Article

A tutorial on Cross Site Scripting Attack - Defense

Vassilis Papaspirou [1], Leandros Maglaras [2,*] and Mohamed Amine Ferrag [3]

1 University of Thessaly, Lamia, Greece
2 School of Computer Science and Informatics, De Montfort University, Leicester LE1 9BH, UK
3 Department of Computer Science, Guelma University, Guelma 24000, Algeria
* Correspondence: leandros.maglaras@dmu.ac.uk

Abstract: Cross-site Scripting attacks (XSS attacks) are listed as the top widespread and critical weakness that can be discovered and exploited as software vulnerabilities. When designing web applications programmers and analysts must follow secure coding rules and try not to leave any loopholes. Experience is a great factor and programmers are unable most of the times to spot all the weak points in an online application. In this article we present a tutorial on launching XSS attacks along with the mitigation actions. The article covers both theoretical and technical aspects of XSS attacks and can be used as a self-learning or teaching tool for security students or professionals.

Keywords: XSS; Cross site scripting; Sql injection

---

1. Introduction

Nowadays the World Wide Web (WWW) has been changed to a multifaceted system by incorporating a wide assortment of parts and advancements including client side advances [1], serverside advances [2], HTTPs Protocol and wide assortment of different innovations. Web applications created on these stages try to cope with a wide range of clients, offering to them rich highlights of these cutting edge innovations. Existing vulnerabilities among these advances present the challenge of applying those protective safety methods for the improvement of web applications. Albeit, current protective measures offer limited support to modern web applications [3].

Accordingly, a high part of Internet based web applications are defenseless against many vulnerabilities. White Hat Security's Website Security Statistics Report [4] offer a sort of recognition on the present issues of security of web applications and worries that enterprises should bargain for playing out the online business in a protected manner. This webiste has been circulating the security bits of knowledge report on the WWW since year 2006. They have examined vulnerabilities on popular applications: Managing an account, Monetary Administrations, Wellbeing Care, Protections and Retail. They have broken down the vulnerabilities of those independently. In light with some parameters they set up a score card for these spaces:

- Always Vulnerable
- Frequently Vulnerable
- Regularly Vulnerable
- Occasionally Vulnerable +
- Rarely Vulnerable

Common Weakness Enumeration (CWE™) is a view to the top most most dangerous software errors – weaknesses (CWE). The CWE list is maintained by MITRE Corporation[5] and it has XSS attack as the top widespread and critical weakness that can be discovered and exploited as software vulnerabilities. XSS Attacks are simple attacks, since it's very simple to find exploitable vulnerabilities on modern websites. In general viably anticipating XSS vulnerabilities is likely to include a combination of several measures including static testing, code audits, and dynamic testing along with applying secure coding techniques.

In this paper we present 2 concrete examples of XSS attacks (Javascript attack and SQL injection). JavaScript injection is a process by which we can insert and use our own JavaScript code in a page, either by entering the code into the address bar, or by finding an XSS vulnerability in a website. SQL infusion may be a web security helplessness that permits an assailant to meddled with the questions that an application makes to its database. It permits an attacker to see information that they are not regularly able to recover. This might incorporate information having a place to other clients, or any other information that the application itself is able to access. In numerous cases, an aggressor can alter or erase this information, causing tireless changes to the application's substance or behavior. In a few circumstances, an aggressor can raise an SQL infusion assault to compromise the basic server or other back-end framework, or perform a denial-of-service attack.

The contributions of the paper are:

- It presents theoretical and technical information about XSS vulnerabilities and attacks
- It analytically presents simple scenarios of XSS attacks in form of javascript and SQL attacks
- It is accompanied by code that is uploaded on GitHub:

  https://github.com/vapapaspirou/javascript-and-sqlinjection
- It can be used as a self-learning or teaching tool for security students or professionals

The rest of the paper is structured as follows. Section 2 presents XSS attack concept. Section 3 presents the different types of XSS attacks. Section 4 presents the javascript attack-defence tutorial example. Section 5 presents the SQL injection attack-defence tutorial example> Finally Section 6 concludes the paper.

2. What is XXS

Cross-scripts (also called XSS) are internet protection vulnerabilities that enable an attacker to use users' interactions with an application. It allows an attacker to ignore the security policy, which is meant to differentiate distinct websites from one another. Typically, inter-site script vulnerabilities permit an attacker to hide himself as a user, operate any moves that the consumer might also perform, and get right of entry to any of the information of the user. Although there is no single, standardized classification for XSS attacks they may be classified in 3 types. At least two primary sectors of XSS flaws may be distinguished: non-persistent and persistent [6]. Some sources further divide these two groups into traditional (caused by server-side code flaws) and DOM-based (in client-side code).

Non-persistent XSS vulnerabilities in a web application seem to permit malevolent destinations to attack its clients who are utilizing the app while being logged within. The non-persistent cross-site scripting vulnerability is by far the foremost fundamental sort of web vulnerability. These gaps appear up when the information is given by a web client, most commonly in HTTP inquiry parameters (e.g. HTML shape submission), and have impacts instantly in the event that the server-side of the net app parses and shows a page to the client, without properly sanitizing it.

Since HTML archives have a level, serial structure that blends control statements, formatting, and the genuine substance, any non-validated user-supplied information included within the resulting page without legitimate HTML encoding, may lead to markup injection [7]. A reflected attack is typically conveyed through e-mail or an impartial web location. The trap is an innocent-looking URL, pointing to a trusted location but containing the XSS vector. In the event that the trusted location is defenseless to the vector, clicking the interface can cause the victim's browser to execute the infused script.

The determined (or put away) XSS vulnerability could be a more annihilating variation of a cross-site scripting imperfection: it happens when the information given by the aggressor is spared by the server, and then permanently shown on "normal" pages returned to other clients within the course of regular browsing. A classic illustration is on online message boards where clients are permitted to post HTML designed messages for other clients to read. Persistent XSS vulnerabilities can be more critical than other sorts since an attacker's malicious script is rendered naturally, without the goal to independently target victims or bait them to a third-party site [8]. Especially within social networks, the code would self-propagate over accounts, making a sort of client-side worm [9].

The strategies of injection can shift a awesome bargain; in a few cases, the aggressor may not even need to specifically associate with the vulnerability itself to misuse such a hole. Any data received by the net application (by means of e-mail, framework logs, IM etc.) that can be controlled by an attacker may become an attack vector.

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

The noxious substance sent to the internet browser frequently appears as a portion of JavaScript, however may likewise incorporate HTML, Flash, or some other sort of code that the program may execute. The assortment of attacks dependent on XSS is practically boundless, however they normally incorporate transmitting private information, similar to treats or other meeting data, to the vivtim, diverting the casualty to web content constrained by the aggressor, or performing different malevolent procedure on the client's machine under the appearance of the legitimate website.

XSS attacks can also be categorized into two groups: stored and reflected. There is a third, much less well-known type of XSS attack called DOM Based XSS[10].

3. Types of Attack

3.1. Stored XSS Attacks

As mentioned in the previous section, in stored attacks the injected script is stored permanently on the target. This can be a database, a forum, a visitor log, several comment fields among other. The victim that visits the infected target and requests some information, retrieves the malicious scripts from the server. Stored XSS is reffered as Persistend threats.

3.2. Reflected XSS Attacks

Reflected assaults are those where the implanted substance is reflected off the net server as a response that consolidates some of the data sent to the server. Reflected attacks are passed on to casualties using other routes, for example, in an e-mail message. At the point when a client is tricked into tapping on a malignant link or indeed basically examining to a pernicious site, the embedded code mirrors the attack back to the client's program. The program at that point executes the code since it originated from a "trusted" server. Reflected XSS to boot a few of the time implied to as Non-Persistent or Type-II XSS.

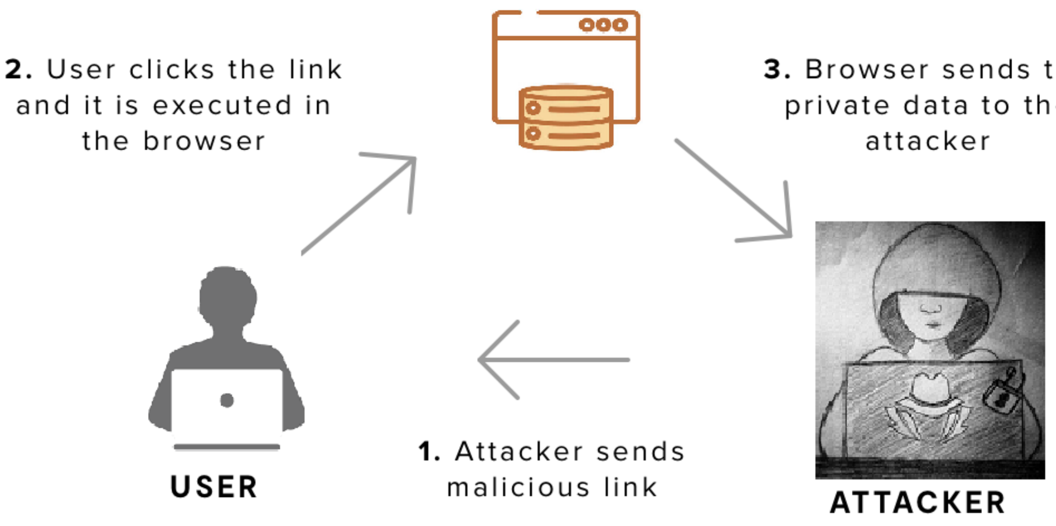

2. User clicks the link and it is executed in the browser
3. Browser sends the private data to the attacker
USER
1. Attacker sends malicious link
ATTACKER

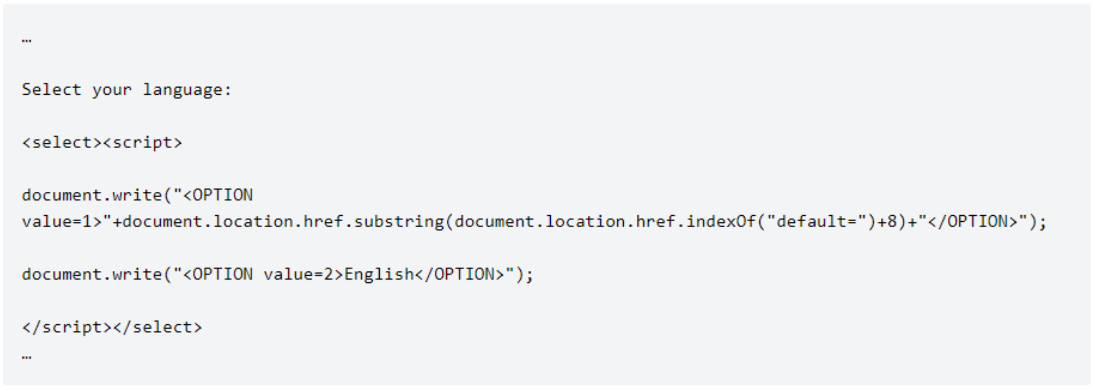
Figure 1. Reflected attack XSS.

3.3. DOM Based XSS

DOM Based XSS (or because it is called in a few writings, "type-0 XSS") is an XSS assault wherein the payload is executed as a result of modifying the DOM "environment" within the victim's browser utilized by the first client side script, so that the client side code runs in an "unexpected" way. That's , the page itself does not alter, but the client side code contained within the page executes in an unexpected way due to the malevolent adjustments that have happened within the DOM environment. This is in differentiate to other XSS assaults, wherein the payload is set within the page (due to a server side attack).

Example

Suppose the following code is used to create a form to let the user choose his/her preferred language. A default language is also provided in the query string, as the parameter "default".



```
…

Select your language:

<select><script>

document.write("<OPTION
value=1>"+document.location.href.substring(document.location.href.indexOf("default=")+8)+"</OPTION>");

document.write("<OPTION value=2>English</OPTION>");

</script></select>
…
```

Figure 2. Example Code for creation a form.

The URL for the page is:

http://www.some.site/page.html?default=French

If we send the following URL to a victim, then an attack(DOM Based XSS) can be executed to the page.

http://www.some.site/page.html?default=<script>alert(document.cookie)</script>

The browser sends a request for a page which has an alert (document cookie) in the URL. This happens the moment the victim clicks on the link.

/page.html?default=<script>alert(document.cookie)</script>

After the above move takes action, the server responds with the Javascript code which is in the page. A DOM object is created by the browser for the page, where the document.location object contains the URL:

http://www.some.site/page.html?default=<script>alert(document.cookie)</script>

The initial Javascript code within the page does not anticipate the default parameter to contain HTML markup, and as such it essentially echoes it into the page (DOM) at runtime. The browser at that point executes the attacker's script.

The default parameter which contain HTML markup is not expected from the original Javascript code which is in the page, so it returns it into the page (DOM) at runtime. The resulting page finally executes the script from the attacker.

alert(document.cookie)

Preventing cross-site scripting is trivial in some instances can be difficult depending on the complexity of the application and the ways it handles user-controllable data. In general, successfully preventing XSS vulnerabilities is probable to contain a combination of the following measures:

- Filter enter on arrival. At the point where consumer input is received, filter as strictly as feasible based on what is predicted or legitimate input.
- Encode information on output. At the point where user-controllable records is output in HTTP responses, encode the output to stop it from being interpreted as energetic content. Depending on the output context, this would possibly require making use of combos of HTML, URL, JavaScript, and CSS encoding.
- Use gorgeous response headers. To stop XSS in HTTP responses that don't seem to be meant to contain any HTML or JavaScript, you can use the Content-Type and X-Content-Type-Options headers to make sure that browsers interpret the responses in the way you intend.
- Content Security Policy. As a last line of defense, you can use Content Security Policy (CSP) to reduce the severity of any XSS vulnerabilities that still occur.

Encode data on output

Before client-controllable information is kept in touch with a website, encoding can be implemented easily, on the basis that the environment you are writing examines what kind of encoding you need to use. Values interior a JavaScript string, for instance, require a choice of getting away from those in an HTML environment. In an HTML context, you convert non-whitelisted values into HTML entities:

$$< convertsto : \&lt;$$

$$> convertsto : \&gt;$$

In a JavaScript string context, non-alphanumeric values should be Unicode-escaped:

$$< convertsto : \backslash u003c$$

$$> convertsto : \backslash u003e$$

You'll have to add different layers of encoding at the right request. For instance, you have to manage both the JavaScript setting and the HTML setting to securely include client contribution within an occasion handler. So you have to get Unicode-away from the data first, and then HTML-encode it:

<a href="#" onclick="x='This string requires two layers of escaping'">test</a>

Validate entry on arrival:

Encoding is probably the most critical XSS barrier line, but it is not sufficient in each particular situation to avoid XSS vulnerabilities. You should also authorize the input as carefully as possible, specifically when it is first obtained from a client. Input validation examples include:

- If a user submits a URL that is returned in a reply, verify that it starts offevolved with a tightly closed protocol such as HTTP and HTTPS. Otherwise someone may use a malicious pr to hack your website online
- Validating that the value actually includes an integer, if a user presents a value that was supposed to be binary.
- The validation of the input requires only the expected character collection. Input validation by way of blocking off invalid enter ought to preferably work. An alternative method is extra inclined to error, attempting to easy invalid input to make it valid, and be avoided where feasible

Whitelisting vs blacklisting:

Validation of inputs should usually use whitelists rather than blacklists. For example, just make a list of secure protocols (HTTP, HTTPS) instead of trying to make a list of all harmful protocols (javascript, info, etc.), and disallow anything not on the list. When new harmful protocols emerge, this will ensure that your protection does not break and make it less vulnerable to attacks that attempt to obfuscate invalid values to evade a blacklist.

Allowing "safe" HTML:

Permitting clients to post HTML markup ought to be maintained a strategic distance from at every possible opportunity, yet now and then it's a business prerequisite. For instance, a blog website may permit remarks to be posted containing some constrained HTML markup. The exemplary methodology is to attempt to sift through conceivably unsafe labels and JavaScript. You can attempt to execute this utilizing a whitelist of safe labels and characteristics, yet because of errors in program parsing motors and peculiarities like transformation XSS, this methodology is incredibly hard to actualize safely. The least awful alternative is to utilize a JavaScript library that performs sifting and encoding in the client's program, for example, DOMPurify. Different libraries permit clients to give content in markdown organization and convert the markdown into HTML. Sadly, every one of these libraries have XSS vulnerabilities occasionally, so this is certifiably not an ideal arrangement. On the off chance that you do utilize one you should screen intently for security refreshes.

How to prevent XSS using a template engine Server-side template engines such as Twig and Free Marker are used by many modern websites to embed complex content into HTML. These usually describe their own system of escaping. For instance, you can use the e) (filter in Twig, with an argument that defines the context:

user.firstname | e('html')

Some other template engines, such as Jinja and React, by contrast, avoid dynamic content, effectively preventing most XSS occurrences. When you decide whether to use a given template engine or system, we suggest carefully checking escape features.

How to stop XSS in PHP:

There is a built-in encoding feature in PHP for entities called html entities. When within an HTML background, you can call this feature to escape your input. With three arguments, the function should be called:

- Your input string.
- ENT_QUOTES, which is a flag that specifies all quotes should be encoded.
- The character set, which in most cases should be UTF-8

For example: <?php echo htmlentities($input, ENT_QUOTES, 'UTF-8');?>

When in a JavaScript string context, you need to Unicode-escape input as already described. Unfortunately, PHP doesn't provide an API to Unicode-escape a string. Here is some code to do that in PHP:

```
    <?php
function jsEscape($str) {
  $output = '';
  $str = str_split($str);
  for($i=0;$i<count($str);$i++) {
   $chrNum = ord($str[$i]);
   $chr = $str[$i];
   if($chrNum === 226) {
     if(isset($str[$i+1]) &&ord($str[$i+1]) === 128) {
       if(isset($str[$i+2]) &&ord($str[$i+2]) === 168) {
         $output .= '\u2028';
         $i += 2;
         continue;
       }
       if(isset($str[$i+2]) &&ord($str[$i+2]) === 169) {
         $output .= '\u2029';
         $i += 2;
         continue;
       }
     }
   }
   switch($chr) {
     case "'":
     case '"':
     case "\n";
     case "\r";
     case "&";
     case "\\";
     case "<":
     case ">":
       $output .= sprintf("\\u%04x", $chrNum);
     break;
     default:
       $output .= $str[$i];
     break;
    }
  }
  return $output;
}
?>
```

Here we see how to use the jsEscape function in PHP:
<script>x = '<?php echo jsEscape($_GET['x'])?>';</script>
Alternatively, we could use a template engine.
How to prevent XSS client-side in JavaScript.
To escape user input in an HTML context in JavaScript, you need your own HTML encoder. Here is some example JavaScript code that converts a string to HTML entities:
function
htmlEncode(str)
$returnString(str).replace(/[\wedge\backslash w.]/gi, function(c)$

return '&#'+c.charCodeAt(0)+';';

);

We would use then this function as follows:

<script>document.body.innerHTML = htmlEncode(untrustedValue)</script>

If your input is inside a JavaScript string, you need an encoder that performs Unicode escaping. Here is a sample Unicode-encoder: function jsEscape(str)

$returnString(str).replace(/[\wedge\backslash w.]/gi, function(c)$

return '\\u'+('0000'+c.charCodeAt(0).toString(16)).slice(-4);

);

You would then use this function as follows:

$< script > document.write('< script > x = "' + jsEscape(untrustedValue) + '";< \vee script >') < /script >$

How to prevent XSS in jQuery:

The foremost broadly recognized sort of XSS in jQuery is the point at which you pass client request to a jQuery selector. Web engineers would regularly utilize location.hash and pass it to the selector which would cause XSS as jQuery would render the HTML. jQuery seen this issue and settled their selector method of reasoning to check at whatever point input begins with a hash. Directly jQuery will conceivably render HTML if the principal character could be a <. On the off chance merely pass untrusted data to the jQuery selector, ensure you viably escape this value by using the jsEscape command.

Content security approach (CSP) is the final line of defence against cross-site scripting. In case that your XSS counteractions fail, you'll be able to utilize CSP to stop XSS by limiting what an attacker can do. CSP lets you control diverse things, for instance, notwithstanding of whether exterior substance can be stacked and whether inline substance will be executed. To communicate CSP you've got to consolidate a HTTP response header called Content-Security-Policy. An example CSP is as follows: default-src 'self'; script-src 'self'; object-src 'none'; frame-src 'none'; base-uri 'none';

This approach indicates that assets such as pictures and scripts can be stacked as a page. So indeed in case an attacker can effectively infuse an XSS payload they are counted as stack assets from the same root. This enormously decreases the chance that an attacker can misuse the XSS vulnerability. If you require stacking of external assets, guarantee you simply permit scripts that don't allow an attacker to take control of your webpage. For example, in case your whitelist certain spaces at that point an attacker can stack any script from those spaces. Where conceivable, attempt to have assets on your possess space. On the off chance that that's not possible at that point you'll utilize hash- or nonce-based policy to allow scripts on different domains. A nonce could be a arbitrary string that's included as an quality of a script or asset, which can as it were be executed in case the irregular string matches the server-generated one. An attacker is incapable to figure the randomized string and so cannot conjure a script or asset with a substantial nonce and so the asset will not be executed.

4. Tutorial on Javascript Attack

First of all we need some knowledge of PHP,HTML and of course Javascript. We need HTML to write our code and we need Javascript to make our webpage more dynamic and add more effects. In our example we used CSS to make our webpage more beautiful and friendly to human eye.

HTML CODE

First of all we must write the code, for the webpage we want. We constructed a simple one, with a button and a cell which we can write in it. The button will show us what we wrote in the cell. Quick and simple example for our cause, to show the attack. The code for the page is:

<!DOCTYPE html>

<html>

<head>

<title>Javascript Attack</title>

```
<link href="% static ’javascript/css/attack.css’ %" rel="stylesheet">
</head> <body>
<h1 class="search-form-header">Javascript Attack</h1>
<form class="search-form" autocomplete="off">
<input class="search-input" id="query-input" type="text" name="query" />
<button class="search-button" type="submit" role="button">Hit it</button>
</form>
<h3 class="search-query">You texted: <span id="query-output" class="query"></span></h3>
</li>
<script src="% static ’javascript/js/jqValidation.js’ %"></script>
</body>
</html>
```

We will analyze the code later after we construct the javascript file. We make a new file and call it “static”. In there we will put the javascript and the CSS file(optional). Inside the static file we make 1 new file named Javascript. Next, inside Javascript file we make 2 new files called js and css. In the js file we will make the javascript file and in the css file, the css.

Js file

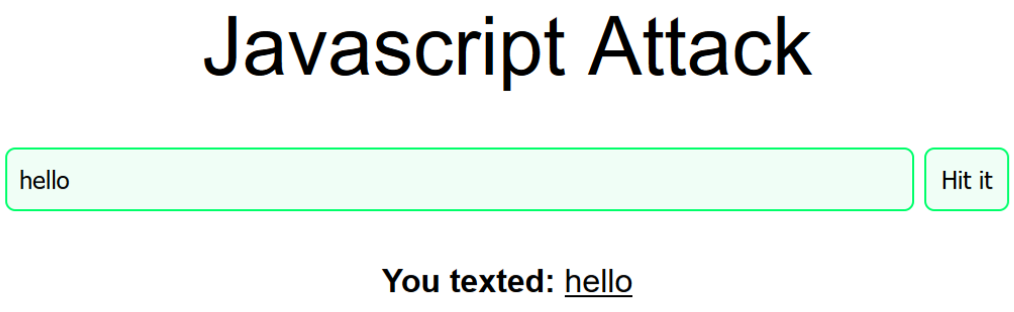
The code for the javascript file is:

```
1 document.cookie = "username=vasilis"
2 document.cookie = "password=123456"
3 if (document.readyState == ’loading’)
document.addEventListener(’DOMContentLoaded’, ready)
else
ready()
function ready()
var query = new URL(window.location).searchParams.get(’query’)
document.getElementById(’query-input’).value = query
document.getElementById(’query-output’).innerHTML = query
```

In the first line we see the document.cookie, which we need it to read and write cookies in our webpage. We insert our username and our password for example. The other command that we must give attention is the document.getElementById. This method returns an Element object representing the element whose id property matches the specified string. At the last row we use this method with innerHTML. This returns what we wrote in the box in our page. The innerHTML property sets or returns the HTML content (inner HTML) of an element.

4.1. How the attack works

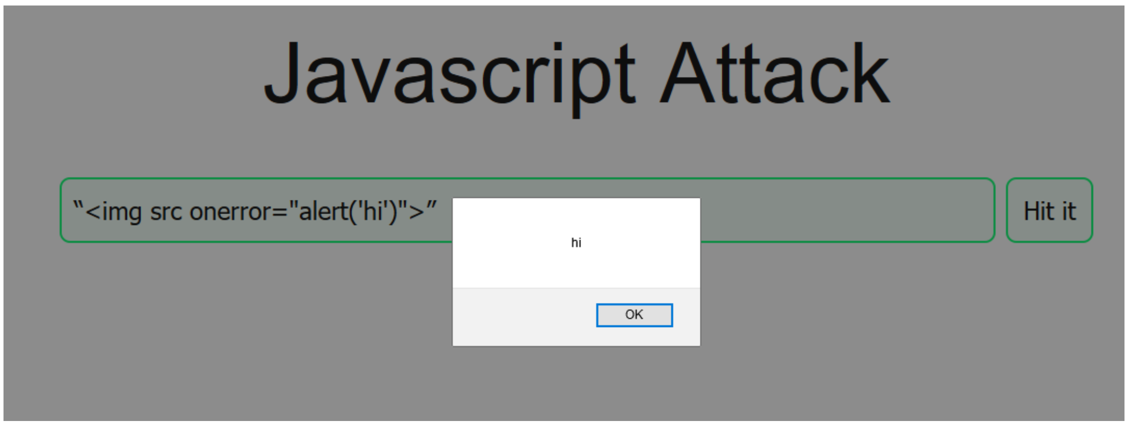
The attack works with some lines of code, that need to be entered in the input box of our website. First we present a simple example to see how our page works. For this example we put the word “hello”.



You texted: hello

Figure 3. Example of hello world

Next, we initiate the attack procedure. First we will try a line of code for understanding the existing vulnerability. The code we can insert is "<img src onerror="alert('hi')">" and the web-page will return the message as shown in Image 4:

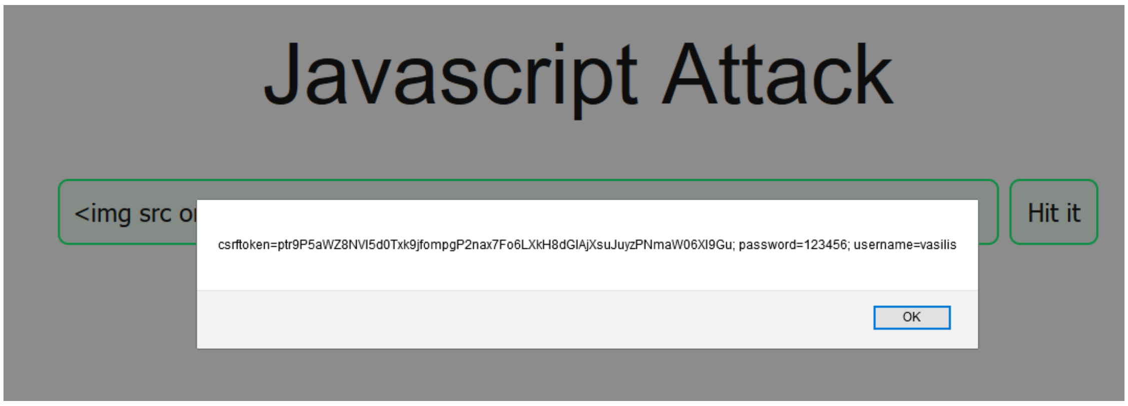


```
"<img src onerror="alert('hi')">"
```

```
hi
```

Figure 4. Our page when we input the command <img src onerror="alert('hi')">.

Now that the attacker knows the vulnerability that exists in the code, he can try and steal our cookies by filling the input box with the line "<img src onerror="alert(document.cookie)">". With this line the page executes the code and the result is to bring up in the alert box the cookies we have, in our case username and password.

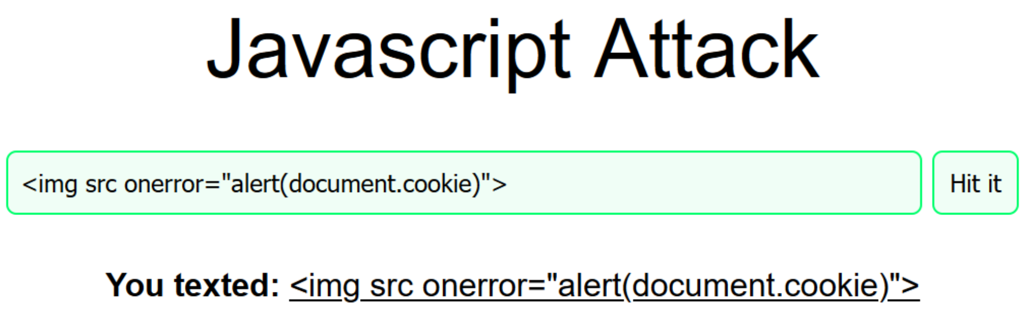


```
csrftoken=ptr9P5aWZ8NVI5d0Txk9jfompgP2nax7Fo6LXkH8dGIAjXsuJuyzPNmaW06XI9Gu; password=123456; username=vasilis
```

Figure 5. <img src onerror="alert(document.cookie)"> code and display

4.2. Mitigation of the attack

Why this happens? Because the page executes the code as it's on the inner code. But why it read it and executed it like it's in the code? The answer is in the javascript code, in line 12 "document.getElementById('query-output').innerHTML = query". The innerHTML is the one which gives us the perforation of our page. The innerHTML property sets or returns the HTML content (inner HTML) of an element. This means that what we write in our box the page will understand as HTML code and it exectues it. How to prevent it? We must change the code. We change innerHTML with innerText. With inner text whatever we write in the box will be interpreted as plain text, even if it includes code.


Javascript Attack

```
<img src onerror="alert(document.cookie)">
```

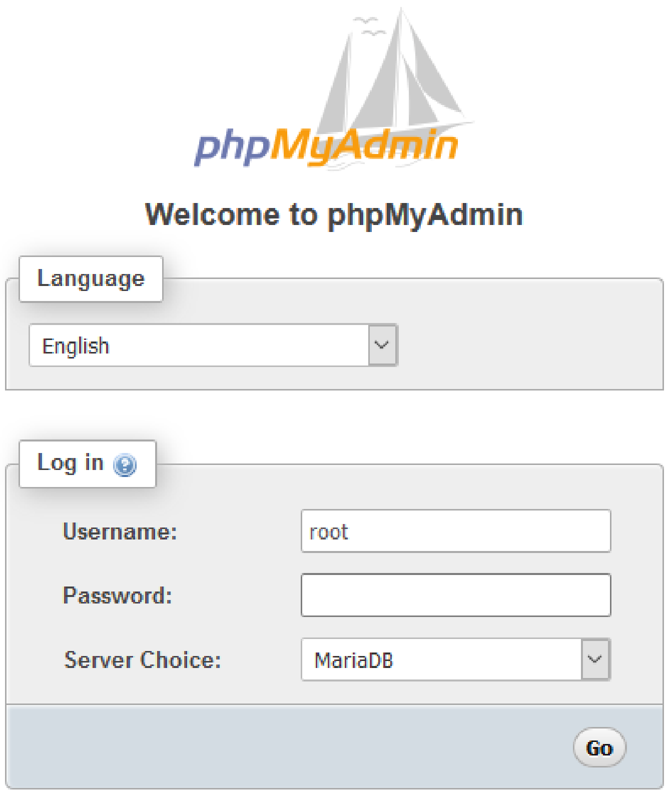
You texted: <img src onerror="alert(document.cookie)">

Figure 6. Webpage after mitigation

Now the attacker can't give any instruction-command to our system and cannot bypass the input page.

5. Tutorial on SQL Injection attack

We open a new file and we name the file sqlinjection.php. We need to name with the php at the end since we will put php code in and sql code also. Then we have to make our database before we continue. We use phpmyadmin, and we must ensure that when we want to go to our virtual environment wamp is already running (down right, at the task bar). We write http://localhost/phpmyadmin at our browser


Welcome to phpMyAdmin

Language: English

Log in

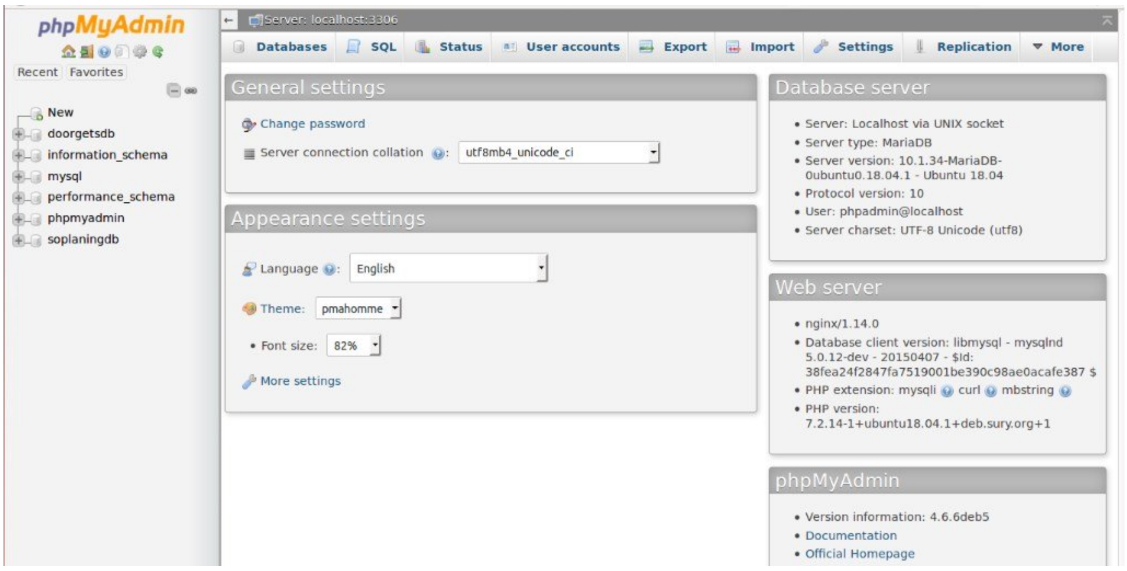
Username: root

Password:

Server Choice: MariaDB

Figure 7. Initial Screen

We fill at username the word "root" and at password we leave it blanc. At server choice we choose what server we want, we picked MariaDB. We hit go and we will enter our main page of phpMyadmin.



General settings

Change password

Server connection collation: utf8mb4_unicode_ci

Appearance settings

Language: English

Theme: pmahomme

Font size: 82%

More settings

Database server

- Server: Localhost via UNIX socket
- Server type: MariaDB
- Server version: 10.1.34-MariaDB-0ubuntu0.18.04.1 - Ubuntu 18.04
- Protocol version: 10
- User: phpadmin@localhost
- Server charset: UTF-8 Unicode (utf8)

Web server

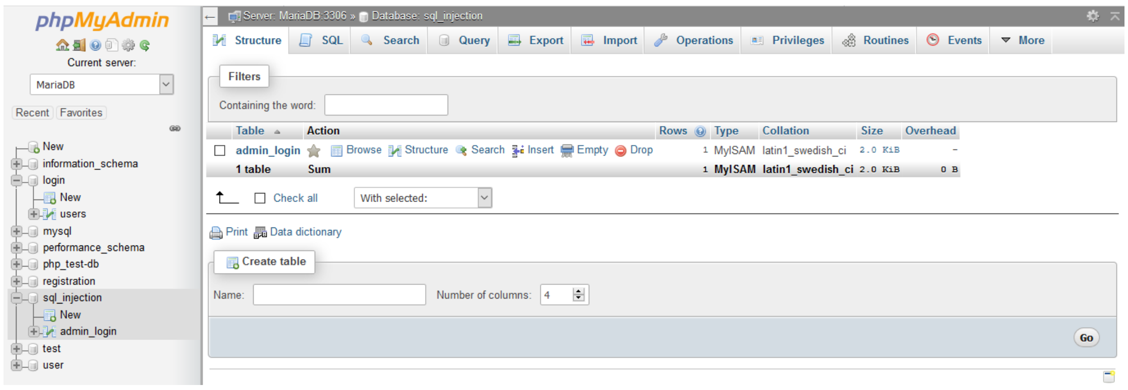
- nginx/1.14.0
- Database client version: libmysql - mysqlnd 5.0.12-dev - 20150407 - $Id: 38fea24f2847fa7519001be390c98ae0acafe387 $
- PHP extension: mysqli curl mbstring
- PHP version: 7.2.14-1+ubuntu18.04.1+deb.sury.org+1

phpMyAdmin

- Version information: 4.6.6deb5
- Documentation
- Official Homepage

Figure 8. Main page of phpMyadmin

Now we can start building our database. We need to build a database with one name and one password for the login page we want to make.We choose new and write a name for the database. We give the name "sql injection" and in the box next to it we can choose freely since it doesn't have any impact. Finally, we hit the button create.


phpMyAdmin

Current server: MariaDB

Recent Favorites

New
information_schema
login
New
users
mysql
performance_schema
php_test-db
registration
sql_injection
New
admin_login
test
user

Server: MariaDB:3306 » Database: sql_injection

Structure SQL Search Query Export Import Operations Privileges Routines Events More

Filters

Containing the word:

Table	Action	Rows	Type	Collation	Size	Overhead
admin_login	Browse Structure Search Insert Empty Drop	1	MyISAM	latin1_swedish_ci	2.0 KiB	-
1 table	Sum	1	MyISAM	latin1_swedish_ci	2.0 KiB	0 B

Check all With selected:

Print Data dictionary

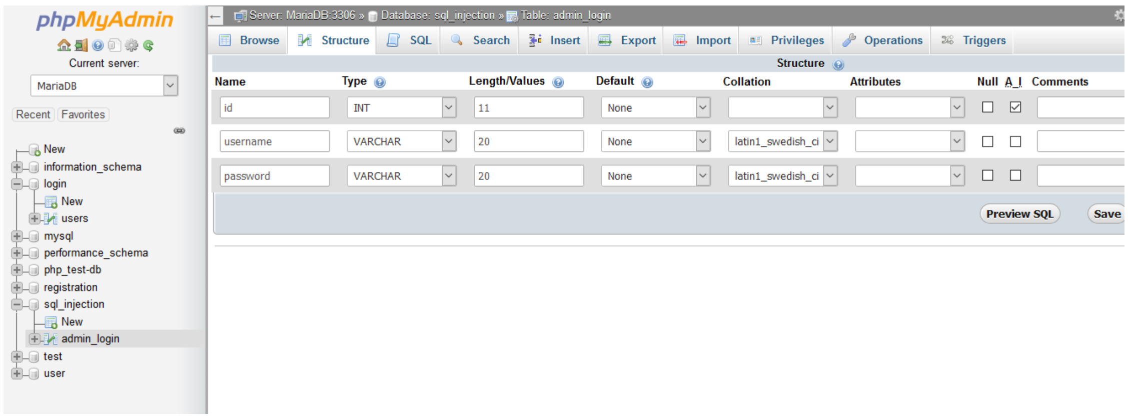
Create table

Name: Number of columns: 4

Go

Figure 9. The database is created

Then we need to create our table which will have our name and password. At create table we name the table, at ours example we named it *ıadmin¡ogin*▌ and at number of columns we select 3. We push the go button.


phpMyAdmin

Current server: MariaDB

Recent Favorites

New
information_schema
login
New
users
mysql
performance_schema
php_test-db
registration
sql_injection
New
admin_login
test
user

Server: MariaDB:3306 » Database: sql_injection » Table: admin_login

Browse Structure SQL Search Insert Export Import Privileges Operations Triggers

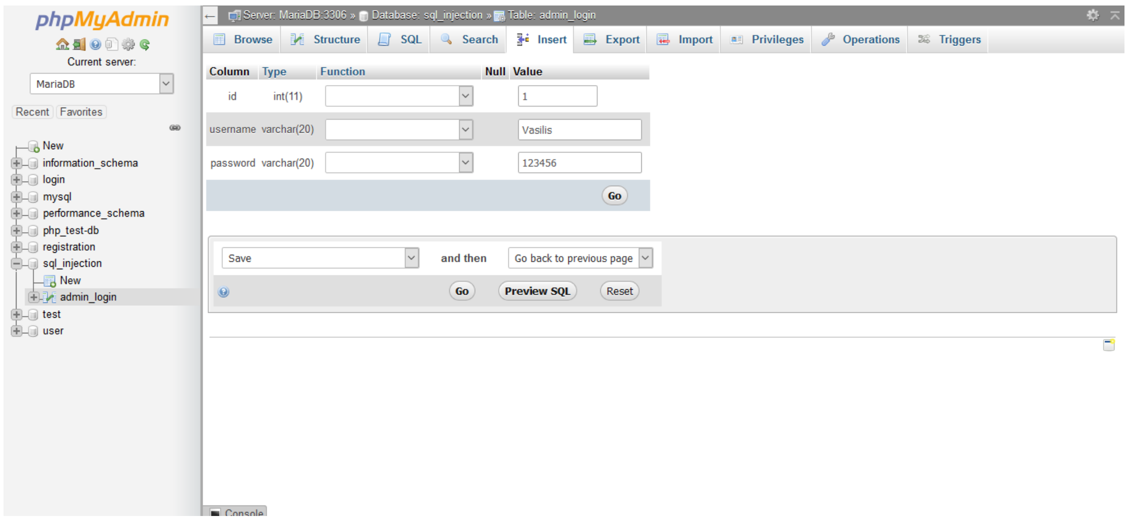
Structure

Name	Type	Length/Values	Default	Collation	Attributes	Null	A_I	Comments
id	INT	11	None			☐	☑	
username	VARCHAR	20	None	latin1_swedish_ci		☐	☐	
password	VARCHAR	20	None	latin1_swedish_ci		☐	☐	

Preview SQL Save

Figure 10. A snapshot of the table

Now we must write our parameters for each column. At first column we put at name the "id", cause we need an id parameter which will increase each time one password is saved. We wrote for id=1, for name= Vasilis and for password=123456.


phpMyAdmin

Current server: MariaDB

Recent Favorites

New
information_schema
login
mysql
performance_schema
php_test-db
registration
sql_injection
New
admin_login
test
user

Server: MariaDB:3306 » Database: sql_injection » Table: admin_login

Browse Structure SQL Search Insert Export Import Privileges Operations Triggers

Column	Type	Function	Null	Value
id	int(11)			1
username	varchar(20)			Vasilis
password	varchar(20)			123456

Go

Save and then Go back to previous page

Go Preview SQL Reset

Console

Figure 11. Inputs to the database tables

The code that we use for executing the SQL injection is presented here. Also the code for all the scenarios presented in this tutorial are on GitHub (https://github.com/vapapaspirou/javascript-and-sqlinjection) and the reader is advised to use them when going through the article.

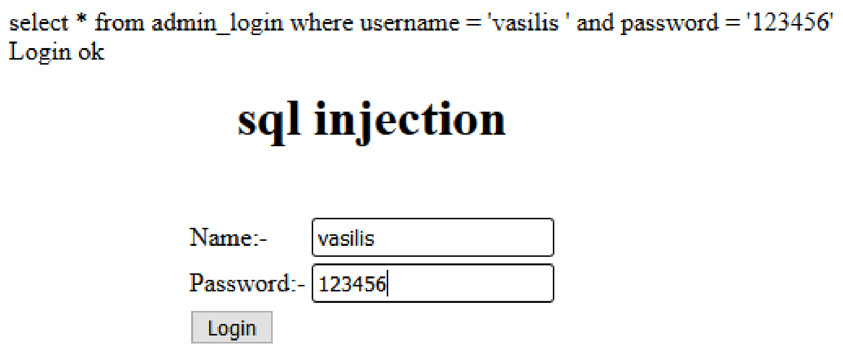
```
<?php

  if(isset(\$\_POST['submit']))

 {

 $\$conn = mysqli_connect("localhost","root","","sql_injection");$

 $\$name =  \$_POST['user'];$

$\$pwd  =  \$_POST['password'];$

$echo \$sql = "select * from admin_login where username = '\$name' and password = '\$pwd' ";$

echo "<br/>";

$\$res = mysqli\_query( \$conn,\$sql);$

$if (mysqli\_num_rows(\$res) >0 )$ {

echo "Login ok";

}
else
{
echo "Login failed";
}
   }
 ?>

 <!DOCTYPE html>
 <html>

 <head>

<meta charset="utf-8">

<title></title>

<link rel="stylesheet" href="">

   </head>

<style type="text/css">

.box input[type="text"]{
```

```
border: none;
border-radius: 3px;
outline:none;
padding: 3px;
}
</style>
 <body>
<div class="box">
<center><form method="post">
<table>
<h1>SQL INJECTION</h1>
<tr>
<td>Name:-</td>
<td><input type="text" name="user" value="" style="border: 0.5px solid \#111111">
</td>
</tr>
<tr>
<td>Password:-</td><br>
<td><input type="text" name="password" value="" style="border: 0.5px solid \#111"></td>
</tr>
<tr>
<td><input type="submit" name="submit" value="Login"></td>
</tr>
</table>
</form></center>
```

```
</div>

 </body>

 </html>
```

5.1. How the attack works

Firstly we must ensure we run wampserver; at the right bottom we will see the wamp icon turned green. The next step is going to our browser and type ergasia.com. That's because we had created the virtual environment with that name. If we hadn't created a new environment, then we should type in the url box, localhost. With that we go to our environment and select the sql injection file and after sql injection php file. The webpage must now be in our browser. Now we write the name and password we have. If the code was correclty inserted, the page must write login ok.



```
select * from admin_login where username = 'vasilis ' and password = '123456'
Login ok
```

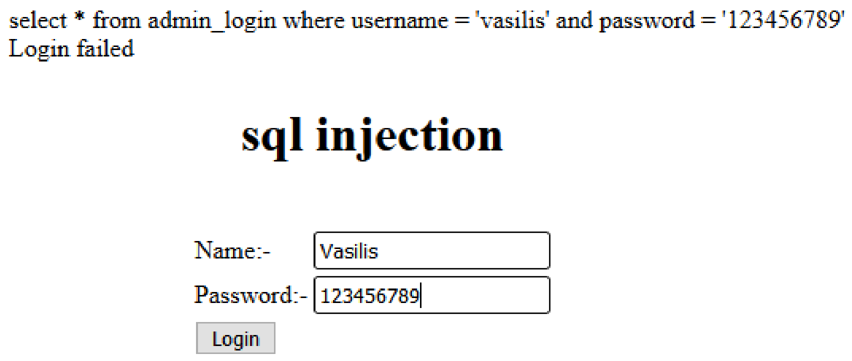
sql injection

Name:- vasilis

Password:- 123456

Figure 12. Succesfull login

Now let's try and put wrong name and password. Login failed will be shown.



```
select * from admin_login where username = 'vasilis' and password = '123456789'
Login failed
```

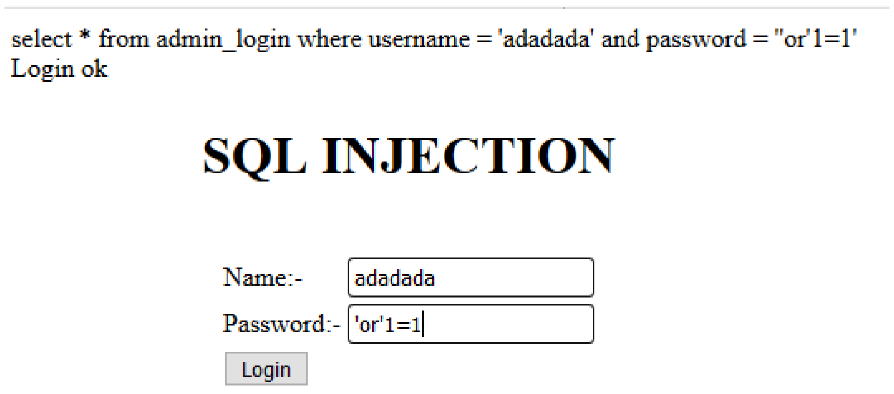
sql injection

Name:- Vasilis

Password:- 123456789

Figure 13. Login failed

Now we will write the line which bypasses our code. At the name we can write anything we want (although some restrictions like the size is considerable) and in password we write the line 'or'1=1.



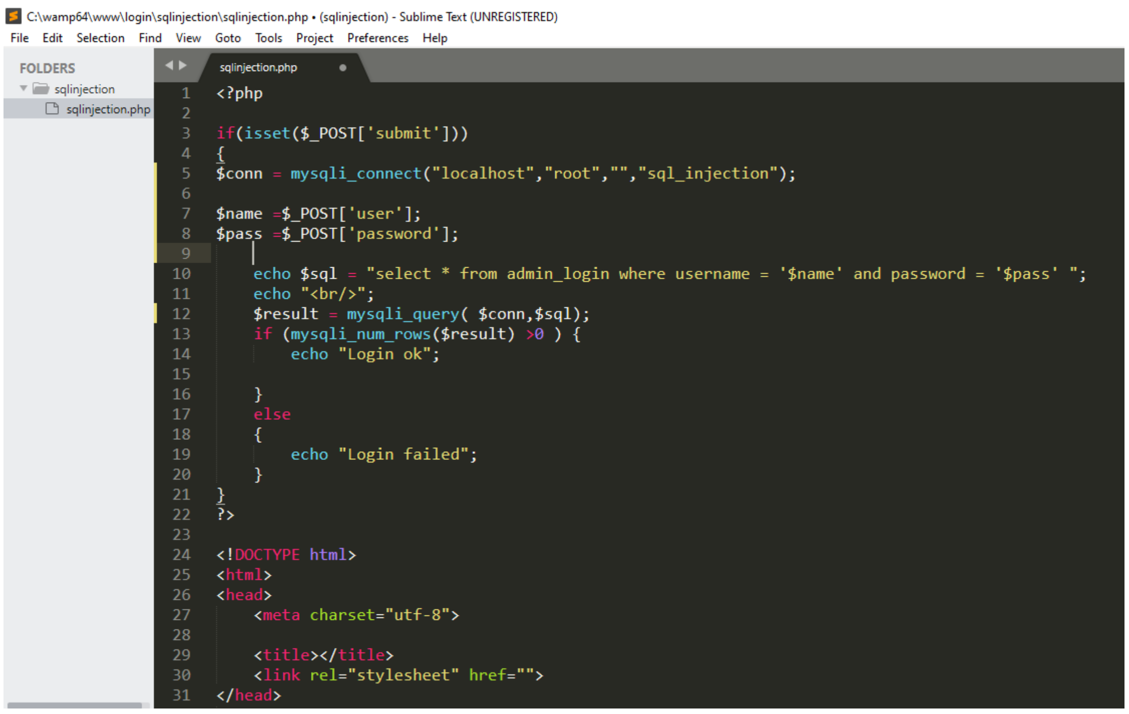
select * from admin_login where username = 'adadada' and password = ''or'1=1'
Login ok

SQL INJECTION

Name:-	adadada
Password:-	'or'1=1

Figure 14. SQL injection successful attack

The reason of this weakness is our code. It's in the meaning what conditions must be true to activate the rest of the code. The password=" or '1' =' 1' condition is always true, so the password verification never happens. It can also be said that the above statement is more or less equal to, and provided that the condition is true then the system will tell, it must continue. Line 7 and 8 are the ones that must change to prevent those attacks. $name = $_POST['user']; and $pass = $_POST['password']; . The $_POST variable is an array of variable names and values sent by the HTTP POST method. The $_POST variable is used to collect values from a form with method="post". Information sent from a form with the POST method is invisible to others and has no limits on the amount of information to send. So our code only tests if the password variable is meeting the true condition. With the 'or'1=1 the condition is always true and so it bypass any check.



C:\wamp64\www\login\sqlinjection\sqlinjection.php • (sqlinjection) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
sqlinjection
sqlinjection.php

```
1  <?php
2
3  if(isset($_POST['submit']))
4  {
5  $conn = mysqli_connect("localhost","root","","sql_injection");
6
7  $name =$_POST['user'];
8  $pass =$_POST['password'];
9
10     echo $sql = "select * from admin_login where username = '$name' and password = '$pass' ";
11     echo "<br/>";
12     $result = mysqli_query( $conn,$sql);
13     if (mysqli_num_rows($result) >0 ) {
14         echo "Login ok";
15
16     }
17     else
18     {
19         echo "Login failed";
20     }
21 }
22 ?>
23
24 <!DOCTYPE html>
25 <html>
26 <head>
27     <meta charset="utf-8">
28
29     <title></title>
30     <link rel="stylesheet" href="">
31 </head>
```

Figure 15. Code of our page where sql injection succeeded

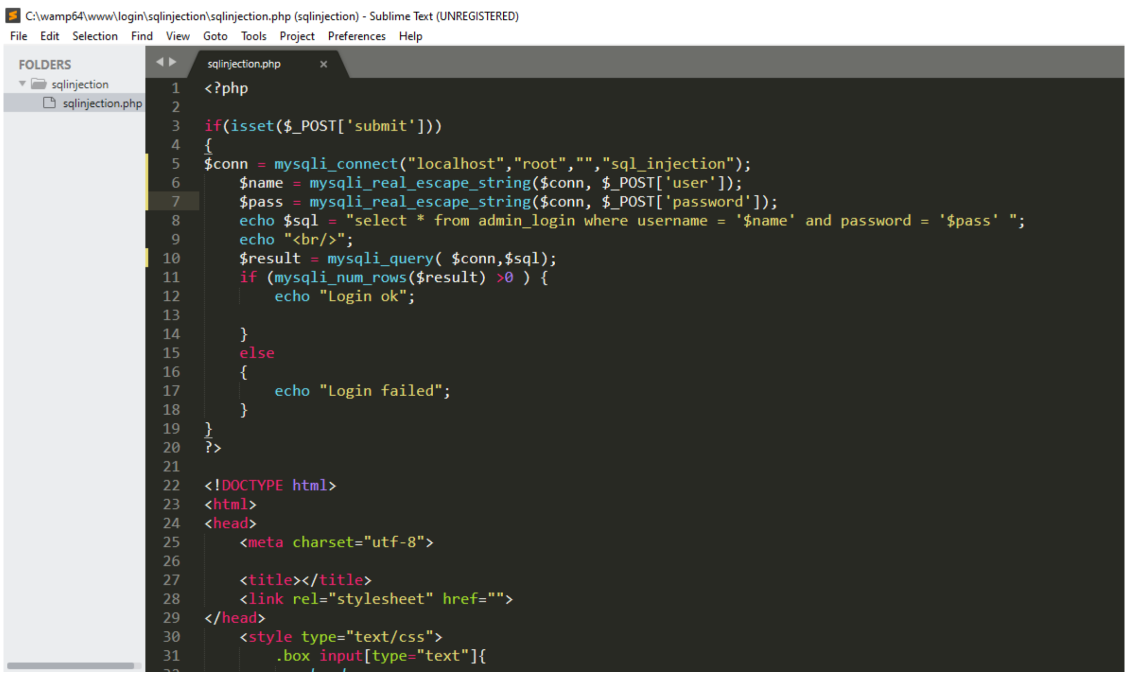
5.2. Mitigation of the attack

In order to stop the attack from taking place, We must change how the program test the values inserted. We can change lines 7 and 8 to a secure code. The new code will be :

$\$name = mysqli\_real\_escape\_string(\$conn, \$\_POST['user']);$

$\$pass = mysqli\_real\_escape\_string(\$conn, \$\_POST['password']);$

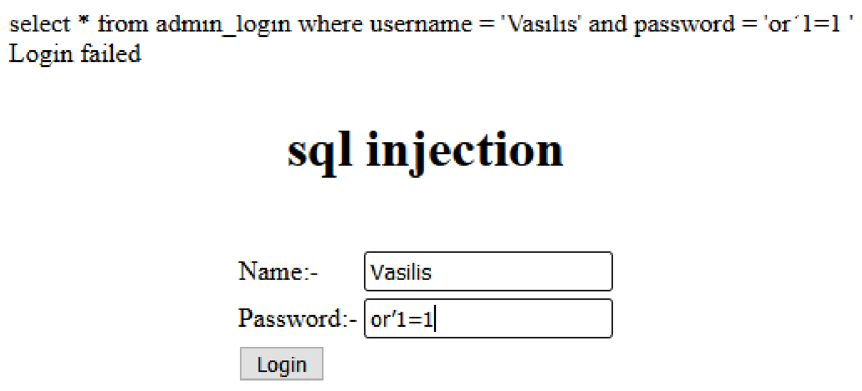
The real_escape_string() / mysqli_real_escape_string() function escapes special characters to a string for use in an SQL query, taking into account the current character set of the connection. This function is used to create a legal SQL string that can be used in an SQL statement. It prepends backslashes to the following characters: \x00, \n, \r, \, ′, " and \x1a. This function must always (with few exceptions) be used to make data safe before sending a query to MySQL


C:\wamp64\www\login\sqlinjection\sqlinjection.php (sqlinjection) - Sublime Text (UNREGISTERED)
File Edit Selection Find View Goto Tools Project Preferences Help
FOLDERS
sqlinjection
sqlinjection.php
sqlinjection.php

```
<?php

if(isset($_POST['submit']))
{
$conn = mysqli_connect("localhost","root","","sql_injection");
    $name = mysqli_real_escape_string($conn, $_POST['user']);
    $pass = mysqli_real_escape_string($conn, $_POST['password']);
    echo $sql = "select * from admin_login where username = '$name' and password = '$pass' ";
    echo "<br/>";
    $result = mysqli_query( $conn,$sql);
    if (mysqli_num_rows($result) >0 ) {
        echo "Login ok";

    }
    else
    {
        echo "Login failed";
    }
}
?>

<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">

    <title></title>
    <link rel="stylesheet" href="">
</head>
    <style type="text/css">
        .box input[type="text"]{
```

Figure 16. Our code after the mitigation

With that change the attacker cannot use the ‘or’1=1 line in order to penetrate our system.



```
select * from admin_login where username = 'Vasilis' and password = 'or´1=1 '
Login failed
```

Name:- Vasilis
Password:- or'1=1

Figure 17. SQL injection is unsuccesfull

6. Discussion - Conclusions

Cross-Site Scripting (XSS) assaults are a sort of ijection, wherein malignant contents are infused into in any case amiable and confided in sites. XSS attacks happen when an aggressor utilizes a web application to send vindictive code, for the most part as a program side content, to an alternate end client. Imperfections that permit these assaults to succeed are very broad and happen anyplace a web application utilizes contribution from a client inside the yield it produces without approving or encoding it.

An attacker can utilize XSS to send a vindictive content to a client. The end client's program has no real way to realize that the content ought not be trusted, and will execute the content. Since it thinks the content originated from a confided in source, the noxious content can get to any treats, meeting tokens, or other delicate data held by the program and utilized with that site. These contents can even modify the substance of the HTML page. For additional subtleties on the various kinds of XSS imperfections.

An attacker who exploits a cross-site scripting vulnerability is usually able to:

- Pretend to be the victim user
- Perform any action that the user is able to perform
- Read all the data that the user can access
- Capture user credentials
- Perform the virtual deployment of the website
- Inject the Trojan's functionality into your website

The real affect of an XSS assault usually depends on the nature of the application, its functionality and information, and the status of the compromised client. For case:

- In a web based application, where all clients are self-contained and all data is open, the affect will regularly be negligible
- In an application holding touchy information, such as managing account exchanges, emails, or healthcare records, the affect will be important
- In case the compromised client has lifted benefits inside the application (for example is an admin), at that point the effect will be large, permitting the attacker to require full control of the application and compromise all clients' information

XSS Attacks are the most popular attacks since they exploit a variety of vulnerabilities [11]. The vulnerabilities can be found using google dev tools and other specialised solutions such as Burp Suite's web vulnerability scanner. The most simple way to prevent any vulnerability is to "escape" the characters a user can input and to sanitise any HTML that appears in the inputs of the web app. The manual testing step of a web application can also search for mirrored and stored XSS witch normally involves sending simple single entries (such as a short alphanumeric string) to each entry point in the application; identifying each location where the sent entry is returned in HTTP responses; and testing each location individually to determine if properly made entries can be used to execute arbitrary JavaScript [12].

Physically testing for DOM-based XSS emerging from URL parameters includes a similar process: setting a few basic interesting inputs, utilizing the browser's designer tools to look the DOM for this input, and testing each area to decide whether it is exploitable [10]. However, other sorts of DOM XSS are harder to identify. To discover DOM-based vulnerabilities in non-URL-based input (such as document.cookie) or non-HTML-based sinks (like setTimeout), there is no substitute for investigating JavaScript code, which can be amazingly time-consuming. Here dev-tools can help by showing all scripts loaded on a page and a tester can ensure that all required scripts are loaded only at the end of the document (this usually prevents the attacker from using some jquery features for example Ajax malevolent requests of the attacker).

In common, viably anticipating XSS vulnerabilities is likely to include several measures:

- At the point where client input is received, use basic input rules
- When client content arrives in HTTP reactions it ought to be encoded to avoid it from being translated as dynamic content. Depending on the settings, this might require applying combinations of HTML, URL, JavaScript, and CSS encoding
- To avoid XSS in HTTP reactions that aren't expecting to contain any HTML or JavaScript, headers can be utilized such as Content-Type and X-Content-Type-Options headers to guarantee that browsers translate the reactions correctly

- As a final line of defense, Content Security Policy (CSP) can be utilized to decrease the severity of any XSS vulnerabilities that still happen

.

XSS vulnerabilities are difficult to anticipate basically since there are so numerous vectors where an XSS assault can be utilized in most applications. As opposed to other vulnerabilities, such as SQL injection or OS command injection, XSS influences the client of the site, making it more difficult to capture and indeed harder to settle. Moreover not at all like SQL injection, which can be dispensed with the right utilize of arranged articulations, there's no single standard or technique to prevent cross-site scripting assaults. Whereas utilizing the security layers just like the ones said over, it's a great way to avoid most XSS attacks, it is fundamental to note that whereas those avoidance strategies would cover most of the XSS attack vectors, they cannot cover everything. It is pivotal to utilize a combination of programmed static testing, code audit, and dynamic testing along with applying secure coding techniques. If these measures are not deployed and run frequently along with other security measures like Intrusion Detection Systems [13] the attackers can take control of essential services and sensitive data.

As cyber attacks on critical systems continue to rise, awareness and training of the appropriate personnel is very important and for that reason technical solutions should be combined with those [14]. The danger is still very large from cross site scripting attacks that can lead to data breaches. As long as users tend to browse the internet even more than before they are exposed to several threats. When designing a web application We must follow as much as we can security and privacy preservation rules and try not to leave any loopholes to programs. Experience is a great factor for programmers to know these attacks and articles that present simple attack defense scripts are of great need especially for junior programmers and students. This article presents simple scenario where XSS scripting attacks take place and proposes several defense actions. The presented tutorial examples are accompanied by the respective code that allows readers to experiment and learn.

Funding: This research received no external funding.

Conflicts of Interest: All authors declare no conflict of interest.

1. Flanagan, D.; Like, W.S. JavaScript: The Definitive Guide, 5th, 2006.
2. MacDonald, M.; Szpuszta, M.; Lair, R.; Lefebvre, J. *Pro Asp. net 2.0 in C# 2005*; Springer, 2005.
3. Rahman, R.U.; Wadhwa, D.; Bali, A.; Tomar, D.S. The Emerging Threats of Web Scrapping to Web Applications Security and Their Defense Mechanism. In *Encyclopedia of Criminal Activities and the Deep Web*; IGI Global, 2020; pp. 788–809.
4. Security, W. Threat Reports, 2020. Available at https://www.whitehatsec.com/resources-category/threat-reports/.
5. MITRE. Common Weakness Enumeration, 2020. Available at http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html.
6. Rodríguez, G.E.; Torres, J.G.; Flores, P.; Benavides, D.E. Cross-site scripting (XSS) attacks and mitigation: A survey. *Computer Networks* **2020**, *166*, 106960.
7. Al-Khurafi, O.B.; Al-Ahmad, M.A. Survey of web application vulnerability attacks. 2015 4th International Conference on Advanced Computer Science Applications and Technologies (ACSAT). IEEE, 2015, pp. 154–158.
8. Tripathi, P.; Thingla, R. Cross Site Scripting (XSS) and SQL-Injection Attack Detection in Web Application. Proceedings of International Conference on Sustainable Computing in Science, Technology and Management (SUSCOM), Amity University Rajasthan, Jaipur-India, 2019.
9. Chaudhary, P.; Gupta, B.; Gupta, S. Cross-site scripting (XSS) worms in Online Social Network (OSN): Taxonomy and defensive mechanisms. 2016 3rd International Conference on Computing for Sustainable Global Development (INDIACom). IEEE, 2016, pp. 2131–2136.

10. Gupta, S.; Gupta, B.B.; Chaudhary, P. Hunting for DOM-Based XSS vulnerabilities in mobile cloud-based online social network. *Future Generation Computer Systems* **2018**, *79*, 319–336.
11. Nagar, N.; Suman, U. Analyzing virtualization vulnerabilities and design a secure cloud environment to prevent from XSS attack. *International Journal of Cloud Applications and Computing (IJCAC)* **2016**, *6*, 1–14.
12. Sarmah, U.; Bhattacharyya, D.; Kalita, J.K. A survey of detection methods for XSS attacks. *Journal of Network and Computer Applications* **2018**, *118*, 113–143.
13. Stewart, B.; Rosa, L.; Maglaras, L.A.; Cruz, T.J.; Ferrag, M.A.; Simoes, P.; Janicke, H. A novel intrusion detection mechanism for scada systems which automatically adapts to network topology changes. *EAI Endorsed Transactions on Industrial Networks and Intelligent Systems* **2017**, *4*.
14. Cook, A.; Smith, R.; Maglaras, L.; Janicke, H. Using gamification to raise awareness of cyber threats to critical national infrastructure. BCS, 2016.