

Secure and Scalable SMS Report

Zuhayer Alam

[illegible]

Contents

Introduction	2
Table of VMs	2
Vulnerabilities of Previous SMS Version	3
Preventing User from Using HTTP	6
Preventing SQL Injection	6
Preventing Cross-Site Scripting Attack	8
HAProxy Statistics (HTTPS & TCP)	10
Graph Results (HTTPS)	10
Graph Results (TCP)	12
Analysis of the Data	13

Introduction

This report briefly presents the structure of the Shopping Management System project developed as part of the assessment tasks of KIT214. In the first assignment, the SMS website was developed without any substantial security measures to prevent SQL Injection and Cross-Site Scripting Software attacks. Later in part 2 of the assignment, the website was improved with security features and scalability. The report shows how both versions of the website (vulnerable and secured) compare to each other and also includes the performance graphs and analysis of the performance of the website which was conducted by Jmeter.

Table of VMs

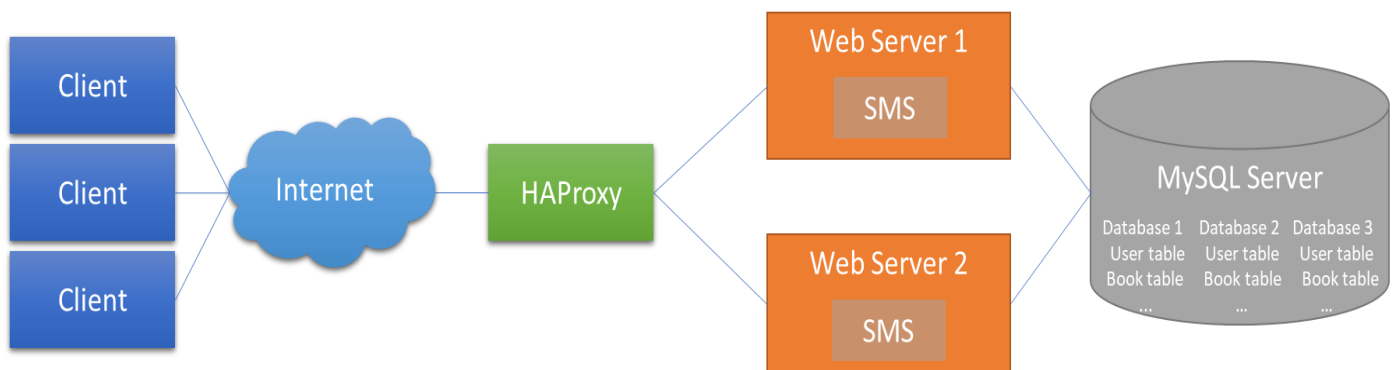


Figure 1: Client-Server model of the Shopping Management System Website

The above model was taken into consideration during the implementation of the improved version of the website and therefore, four Virtual Machines were utilised.

VM Type	Server Name	IP Address
HAProxy	vm-144-6-227-111.rc.tasmania.nectar.org.au	144.6.227.111
Web Server 1	vm-144-6-229-13.rc.tasmania.nectar.org.au	144.6.229.13
Web Server 2	vm-144-6-230-232.rc.tasmania.nectar.org.au	144.6.230.232
MySQL Server	vm-144-6-229-241.rc.tasmania.nectar.org.au	144.6.229.241

Vulnerabilities of Previous SMS Version

Some of the key vulnerabilities of the previous SMS version are the following:

1. The website did not use an SSL certificate. The traffic would go through port 80 and was not secure. A load Balancer was not implemented which means any kind of unavailability in one server would cause the whole website to go down. The website was not scalable.
2. The website did not have any substantial protection from SQL Injection attack. For example, the website allows an attacker to bypass authentication (Figures 2, 3 and 4).

The reason why this code works is that there was no way for the system to distinguish between the user input and SQL queries written in the code.

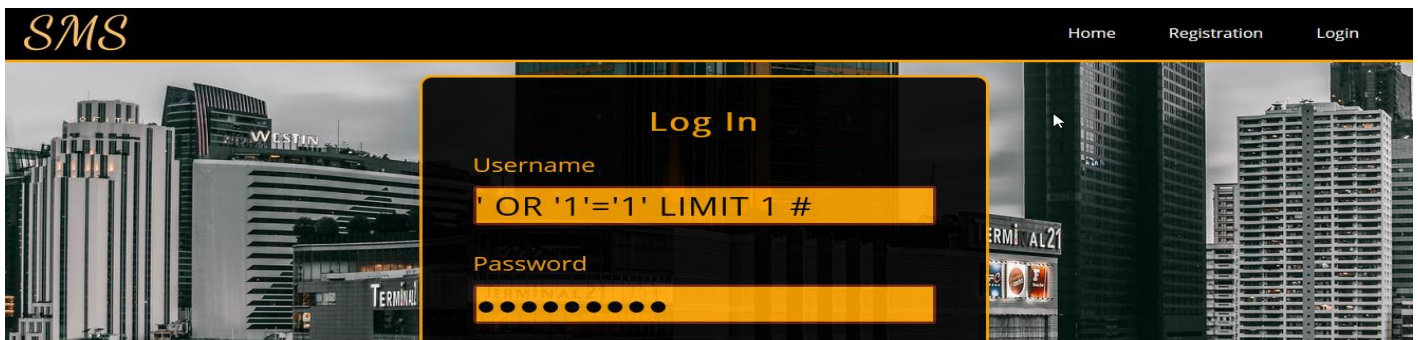


Figure 2: Typing ' OR '1'='1' LIMIT 1 # would allow a user to bypass authentication

```
user.php
Model > user.php
1 k?php
2 class User {
3     public function __construct() {
4     }
5     public function getValidation($username, $password) {
6         global $mysqli;
7         $mysqli = Self::dboneconn($mysqli);
8         $result = $mysqli->query("SELECT username FROM user WHERE username LIKE '$username' AND password LIKE '$password'");
9         $result_cnt_dbone = $result->num_rows;
10        $mysqli = Self::dbtwoconn($mysqli);
11        $result = $mysqli->query("SELECT username FROM user_old WHERE username LIKE '$username' AND password LIKE '$password'");
12        $result_cnt_dbtwo = $result->num_rows;
13        if ($result_cnt_dbone == 1 || $result_cnt_dbtwo == 1) {
```

Figure 3: Vulnerable code for SQL Query in line 8 of the Model/user.php file

When the attacker writes the code ' OR '1'='1' LIMIT 1 # in the user input it will replace the code as the image in Figure 4.

```
6 $mysqli;
7 $li = Self::dboneconn($mysqli);
8 $lt = $mysqli->query("SELECT username FROM user WHERE username LIKE ' ' OR '1'='1' LIMIT 1 # ' AND password LIKE '$password'");
9 $lt_cnt_dbone = $result->num_rows;
10 $li = Self::dbtwoconn($mysqli);
```

Figure 4: SQL Query meaning changes when the attacker injects code in username input

This will return the first row of the user table (as it is set to limit 1) and use that user information to bypass authentication. As observed in the image, the symbol '#' comments out the rest of the query.

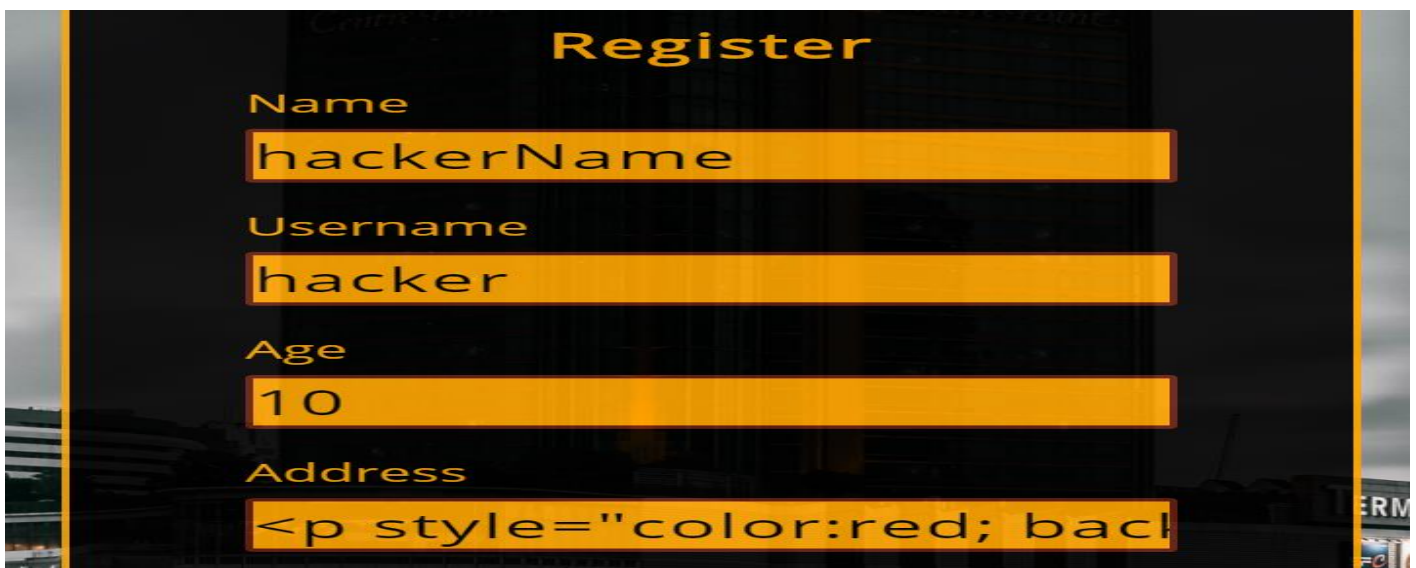
3. No protection from Cross-Site Scripting Software attacks. The website did not have any filtering mechanism to prevent a user from entering a script inside the user inputs (Figure 5).

```
Controller > controller.php
11 private $view;
12 public function __construct() {
13     $this->view = new View(NULL,NULL);
14 }
15 public function execute() {
16     if (isset($_GET['action']) && !empty($_GET['action'])) {
17         $this->{$_GET['action']}();
18     }
19     else if (isset($_POST['loginsubmit'])) {
20         $username = $_POST['username'];
21         $password = $_POST['loginpassword'];
22         $this->loginProcess($username, $password);
23     } else if (isset($_POST['registersubmit'])) {
24         $name = $_POST['name'];
25         $username = $_POST['username'];
26         $age = $_POST['age'];
27         $address = $_POST['address'];
28         $password = $_POST['regpassword'];
29         $email = $_POST['email'];
30         $dbchoice = $_POST['dboption'];
31         $this->registrationProcess($name, $username, $age, $address, $password, $email, $dbchoice);
32     } else if (isset($_POST['purchasesubmitnew'])) {
33         $itemID = $_POST['itemID'];
34         $this->purchaseProcessNew($itemID);
35     } else if (isset($_POST['purchasesubmitold'])) {
36         $itemID = $_POST['itemID'];
37         $this->purchaseProcessOld($itemID);
38     } else if (isset($_POST['searchsubmit'])) {
39         $searchtext = $_POST['searchtext'];
40         $this->searchProcess($searchtext);
41     } else {
42         $this->homepage();
43     }
```

Figure 5: Values passed by POST method to Controller/controller.php are stored insecurely

This would allow the attacker to store code or script inside the database which, if retrieved later and displayed in the browser, would execute the script. This is called Stored XSS (Figure 6).

If the hacker wants to store HTML code or any other script in the user table in the SMS database in the database through the registration page (as the user inputs have no filtering), he will be able to do so.



The screenshot shows a web form titled "Register" with the following fields and values:

- Name: hackerName
- Username: hacker
- Age: 10
- Address: `<p style="color:red; background-color:black; font-size:330px"> This is Hacker.</p>`

Figure 6: Hacker entering code through Address field in the register form. Inserted code was `<p style="color:red; background-color:black; font-size:330px"> This is Hacker.</p>`

This is already considered as a vulnerability and the hacker should never be able to enter code inside the database through user input. However, another vulnerability is when displaying the data in the web browser, the PHP code of the vulnerable version of the SMS website does not apply any filter mechanisms (Figure 7).

```
php
ody>';
for ($x = 0; $x < sizeof($result_array); $x++) {
    if (isset($result_array[$x]["type"]) && isset($this->text)) {
        $this->text .= '
        <tr>
        <td> . $result_array[$x]["item_id"] . '</td>
        <td> . $result_array[$x]["item_name"] . '</td>
        <td> . $result_array[$x]["seller"] . '</td>
        <td> . $result_array[$x]["price"] . '</td>
        <td> . $result_array[$x]["type"] . '</td>
        <td> . $result_array[$x]["man_year"] . '</td>
        <td> . $result_array[$x]["availability"] . '</td>
        <form action="./index.php" method="POST">
        <input type="hidden" name="itemID" value="
        <td><input type="submit" name="purchase
        </form>
        </tr>
    }
    if (!isset($result_array[$x]["type"]) && !isset($this->text)) {
        $this->text .= '
        <tr>
        <td> . $result_array[$x]["item_id"] . '</td>
        <td> . $result_array[$x]["item_name"] . '</td>
        <td> . $result_array[$x]["seller"] . '</td>
        <td> . $result_array[$x]["price"] . '</td>
        <td></td>
        <td></td>
        <td> . $result_array[$x]["availability"] . '</td>
        <form action="./index.php" method="POST">
        <input type="hidden" name="itemID" value="
        <td><input type="submit" name="purchasesub
        </form>
    }
}

//Code added for xss demonstration.
$this->username = $_SESSION["username"];
global $mysqli;
$mysqli = new mysqli('144.6.227.111', 'sm
if (mysqli_connect_errno()) {
    printf("Connect failed: %s\n", mysqli
    exit();
}
$result = $mysqli->query("SELECT * FROM u

while ($row = $result->fetch_assoc())
{
    $this->username = $row["username"]
    $this->address = $row["address"];
}

return '<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=de
<title>login</title>
```

Figure 7: Data from the database are not filtered while showing to the user inside View/searchView.php

The consequences are that the hacker's code "<p style="color:red; background-color:black; font-size:330px"> This is Hacker.</p>" (which is stored in the database now in the address field) is executed when the account is accessed (Figure 8). Furthermore, if the hacker writes "<script> window.location = \"https://www.hackersexamplewebsite.com\"; </script>", it takes to the hacker's website when the page is loaded. The hacker could steal session id by running script.

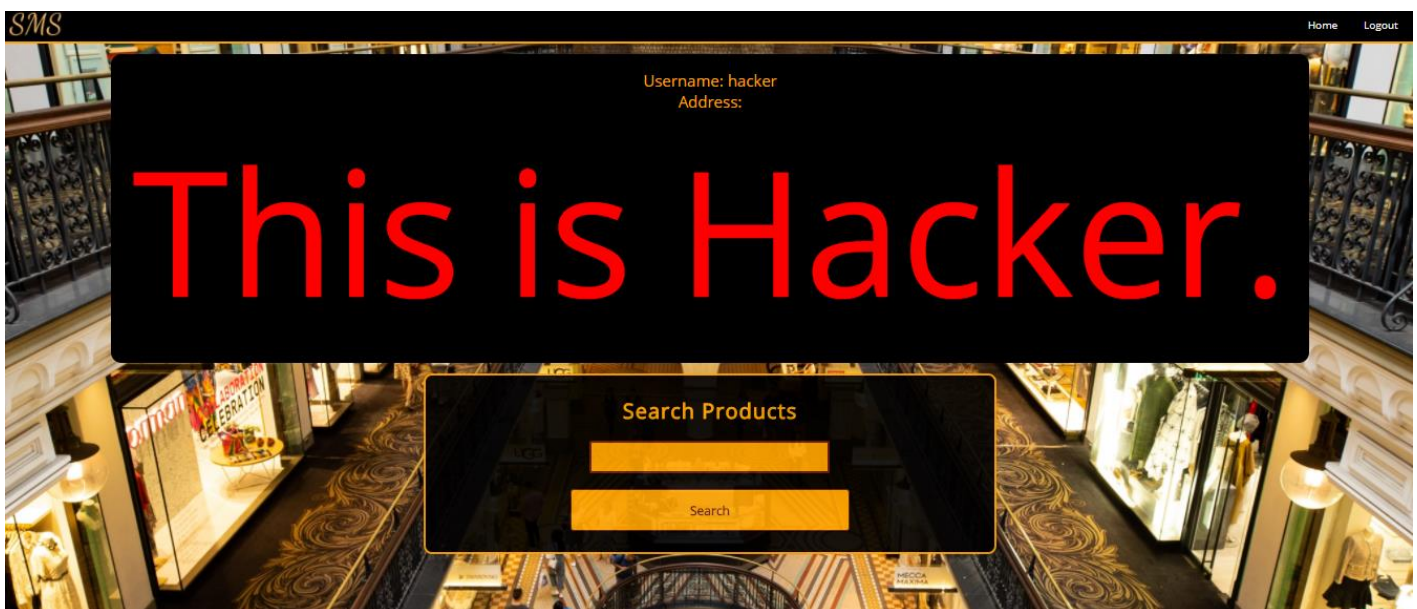


Figure 8: User is not safe when code stored inside the database can execute while running the website

Preventing User from Using HTTP

To solve the first problem mentioned above, the HAProxy load balancer inside the improved SMS is configured to use an SSL certificate and to redirect the user to the HTTPS automatically.

```
frontend HAProxyServer
    bind *:443 ssl crt /etc/ssl/xip.io/xip.io.pem
    redirect scheme https if !{ ssl_fc }
    mode http
    bind 144.6.227.111:80
    default_backend LAMP_WebServers
backend LAMP_WebServers
    mode http
    balance roundrobin
    option forwardfor
    option httpchk HEAD / HTTP/1.1\r\nHost:localhost
    server vm-144-6-229-13.rc.tasmania.nectar.org.au 144.6.229.13:80 check
    server vm-144-6-230-232.rc.tasmania.nectar.org.au 144.6.230.232:80 check

    http-request set-header X-Forwarded-Port %[dst_port]
    http-request add-header X-Forwarded-Proto https if { ssl_fc }
listen stats
    bind 144.6.227.111:9999
    stats enable
    stats show-node
    stats uri /stats
```

Figure 9: HAProxy redirecting user to HTTPS in the frontend settings

Preventing SQL Injection

There are 3 key prevention techniques for SQL Injection implemented in the improved SMS.

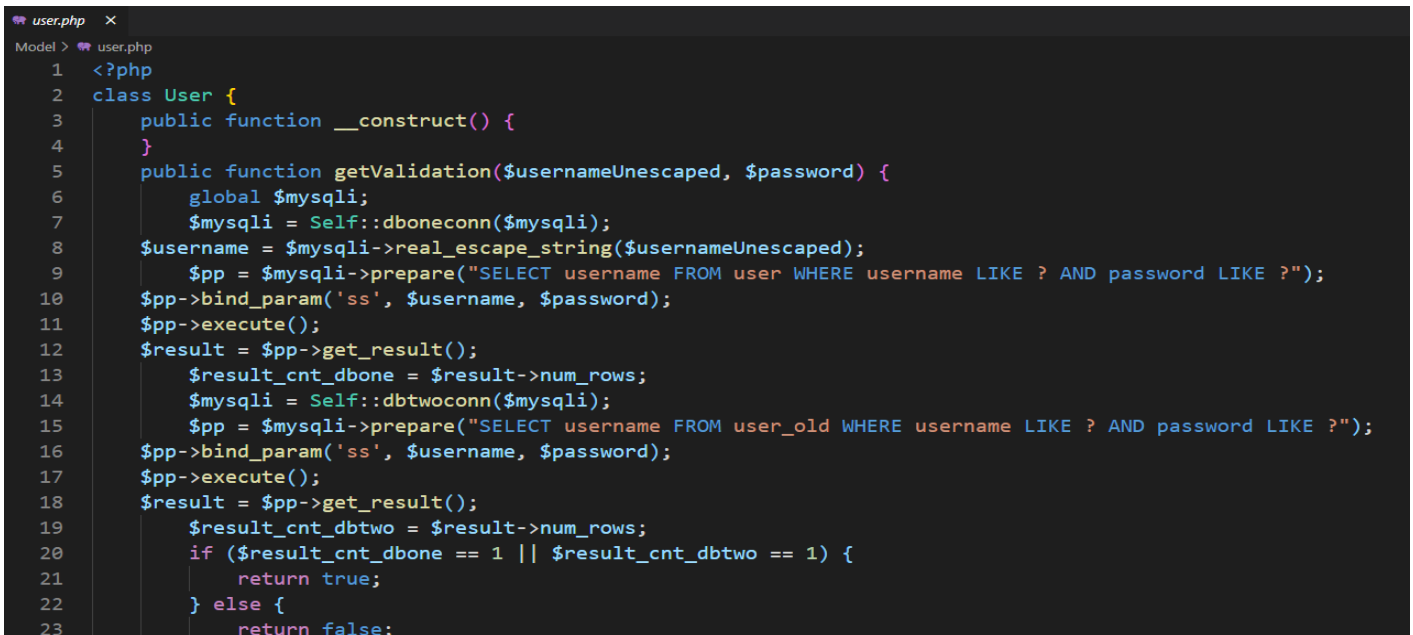
1. Special Characters Escaping. This technique will escape special characters while storing user input inside a variable. This way while using the variable in a SQL query, no code can be executed (Figure 10).

```
user.php x
Model > user.php
1  <?php
2  class User {
3      public function __construct() {
4      }
5      public function getValidation($usernameUnescaped, $password) {
6          global $mysqli;
7          $mysqli = Self::dbconn($mysqli);
8          $username = $mysqli->real_escape_string($usernameUnescaped);
9          $pp = $mysqli->prepare("SELECT username FROM user WHERE username LIKE ? AND password LIKE ?");
```

Figure 10: Escaping special characters in Model/user.php to prevent bypassing authentication

This solved the problem discussed in the vulnerability section. The code ' OR '1'='1' LIMIT 1 # does not work anymore to bypass authentication. Additionally, the second prevention mechanism which is reported next would also protect the website from this very problem.

2. Parameterized Query sends the query to the database and then simply binds the data separately. This way, the bound data and query itself are seen separately by the system (Figure 11).



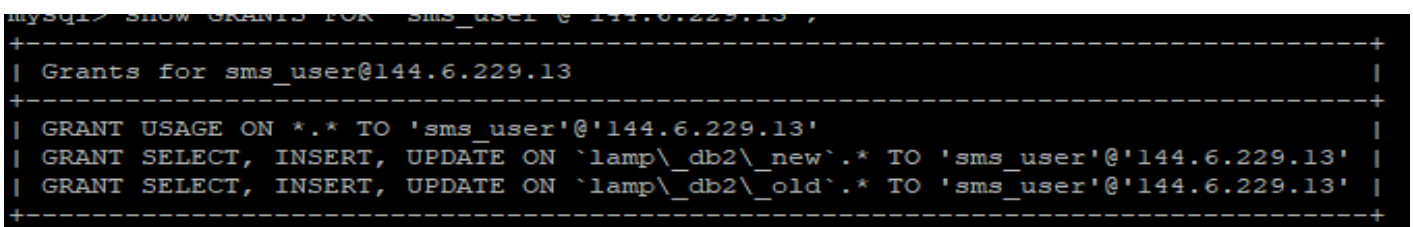
```
1 <?php
2 class User {
3     public function __construct() {
4     }
5     public function getValidation($usernameUnescaped, $password) {
6         global $mysqli;
7         $mysqli = Self::dboneconn($mysqli);
8         $username = $mysqli->real_escape_string($usernameUnescaped);
9         $pp = $mysqli->prepare("SELECT username FROM user WHERE username LIKE ? AND password LIKE ?");
10        $pp->bind_param('ss', $username, $password);
11        $pp->execute();
12        $result = $pp->get_result();
13        $result_cnt_dbone = $result->num_rows;
14        $mysqli = Self::dbtwoconn($mysqli);
15        $pp = $mysqli->prepare("SELECT username FROM user_old WHERE username LIKE ? AND password LIKE ?");
16        $pp->bind_param('ss', $username, $password);
17        $pp->execute();
18        $result = $pp->get_result();
19        $result_cnt_dbtwo = $result->num_rows;
20        if ($result_cnt_dbone == 1 || $result_cnt_dbtwo == 1) {
21            return true;
22        } else {
23            return false;
```

Figure 11: Parameterized Queries are implemented throughout the SMS source code to prevent most kinds of SQL injection

3. Lastly, to stop attackers executing SQL commands which should never be executed through the website, Least Privilege Settings measures are followed (Figures 12 and 13).

Note: The host of the sms_user was later changed to suit two VMs.

These least privilege measures would protect the database from DROP, ALTER, or any other commands that would otherwise cause a lot of damage. For example, the attacker could insert the following code and it would drop the user table in the SMS database.



```
mysql> show GRANTS FOR 'sms_user'@'144.6.229.13';
+-----+
| Grants for sms_user@144.6.229.13 |
+-----+
| GRANT USAGE ON *.* TO 'sms_user'@'144.6.229.13' |
| GRANT SELECT, INSERT, UPDATE ON `lamp\_db2\_new`.* TO 'sms_user'@'144.6.229.13' |
| GRANT SELECT, INSERT, UPDATE ON `lamp\_db2\_old`.* TO 'sms_user'@'144.6.229.13' |
+-----+
```

Figure 12: Only necessary operations for the SMS databases are allowed



```
'sms_user'@'144.6.229.13' | def | USAGE | NO |
'lamp_usr2'@'%' | def | USAGE | NO |
+-----+
```

Figure 13: Only USAGE rights are granted to sms_user

SQL Injection Based on Batched SQL Statements

Most databases support batched SQL statement.

A batch of SQL statements is a group of two or more SQL statements, separated by semicolons.

The SQL statement below will return all rows from the "Users" table, then delete the "Suppliers" table.

Example

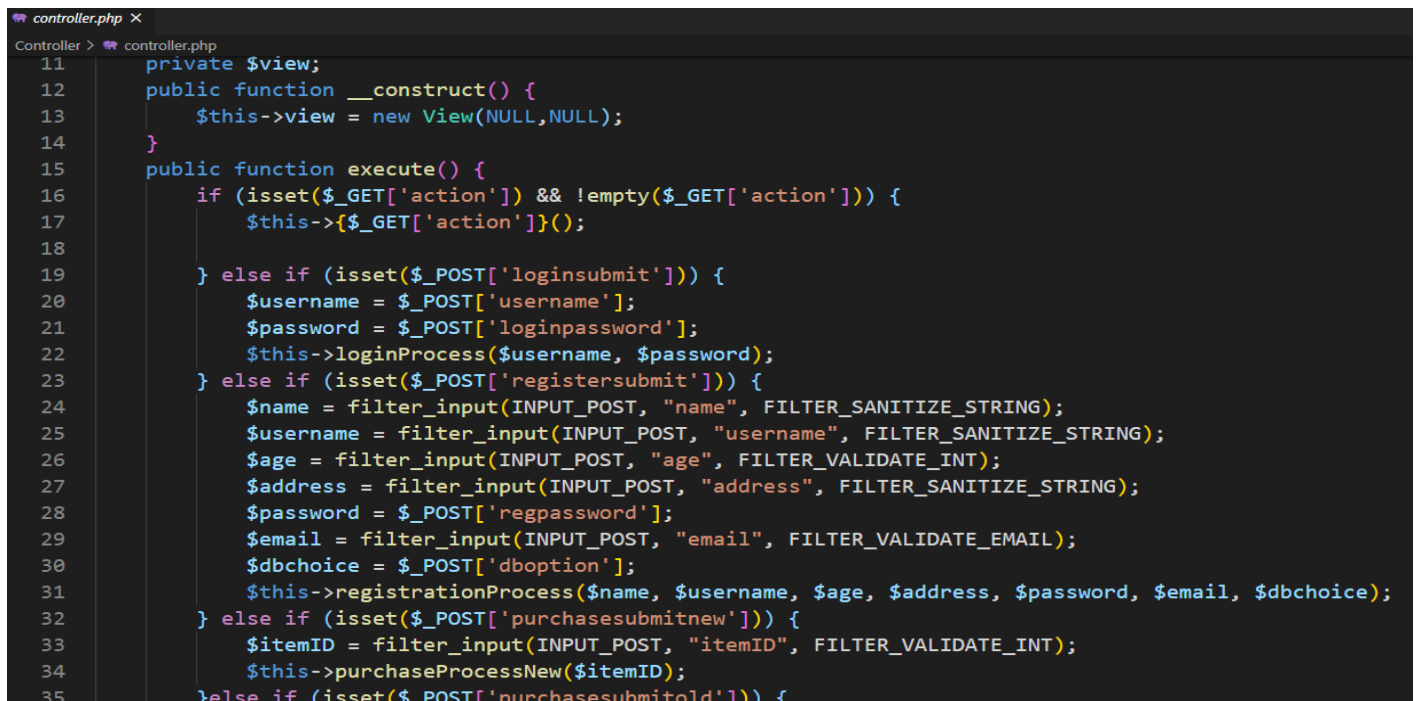
```
SELECT * FROM Users; DROP TABLE Suppliers
```

Figure 14: Without least privilege measures, this attack or similar could be possible (image source:W3Schools.com)

Preventing Cross-Site Scripting Attack

There are two aspects of XSS attack that was focused upon during development.

1. Preventing user to enter scripts or any code through forms. To achieve this, several filtering techniques were utilised (Figure 15).



```
controller.php X
Controller > controller.php
11 private $view;
12 public function __construct() {
13     $this->view = new View(NULL,NULL);
14 }
15 public function execute() {
16     if (isset($_GET['action']) && !empty($_GET['action'])) {
17         $this->{$_GET['action']}();
18     }
19     else if (isset($_POST['loginsubmit'])) {
20         $username = $_POST['username'];
21         $password = $_POST['loginpassword'];
22         $this->loginProcess($username, $password);
23     } else if (isset($_POST['registersubmit'])) {
24         $name = filter_input(INPUT_POST, "name", FILTER_SANITIZE_STRING);
25         $username = filter_input(INPUT_POST, "username", FILTER_SANITIZE_STRING);
26         $age = filter_input(INPUT_POST, "age", FILTER_VALIDATE_INT);
27         $address = filter_input(INPUT_POST, "address", FILTER_SANITIZE_STRING);
28         $password = $_POST['regpassword'];
29         $email = filter_input(INPUT_POST, "email", FILTER_VALIDATE_EMAIL);
30         $dbchoice = $_POST['dboption'];
31         $this->registrationProcess($name, $username, $age, $address, $password, $email, $dbchoice);
32     } else if (isset($_POST['purchasesubmitnew'])) {
33         $itemID = filter_input(INPUT_POST, "itemID", FILTER_VALIDATE_INT);
34         $this->purchaseProcessNew($itemID);
35     } else if (isset($_POST['purchasesubmitold'])) {
```

Figure 15: Filtering of the data received through POST method inside Controller/controller.php

2. The second technique was used to escape retrieved data (from the database) while showing it to the user. The 'htmlspecialchars(\$string, \$allowable_tags)' was implemented to achieve the desired level of security (Figure 16).

```
78 </thead>
79 <body>;
80     for ($x = 0; $x < sizeof($result_array); $x++) {
81         if (isset($result_array[$x]["type"]) && isset($result_array[$x]["man_year"])) {
82             $this->text.= '
83                 <tr>
84                     <td>' . htmlspecialchars($result_array[$x]["item_id"], ENT_QUOTES) . '</td>
85                     <td>' . htmlspecialchars($result_array[$x]["item_name"], ENT_QUOTES) . '</td>
86                     <td>' . htmlspecialchars($result_array[$x]["seller"], ENT_QUOTES) . '</td>
87                     <td>' . htmlspecialchars($result_array[$x]["price"], ENT_QUOTES) . '</td>
88                     <td>' . htmlspecialchars($result_array[$x]["type"], ENT_QUOTES) . '</td>
89                     <td>' . htmlspecialchars($result_array[$x]["man_year"], ENT_QUOTES) . '</td>
90                     <td>' . htmlspecialchars($result_array[$x]["availability"], ENT_QUOTES) . '</td>
91                 <form action="./index.php" method="POST">
92                     <input type="hidden" name="itemID" value="' . $result_array[$x]["item_id"] . '">
30                 while ($row = mysqli_fetch_assoc($result)) {
31                     $this->username = htmlspecialchars($row["username"], ENT_QUOTES);
32                     $this->address = htmlspecialchars($row["address"], ENT_QUOTES);
33                 }
34             }
35     // Above code in this output() function is only for XSS demonstration. And,
```

Figure 16: Escaping data before displaying data to the user in View/searchView.php

HAProxy Statistics (HTTPS & TCP)

← → ↻ ⚠ Not secure | http://144.6.227.111:9999/stats

HAProxy version 2.0.17-1ppa1~xenial, released 2020/08/01

Statistics Report for pid 2716 on pvfjokdlcuoe1

> General process information

pid = 2716 (process #1, nproc = 1, nthread = 1)
uptime = 0s 0m3m45s
system limits: memmax = unlimited; ulimit-n = 4095
maxsock = 4095; maxconn = 2027; maxpipes = 0
current conn = 1; current pipe = 0/0; conn rate = 1/sec; bit rate = 0.543 kbps
Running tasks: 2/15; idle = 100 %

active UP backup UP
active UP, going down backup UP, going down
active DOWN, going up backup DOWN, going up
active or backup DOWN not checked
active or backup DOWN for maintenance (MAINT)
active or backup SOFT STOPPED for maintenance
Note: "NOLOAD/NO DRAIN" = UP with load-balancing disabled.

Display option:

Scope:
• Hide DOWN servers
• Refresh now
• CSV export

External resources:
• Primary site
• Updates (v2.0)
• Online manual

HAProxy Server		Queue			Session rate			Sessions					Bytes				Denied		Errors		Warnings		Server										
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntime	Thrtle		
Frontend					0	444	-	0	2 027	2 027	11 889			2 997 806	19 399 813	0	0	0					OPEN										
					Max connection rate: 1050/s																												
					Max session rate: 444/s																												
					Max request rate: 901/s																												
LAMP_WebServers								Session rate			Sessions					Bytes				Denied		Errors		Warnings		Server							
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntime	Thrtle		
vm-144-6-229-13.rc.tasmania.nectar.org.au		0	0	-	0	451	0	298	-	10 950	10 946	1m54s	1 272 374	8 991 100	0	0	0	0	4	0	2m41s UP	L7OK/302 in 2ms	1	Y	-	0	1	0s	-				
vm-144-6-230-232.rc.tasmania.nectar.org.au		0	0	-	0	450	0	275	-	10 922	10 922	1m54s	1 288 608	10 108 816	0	0	0	0	0	2m41s UP	L7OK/302 in 3ms	1	Y	-	0	1	0s	-					
Backend		0	0	0	901	0	573	203	23 225	21 888	1m54s	2 997 806	19 399 813	0	0	1 357	0	4	0	2m41s UP		2	2	0	1	4s							
stats																																	
		Cur	Max	Limit	Cur	Max	Limit	Cur	Max	Limit	Total	LbTot	Last	In	Out	Req	Resp	Req	Conn	Resp	Retr	Redis	Status	LastChk	Wght	Act	Bck	Chk	Dwn	Dwntime	Thrtle		
Frontend					1	1	-	1	1	2 027	1	0	0	0	0	0	0	0					OPEN										
Backend		0	0	0	0	0	203	0	0	0	203	0	0s	0	0	0	0	0	0	0	0	0	3m45s UP		0	0	0	0	0	0			

Figure 17: HAProxy statistics of performance from the stats page

Graph Results (HTTPS)

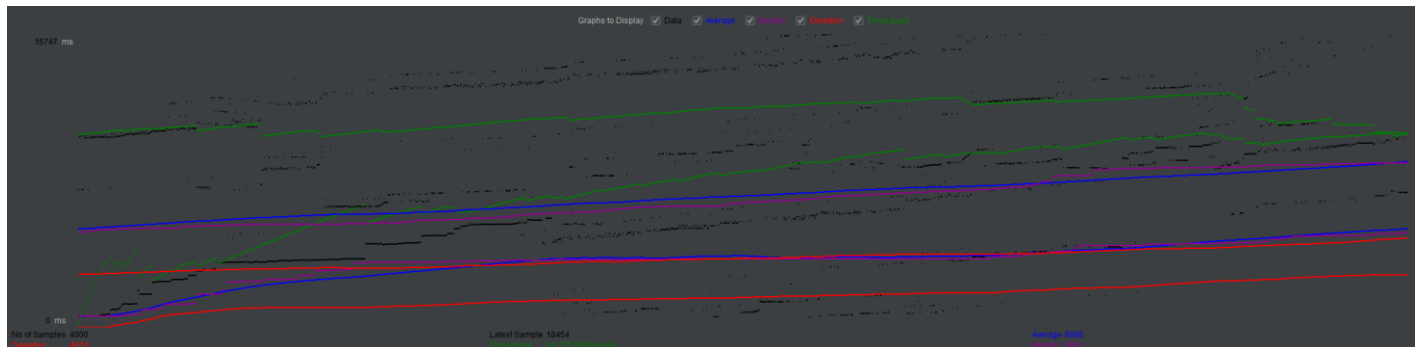


Figure 18: Thread group 1 using constant timer with a thread delay of 1000ms

It can be inferred from the HAProxy statistics and graph results presented above that:

- The average latency response time in milliseconds for each new connection or request is 8908ms(Figure 18)

- The throughput is 37980 bytes/s which is calculated based on:

10,128.292 connections/min / 60 = 168.8 connections/s (Figure 18)

Therefore, based on Figure 17, if 2697806 bytes are transferred over 11989 connections, the number of kilobytes transferred over 168.8 connections = $((2697806/11989) \times 168.8) = 37980$ bytes/s.



Figure 19: Thread group 2 using uniform timer with a thread delay max of 100.0ms and 0ms constant delay offset

It can be inferred from the HAProxy statistics and graph results presented above that:

- The average latency response time in milliseconds for each new connection or request is 15058ms (Figure 19).
- The throughput is 23200 bytes/s which is calculated based on:

$6,210.055 \text{ connections/min} / 60 = 103.5 \text{ connections/s}$ (Figure 19)

Therefore, based on Figure 17, if 2697806 bytes are transferred over 11989 connections, the number of kilobytes transferred over 103.5 connections = $((2697806/11989) \times 103.5) = 23200 \text{ bytes/s}$.

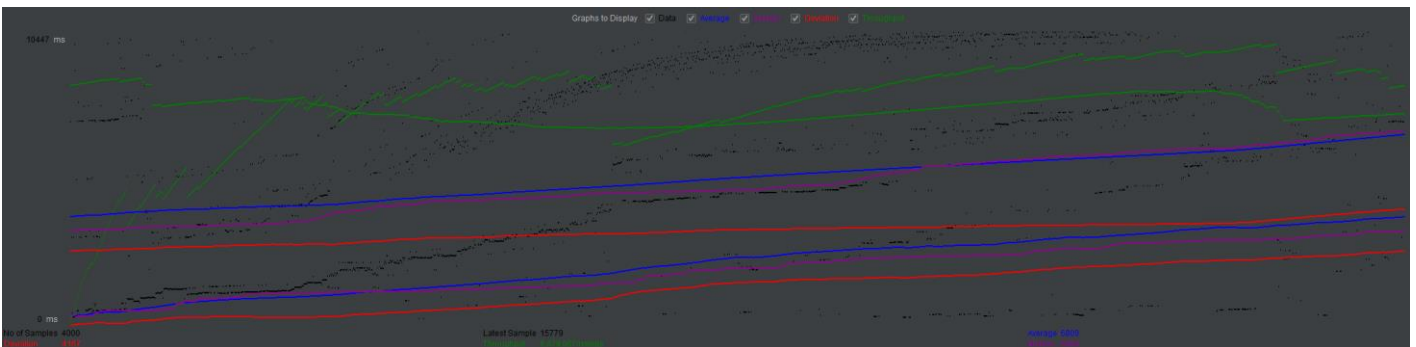


Figure 20: Thread group 3 using gaussian distribution timer with a thread delay deviation of 100.0 ms and constant delay offset of 300ms

It can be inferred from the HAProxy statistics and graph results presented above that:

- The average latency response time in milliseconds for each new connection or request is 6809ms (Figure 20).
- The throughput is 31790 bytes/s which is calculated based on:
 $8,479.067 \text{ connections/min} / 60 = 141.3 \text{ connections/s}$ (Figure 20). Therefore, based on Figure 17, if 2697806 bytes are transferred over 11989 connections, the number of kilobytes transferred over 141.3 connections = $((2697806/11989) \times 141.3) = 31790 \text{ bytes/s}$.

Graph Results (TCP)

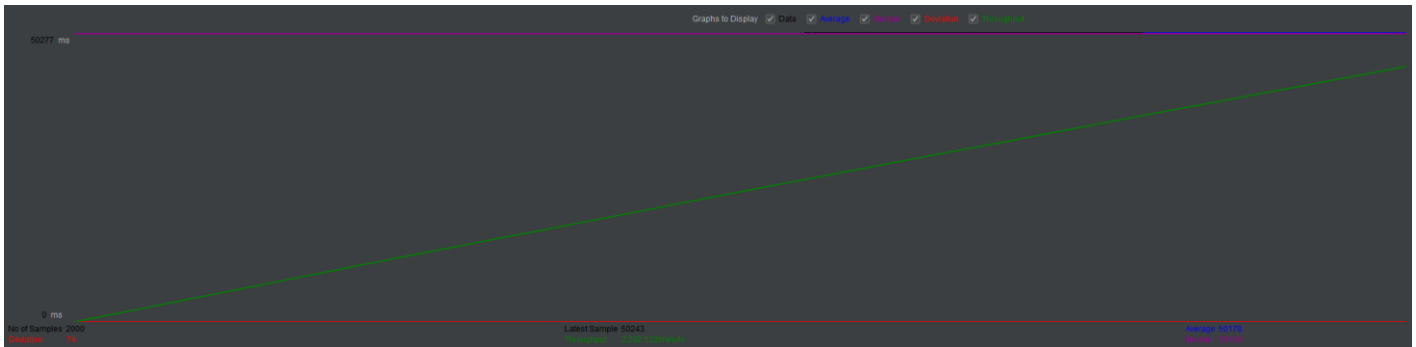


Figure 21: Thread group 1 using constant timer with a thread delay of 1000ms

It can be inferred from the HAProxy statistics and graph results presented above that:

- The average latency response time in milliseconds for each new connection or request is 50178ms (Figure 21).
- The throughput is 8600 bytes/s which is calculated based on:
 $2,292.132 \text{ connections/min} / 60 = 38.2 \text{ connections/s}$ (Figure 21)

Therefore, based on Figure 17, if 2697806 bytes are transferred over 11989 connections, the number of kilobytes transferred over 38.2 connections = $((2697806/11989) \times 38.2) = 8600 \text{ bytes/s}$.

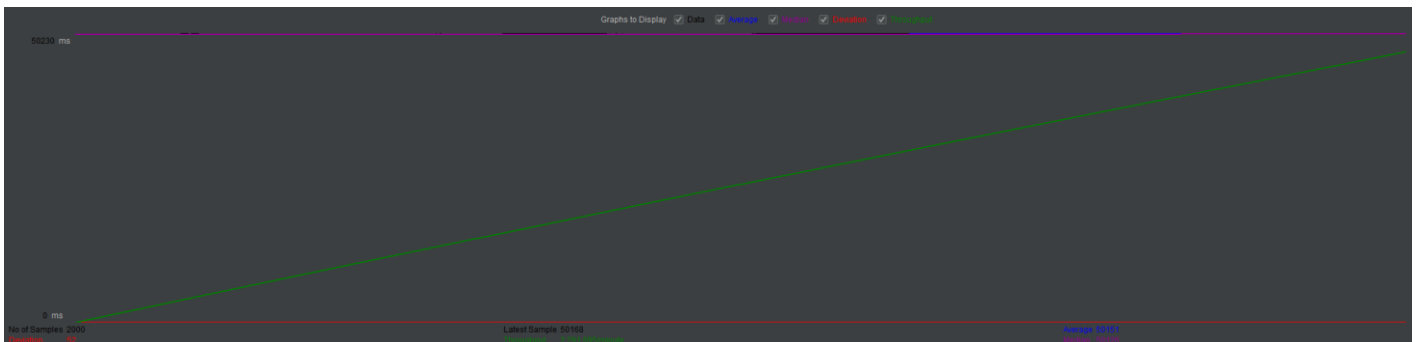


Figure 22: Thread group 2 using uniform timer with a thread delay deviation of 100.0 ms and constant delay offset of 0ms

It can be inferred from the HAProxy statistics and graph results presented above that:

- The average latency response time in milliseconds for each new connection or request is 50151ms (Figure 22).
- The throughput is 8600 bytes/s which is calculated based on:
 $2,291.695 \text{ connections/min} / 60 = 38.19 \text{ connections/s}$ (Figure 22).

Therefore, based on Figure 17, if 2697806 bytes are transferred over 11989 connections, the number of kilobytes transferred over 38.19 connections = $((2697806/11989) \times 38.19) = 8600$ bytes/s.

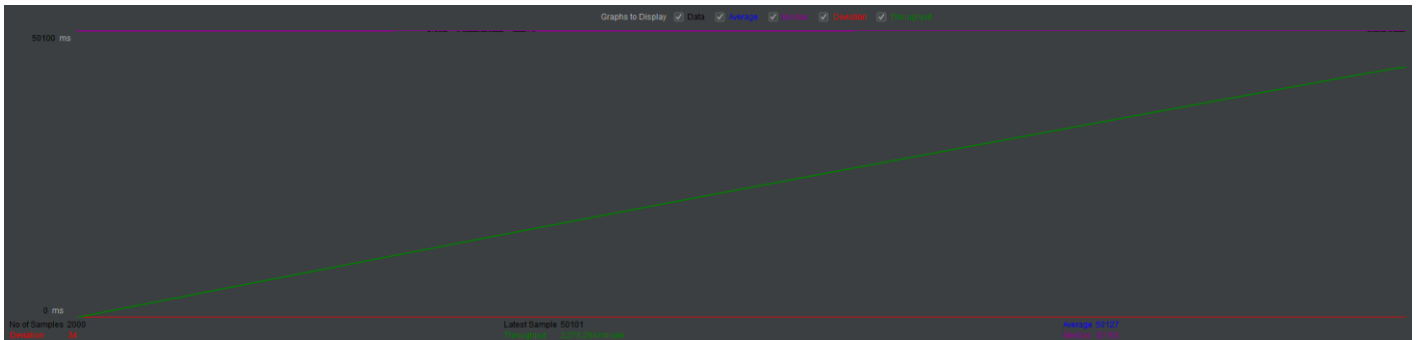


Figure 23: Thread group 3 using gaussian distribution timer with a thread delay deviation of 100.0 ms and constant delay offset of 300ms

It can be inferred from the HAProxy statistics and graph results presented above that:

- The average latency response time in milliseconds for each new connection or request is 50127ms (Figure 23).
- The throughput is 8500 bytes/s which is calculated based on:
 $2,278.294 \text{ connections/min} \times 60 = 37.97 \text{ connections/s}$ (Figure 23)

Therefore, based on Figure 17, if 2697806 bytes are transferred over 11989 connections, the number of kilobytes transferred over 37.97 connections = $((2697806/11989) \times 37.97) = 8500$ bytes/s.

Analysis of the Data

For https:

- Number of requests that can be served is 901/s (Figure 17).
- Latency response time in milliseconds for each new connection or request 10258.3 ms
- Throughput : 30990 bytes/s

For TCP:

- Number of requests that can be served is 901/s (Figure 17).
- Latency response time in milliseconds for each new connection or request 50152ms
- Throughput : 8,566.67 bytes/s

Combining data from both https, tcp tests and HAProxy Statistics:

- Number of requests that can be served is 901/s (Figure 17).
- Latency response time in milliseconds for each new connection or request 30205.167ms
- Throughput : 19,778.3 bytes/s