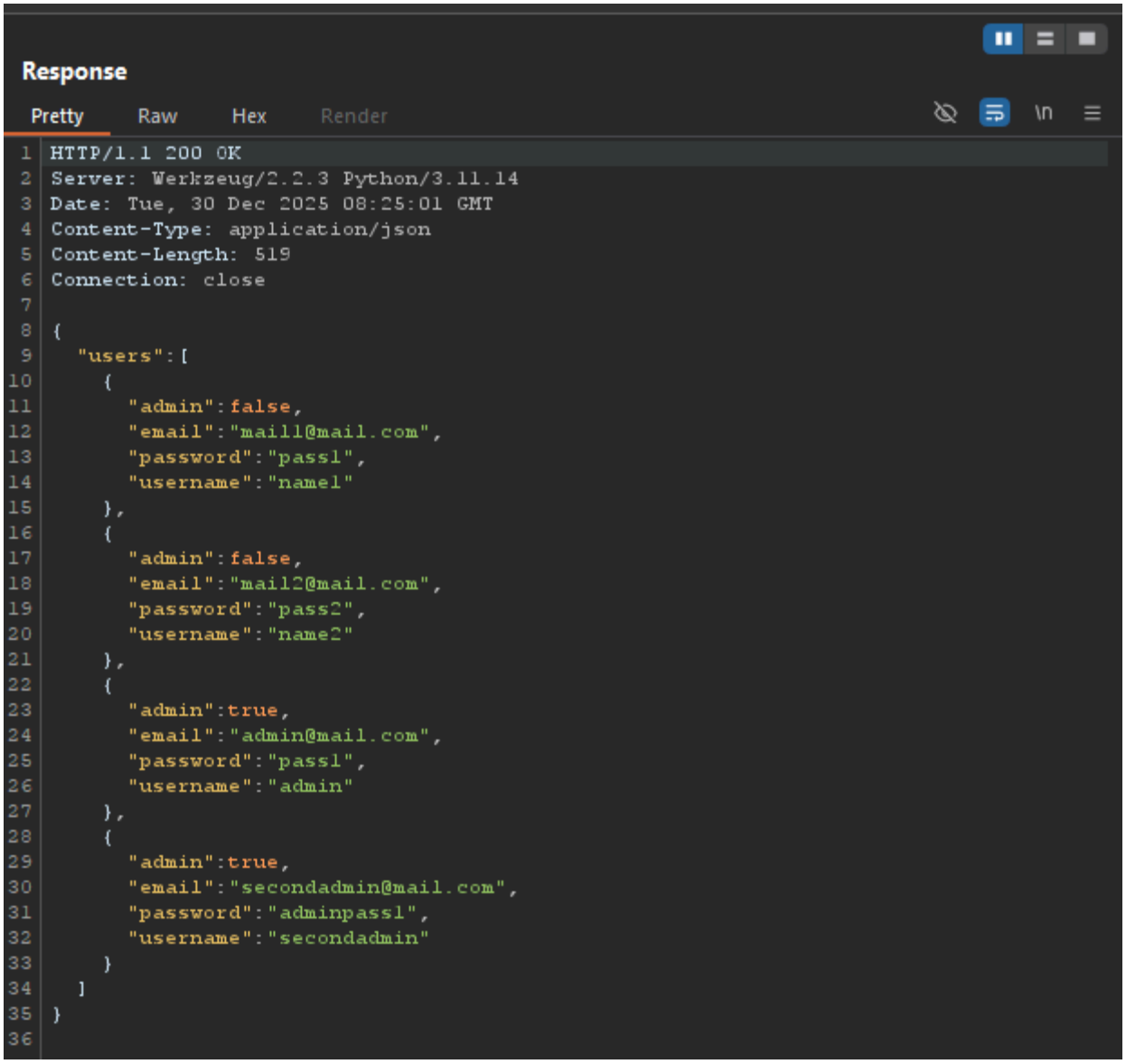
While browsing the `/users/v1/_debug` endpoint, one detail really stands out: admin users have an **admin** property set to **true**. That immediately raises a question: What happens if we try to set that field ourselves? Let's find out by registering a new user and simply including the **admin** property in the request. We'll send the following payload to `/users/v1/register`:

```
{
 "admin": true,
 "username": "secondadmin",
 "password": "adminpass1",
 "email": "secondadmin@mail.com"
}
```



Registered a new admin by just adding admin true

And… it works. The request went through without any complaints. Let's head back to `/users/v1/_debug` and check the user list again to confirm our newly added admin.

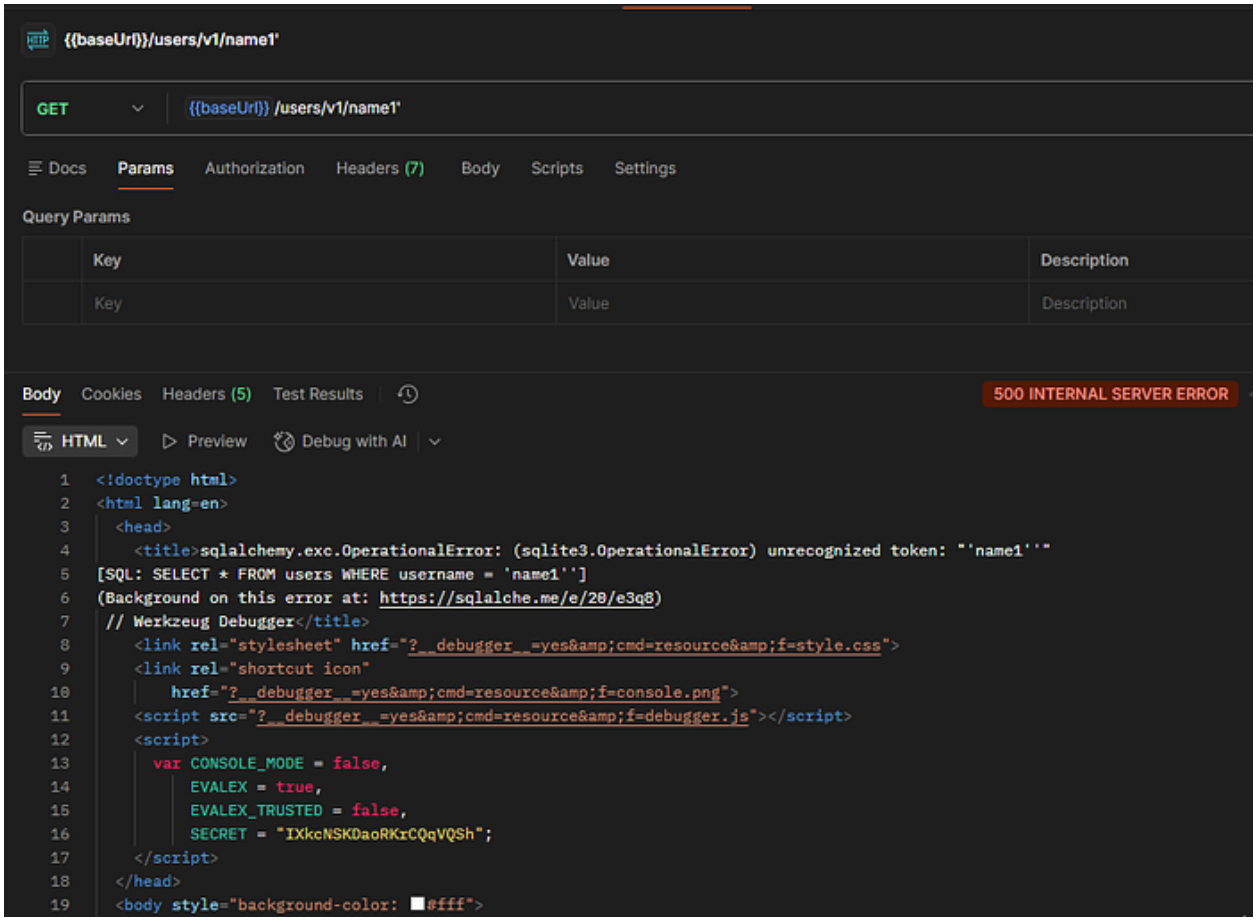


Sure enough, our new admin is there. Now we have two separate ways to gain admin access to the API. And yeah, I used Burp Suite for this part.

5. SQL Injection

To check for SQL injection, all we have to do is something very simple: add a single quote to the username when requesting a user. When we do that, the API doesn't handle it gracefully. Instead, it crashes and throws back a raw SQL error:

```
sqlalchemy.exc.OperationalError: (sqlite3.OperationalError) unrecognized token:
"'name1'"
[SQL: SELECT * FROM users WHERE username = 'name1'']
(Background on this error at: https://sqlalche.me/e/20/e3q8)
```



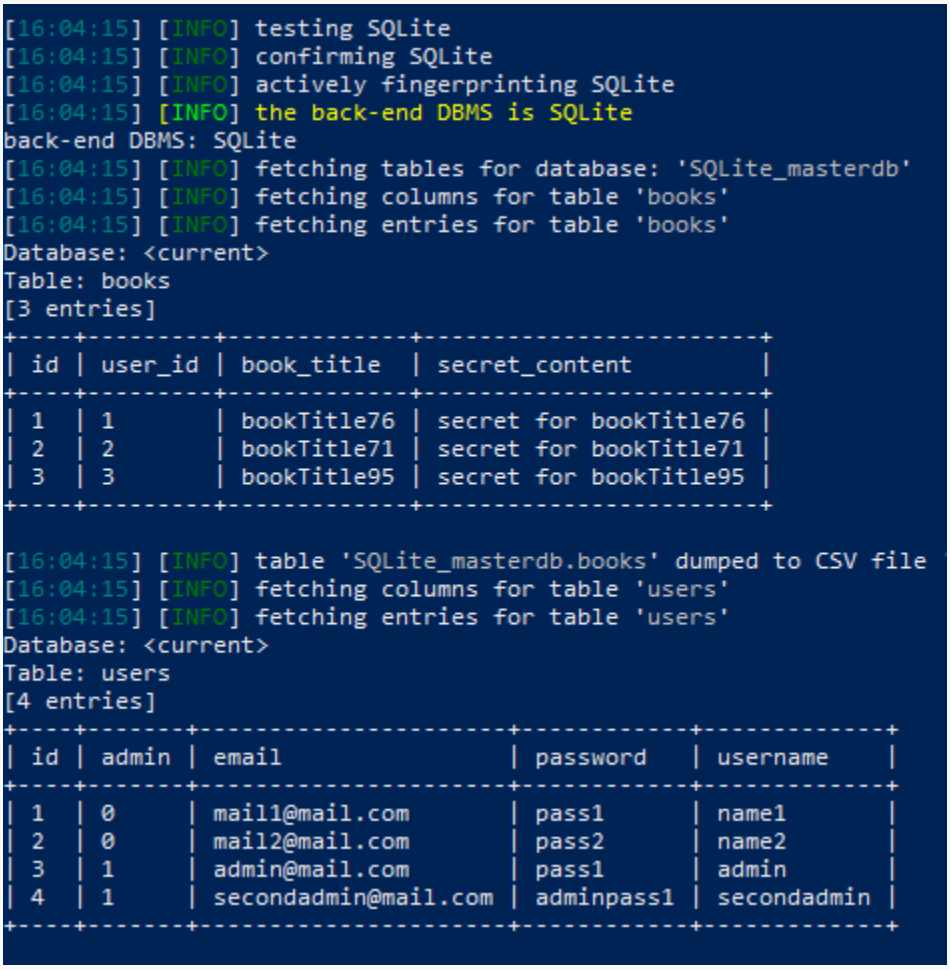
SQL injection

This tells us a lot more than it should. Not only does it confirm that user input is being directly inserted into a SQL query, but it also exposes the underlying database engine and query structure. So, at this point, we already know that the database is **SQLite**. That makes the next step pretty straightforward. We can now bring in **SQLmap** and let it do the heavy lifting. By

pointing it at the vulnerable endpoint and telling it which database engine we're dealing with, SQLmap is then able to extract and dump the entire database.

Running a command like this will do exactly that:

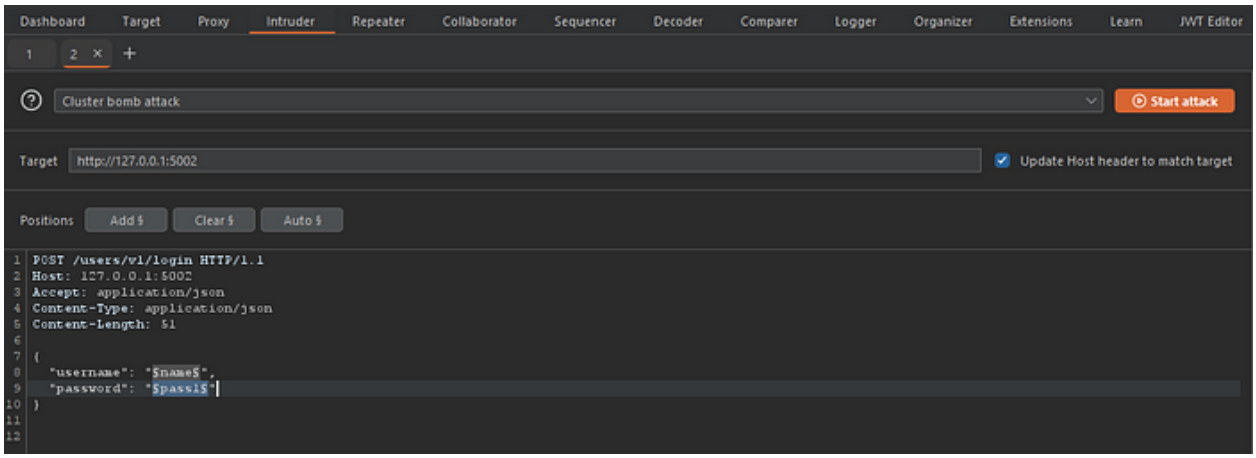
```
sqlmap -u http://localhost:5002/users/v1/name1* --dbms=sqlite --dump
```



Database information dumped as a CSV file

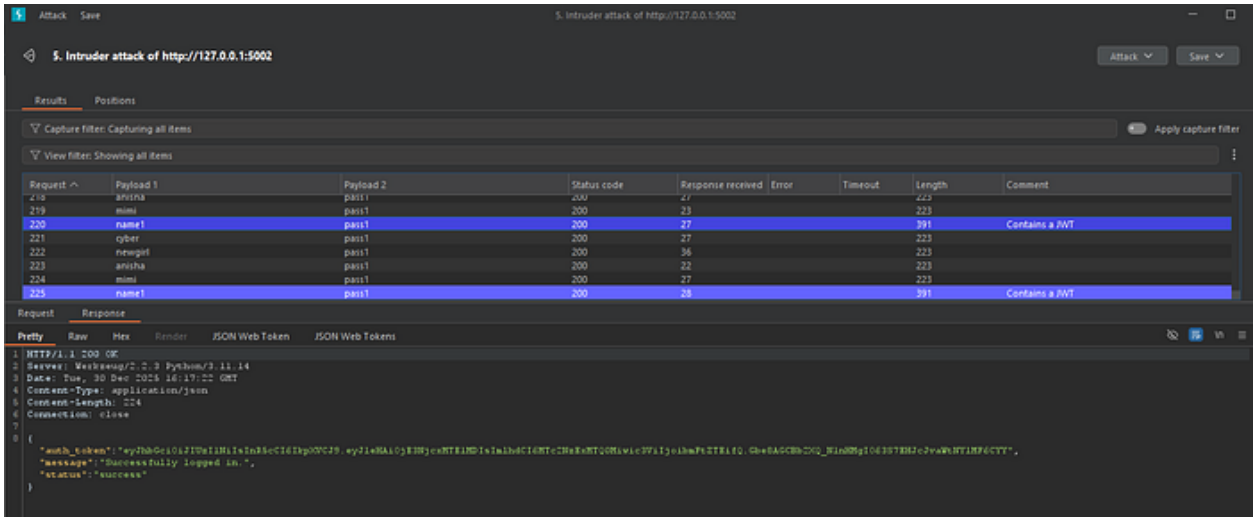
6. Lack of Resource Limits & Rate Limiting

Here, you will see how the application doesn't care how many requests you send. There's no throttling, no delays, and no blocking, no matter how aggressive the traffic is. Let's try brute-forcing the login endpoint using **Burp Intruder.** Let's set up a **Cluster Bomb** attack and define variables for both the username and the password.



Cluster bomb attack

Now, we're able to send **well over 200 requests** with nothing stopping us. No CAPTCHA, no temporary lockout, no rate limit warnings. Even worse, we got a valid login.

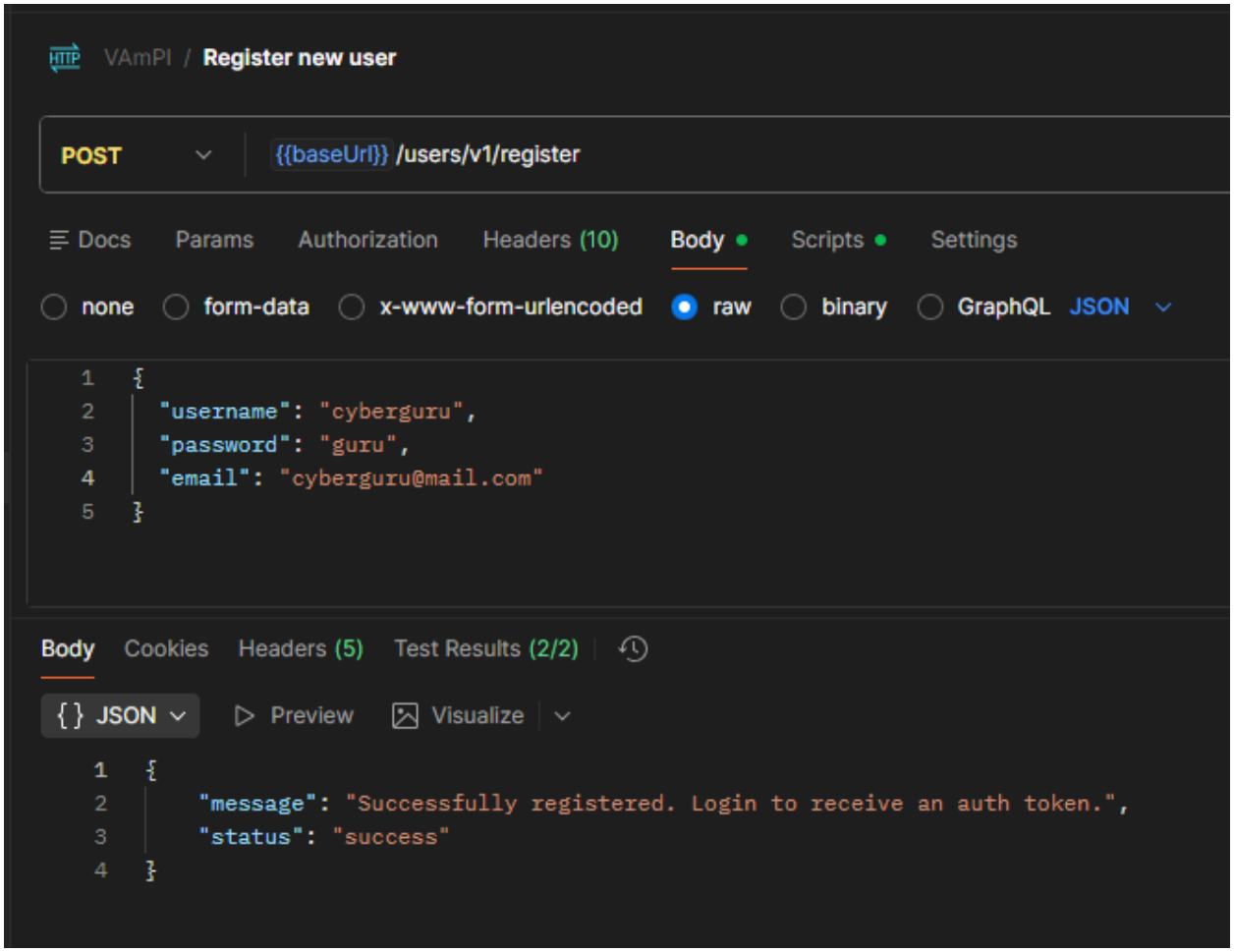


No rate limiting

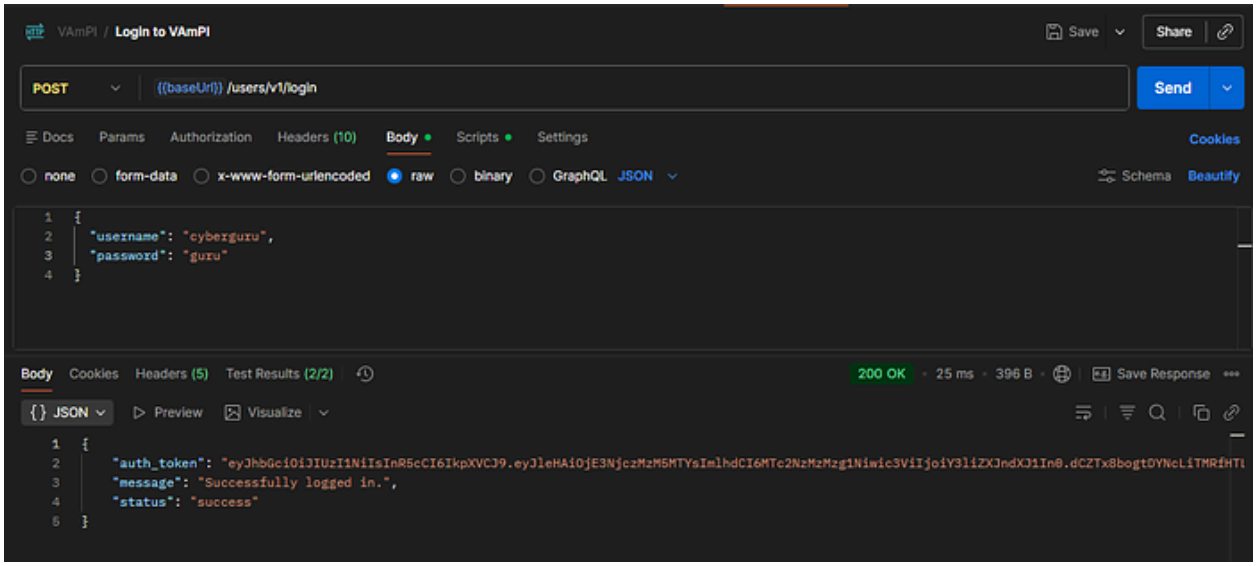
Peace Dennis

Part 2: Authenticated Testing

Now that we've seen what's possible without logging in, let's switch gears and see what happens once we **do** authenticate. First, let's register a new user by sending a **POST** request to `/users/v1/register` with our user details. Once that's done, we will then log in using the same credentials to receive an `auth_token`.



New user



New user's successful login

From this point on, the token received will allow us to make authenticated requests to the API. Every protected endpoint expects it, and without it, access will be denied. The token expires after 60 seconds, so you can always visit the login endpoint to log in again and obtain a new token to continue.

1. Broken Object Level Authorization (BOLA)

According to the API's own description, each book belongs to a specific user, and only the owner of that book should be able to view its secret. We also know there's an endpoint that lists books: `/books/v1`.

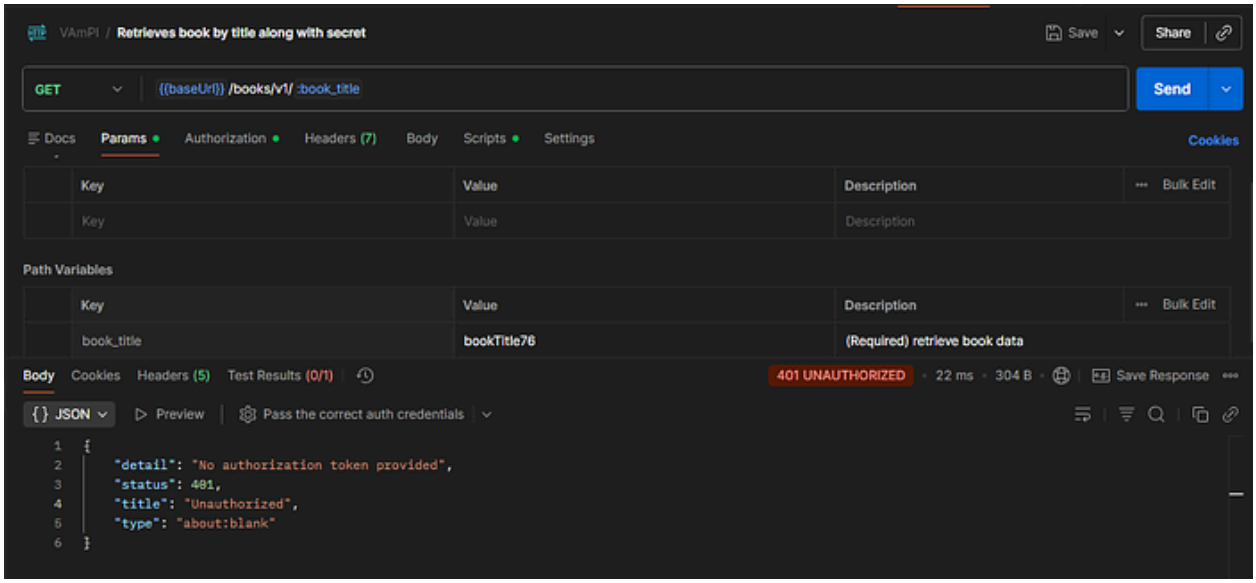
Let's start by calling it and seeing what it returns.

Retrieved all books

So far, everything looks fine. The response matches what the API promises and doesn't expose anything it shouldn't, and this actually works without needing authentication.

Peace Dennis

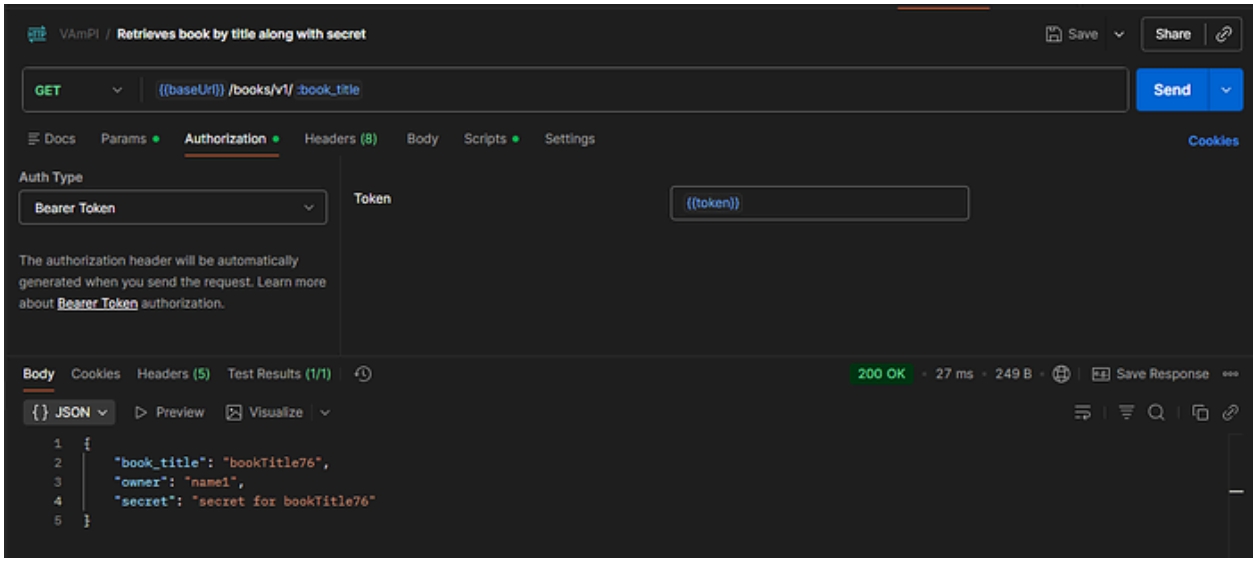
Next, let's try requesting a single book directly using `/books/v1/bookTitle76`, without providing any authentication token.



Authorization is required to retrieve a single book

Again, the behavior is exactly what we expected, the request is rejected.

But now comes the interesting part. What happens if we include our **auth_token**, even though we never created this book and don't own it?



Retrieved the book secret that belongs to another user

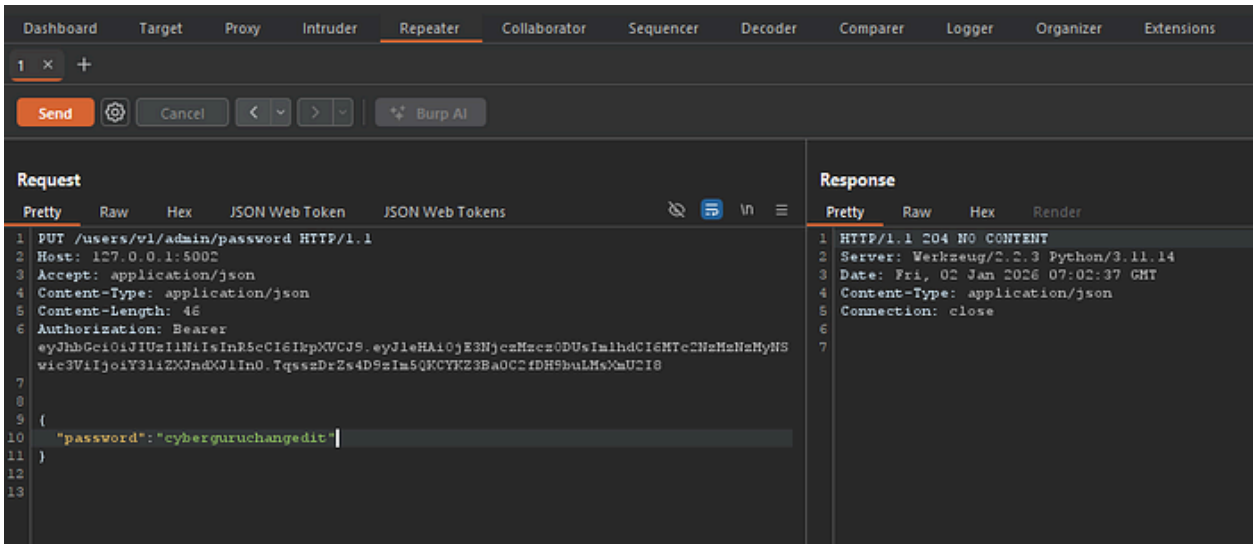
And… bingo! The request succeeds.

This means any valid token, belonging to *any* user, can be used to access the secret content of *any* user's book. Ownership isn't being checked at all. That's a clear case of **Broken Object Level Authorization (BOLA)**. The API verifies that you're logged in, but it doesn't verify *what* you're allowed to access. Once you have a token, the doors are wide open.

2. Unauthorized Password Change — Broken Function Level Authorization (BFLA)

There's an endpoint that lets a user change a password using a `PUT` request: `/users/v1/<username>/password`. Changing *your own* password makes sense, but the real question is: can you change **someone else's password**?

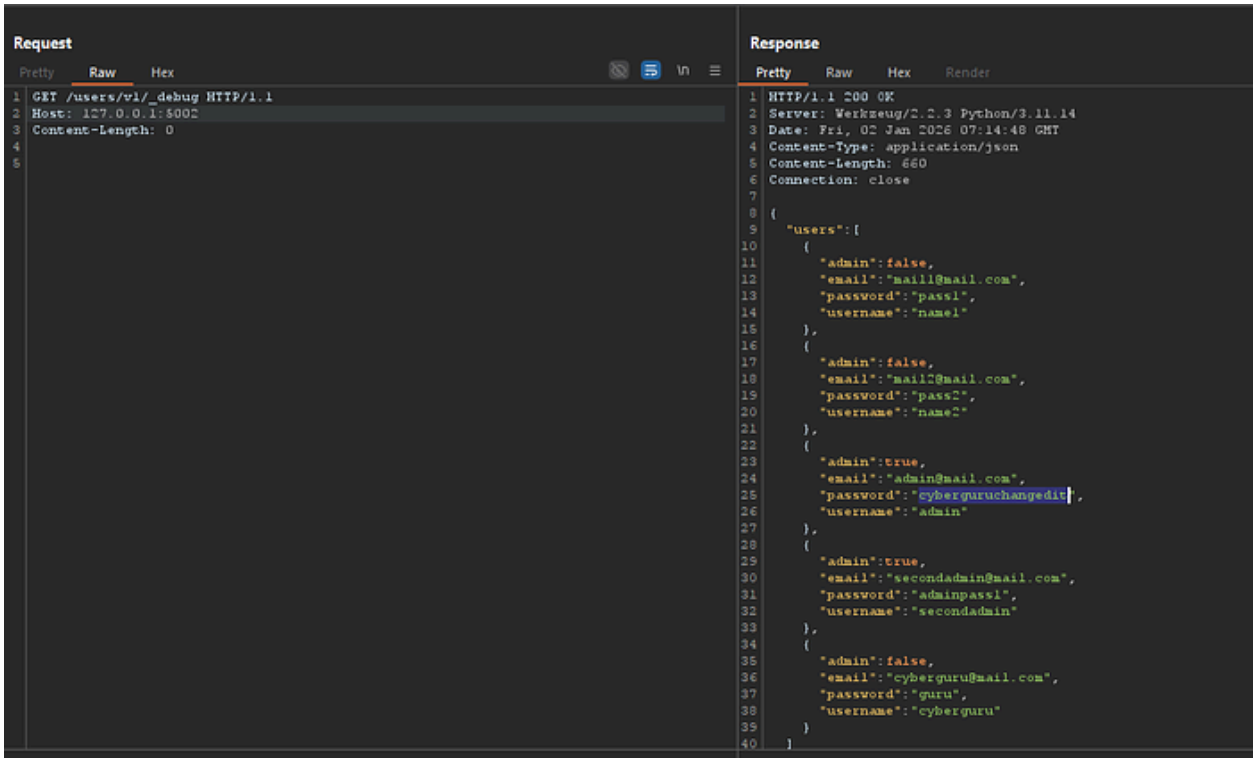
To find out, let's try something simple. We'll attempt to change the **admin user's** password, but instead of using an admin token, we'll use the token from a regular, non-admin user, let's say the **cyberguru** account.



Password changed for admin

So we sent a password change request targeting the admin user using the **cyberguru** user's token. And… it worked!

The admin's password is successfully changed, even though the request came from a completely different, non-privileged user.



Admin password changed

3. JWT Authentication Bypass via Weak Signing Key

There's another serious issue hiding in the authentication system. When a user logs in, the API issues a **JWT token** that expires after 60 seconds. That sounds reasonably secure, right? But the problem isn't the expiry time, it's the **signing key** used to protect the token. The key is weak enough that it can be cracked using tools like **jwttool**.

Before we move forward, let's start by decoding the token so we can see what's inside it.

```
====================
Decoded Token Values:
====================

Token header values:
[+] alg = "HS256"
[+] typ = "JWT"

Token payload values:
[+] exp = 1767342025    ==> TIMESTAMP = 2026-01-02 09:20:25 (UTC)
[+] iat = 1767341965    ==> TIMESTAMP = 2026-01-02 09:19:25 (UTC)
[+] sub = "cyberguru"

Seen timestamps:
[*] exp was seen
[*] iat is earlier than exp by: 0 days, 0 hours, 1 mins
[-] TOKEN IS EXPIRED!

----------------------
JWT common timestamps:
iat = IssuedAt
exp = Expires
nbf = NotBefore
----------------------
```

The decoded token

Now that we know its structure, the next step is to crack the signing key. With a weak key like the one we have, this shouldn't take long.

The cracked secret key

And this is where things get risky.

After recovering the signing key, we can forge our own JWT. That means we're no longer limited to the role or permissions we were originally given. We can modify the token to claim **admin privileges**, push the expiration time far into the future, reset the `iat` to the current time, and then re-sign the token using the cracked key.

```
Please select a field number:
(or 0 to Continue)
> 0

Token payload values:
[1] exp = 1767342025    ==> TIMESTAMP = 2026-01-02 09:20:25 (UTC)
[2] iat = 1767341965    ==> TIMESTAMP = 2026-01-02 09:19:25 (UTC)
[3] sub = "cyberguru"
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 1

Current value of exp is: 1767342025
Please enter new value and hit ENTER
> 1798877965
[1] exp = 1798877965    ==> TIMESTAMP = 2027-01-02 09:19:25 (UTC)
[2] iat = 1767341965    ==> TIMESTAMP = 2026-01-02 09:19:25 (UTC)
[3] sub = "cyberguru"
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 2

Current value of iat is: 1767341965
```

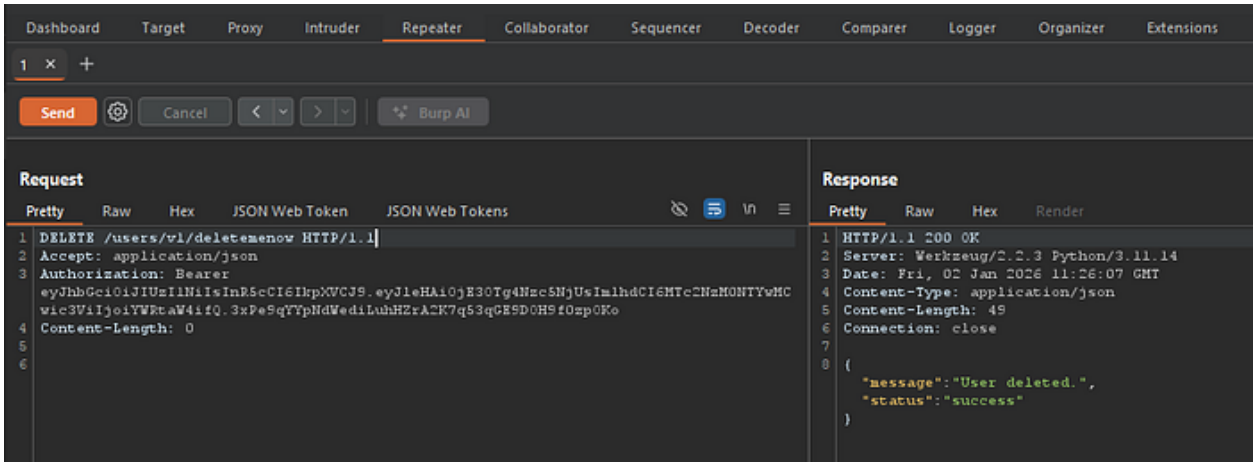
Changed exp value

Peace Dennis

```
Current value of iat is: 1767341965
Please enter new value and hit ENTER
> 1767345600
[1] exp = 1798877965    ==> TIMESTAMP = 2027-01-02 09:19:25 (UTC)
[2] iat = 1767345600    ==> TIMESTAMP = 2026-01-02 10:20:00 (UTC)
[3] sub = "cyberguru"
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 3

Current value of sub is: cyberguru
Please enter new value and hit ENTER
> admin
[1] exp = 1798877965    ==> TIMESTAMP = 2027-01-02 09:19:25 (UTC)
[2] iat = 1767345600    ==> TIMESTAMP = 2026-01-02 10:20:00 (UTC)
[3] sub = "admin"
[4] *ADD A VALUE*
[5] *DELETE A VALUE*
[6] *UPDATE TIMESTAMPS*
[0] Continue to next step

Please select a field number:
(or 0 to Continue)
> 0
jwttool_6d3f052eed857262550281caa6efd853 - Tampered token - HMAC Signing:
[+] eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjE3OTg4Nzc5NjUsImlhdCI6MTc2NzM0NTYwMCwic3ViIjoiYWRtaW4ifQ.
3xPe9qYYpNdWediLuhHZrA2K7q53qGE9DOH9f0zpOKo
```

Changed iat and sub values

Now comes the real test. Using this forged admin token, let's send a request to delete another user.



Deleted a user using the modified token

And… voilà. The API accepts the token without question, and the user is deleted. At this point, authentication is completely broken. Anyone who cracks the signing key can impersonate an admin, perform destructive actions, and fully compromise the application.

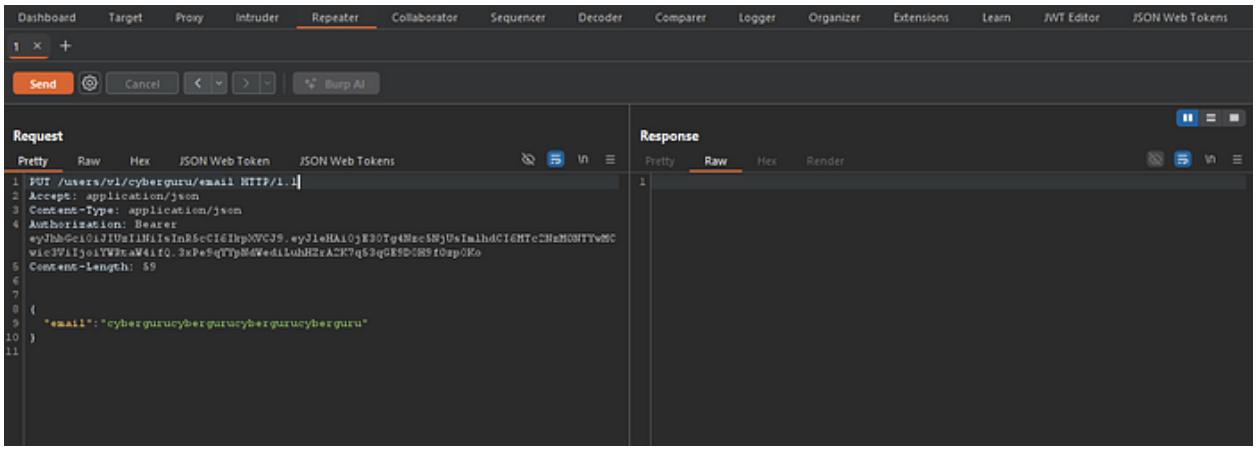
4. ReDoS (Regular Expression Denial of Service)

This issue is a form of **self–denial of service**, where the API essentially overwhelms itself. Instead of crashing the app directly, you give it input that forces it to do an insane amount of work until it slows down or completely hangs. Most of the time, this happens because user input isn't being handled safely.

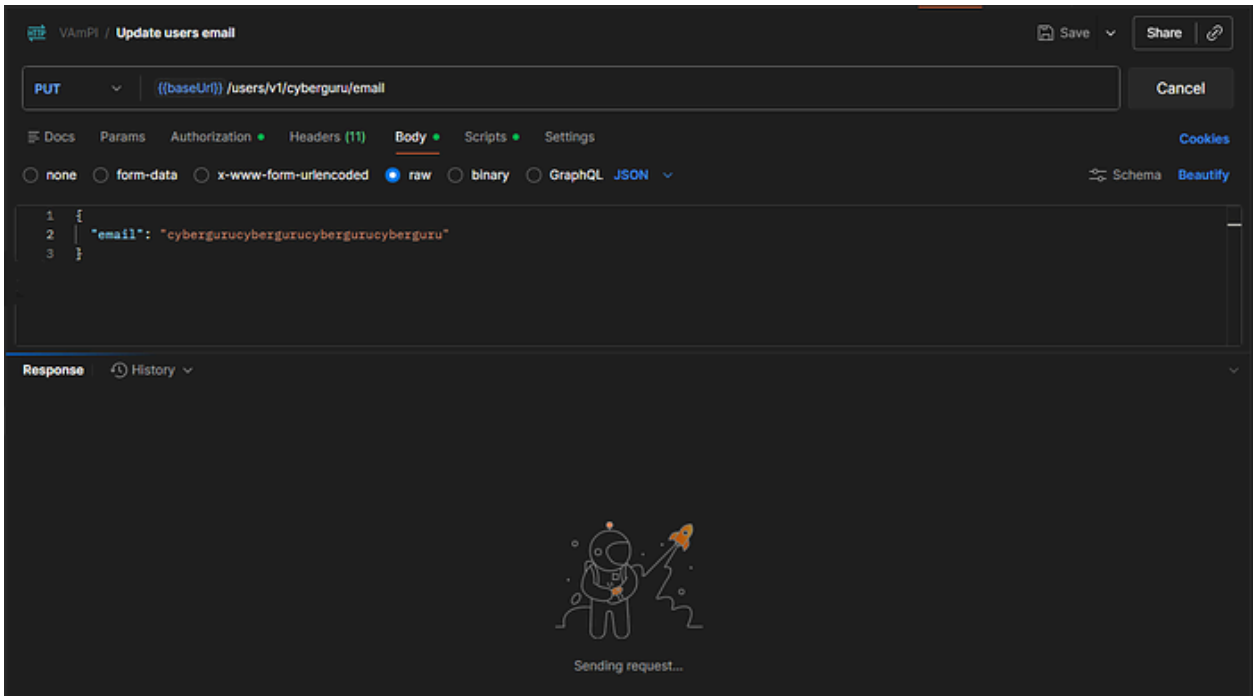
In this case, we can bring the application to a stop by sending an **authenticated** request to `/users/v1/{username}/email` for updating the email address and supplying a very long string that clearly isn't a real email address. For example:

```
{
 "email": "cybergurucybergurucybergurucyberguru"
}
```

Once this request is sent, the API stops responding. The longer the string, the longer the app takes to recover, sometimes you'll even need to restart the service entirely, especially if you are working with Postman.



The API stopped responding in Burp Suite



The API went into a loop on Postman

Peace Dennis

What's happening behind the scenes is that the email field is being validated using a poorly designed regular expression. That regex takes a *very* long time to decide whether the input is a valid email, and during that time, the application is essentially stuck. This is a classic **ReDoS vulnerability**, and it was quite easy to trigger.