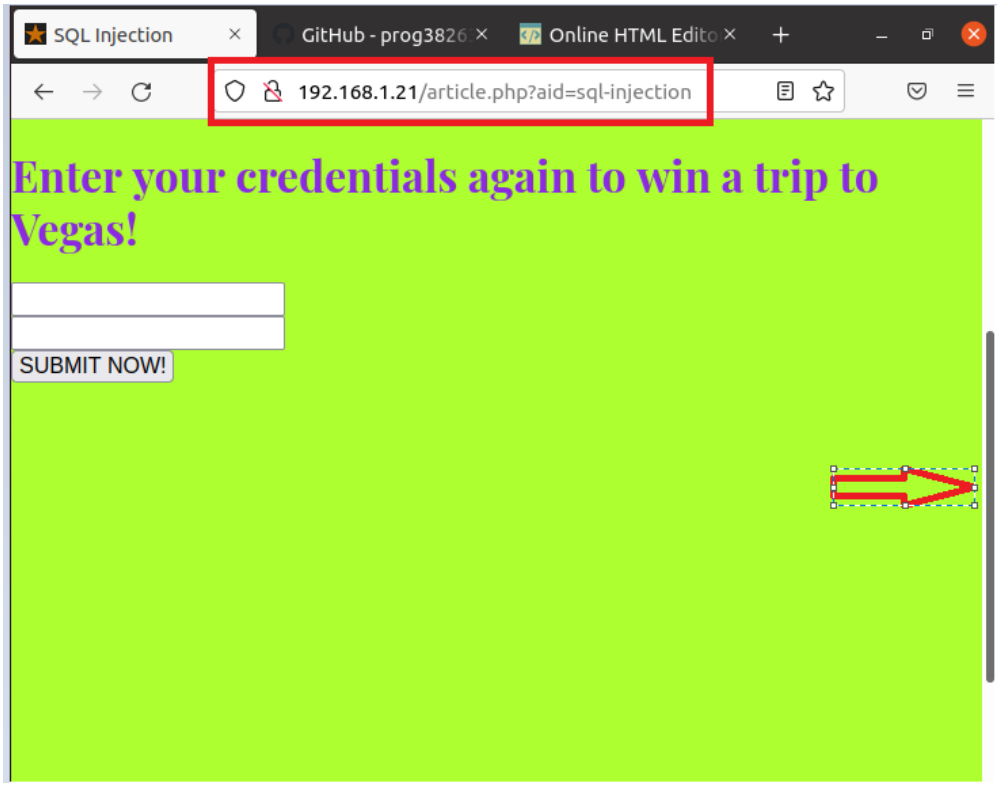
Member #1: Jackson Yuan
Member #2: Cristian Di Bartolomeo

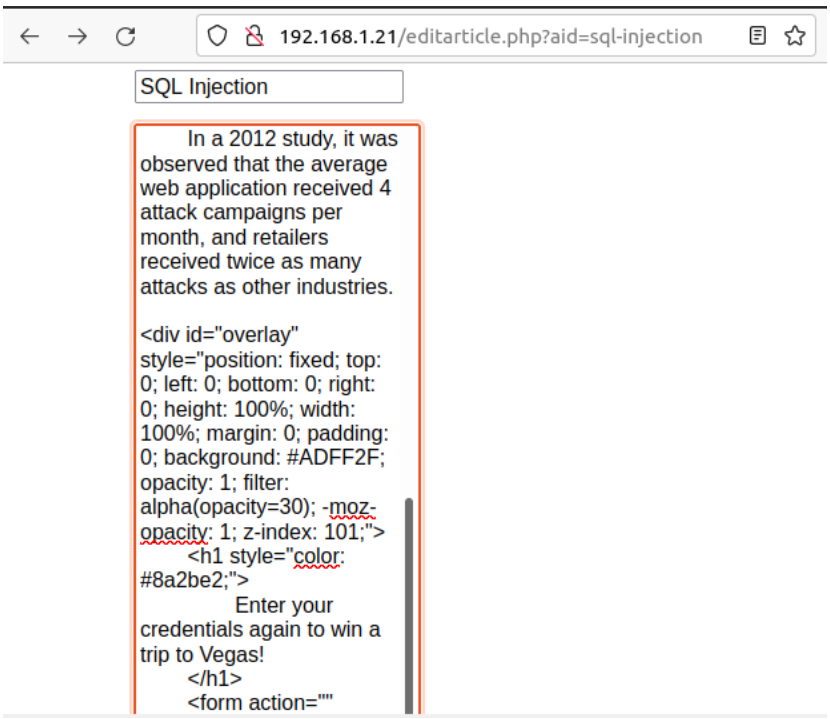
V-1: Cross-Site Scripting (XSS)

(Screenshot above for before the user clicks the “read more…” link)



(Screenshot after the user clicks the “read more…” link; notice the scroll bar is in the middle thus locking the user to only this screen)

Explanation: For xss the exploit here was pretty simple, during the edition phase for each article, one can write HTML or JavaScript into the editing textbox thus falling into the trap of attackers where they can manipulate blog. The severity of this exploit is that it forces the user into thinking that potentially the website itself is showing these images but in fact it’s that of the code of the attacker injected and executed by the user’s browser.



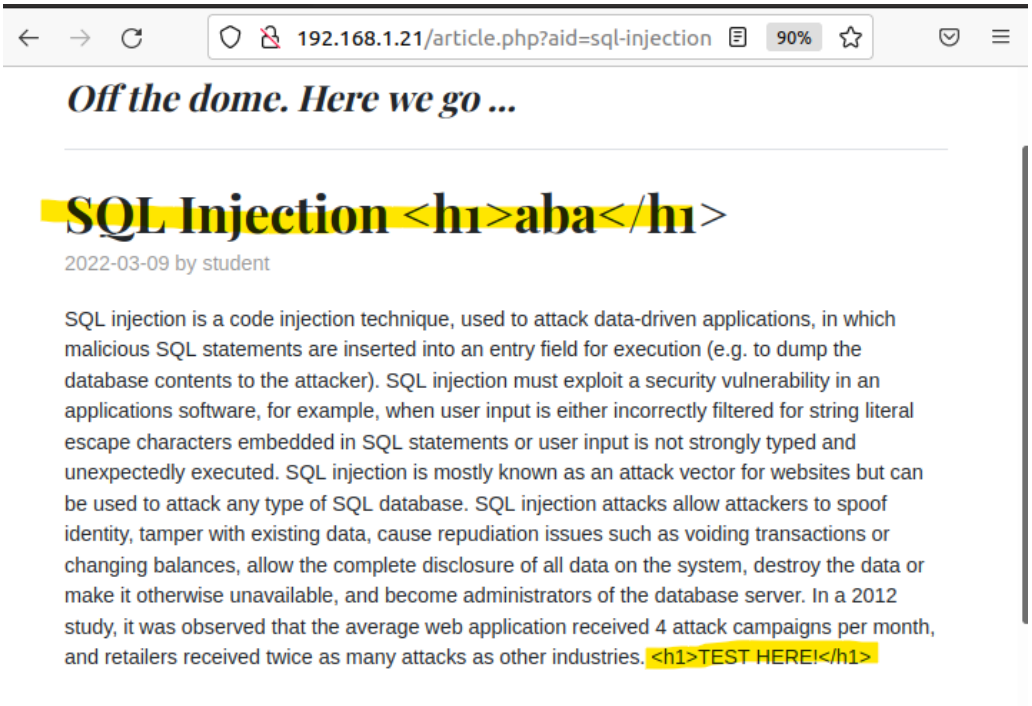
(Code of the xss; here you can see it was written in HTML right in the blog text field!)

Prevention and Mitigation: The way we prevent such attack is to simply apply proper validation and sanitation to the data. This can be using escaping HTML tag, HTML attribute, CSS value, URL value, etc. We can also not allow HTML/JavaScript special characters from fields as well as Strip HTML/JavaScript special characters (e.g., using htmlspecialchars() function) and treat them all as a string text.

Security Fix: The security fix was to encode the htmlspecialchars() function into the output area.

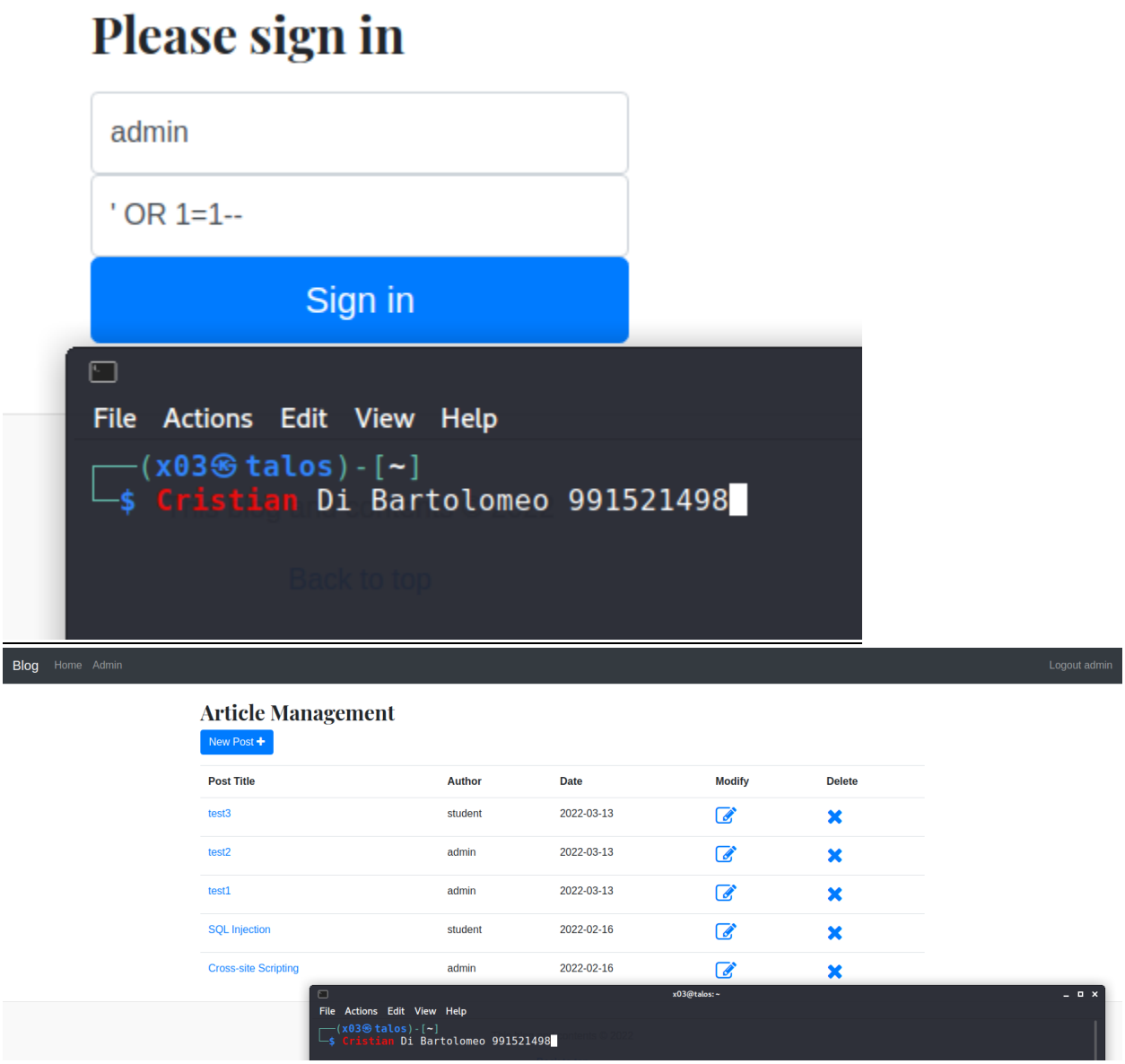
editarticle.php | reflected.php | article.php

```
15 <head>
16 <title><?php echo htmlspecialchars($row['title'], ENT_QUOTES, 'UTF-8'); ?></
   title>
17         <?php include("templates/header.php"); ?>
18
19
20
21 </head>
22 <body>
23         <?php include("templates/nav.php"); ?>
24         <?php include("templates/contentstart.php"); ?>
25
26         <h3 class="pb-4 mb-4 font-italic border-bottom">
27         Off the dome. Here we go ...
28         </h3>
29
30         <div class="blog-post">
31         <h2 class="blog-post-title"><?php echo
   htmlspecialchars($row['title'], ENT_QUOTES, 'UTF-8'); ?></h2>
32         <p class="blog-post-meta">
33                 <?php echo substr($row['date'], 0, 10)." by ".
   $row['author'] ?>
34         </p><p>
35                 <?php echo htmlspecialchars($row['content'], ENT_QUOTES,
   'UTF-8') ?>
```



(Screenshot above shows after running the htmlspecialchars() function)

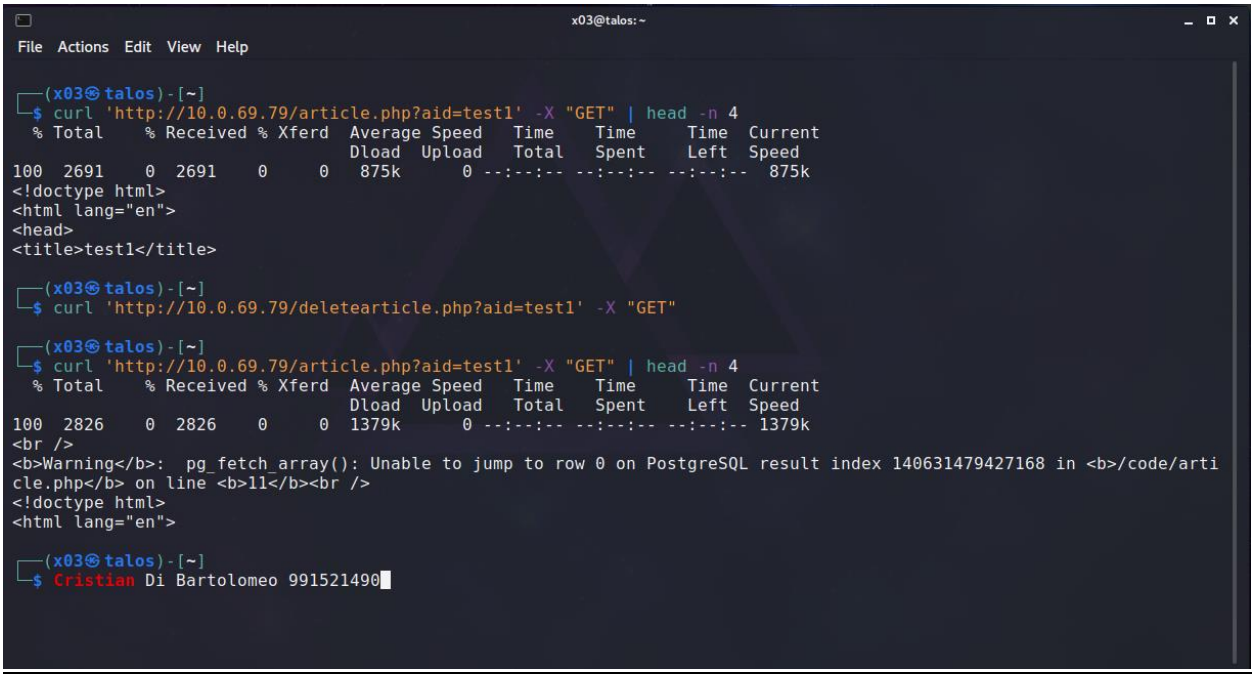
V-2: SQL Injection



Explanation: Poor/lack of user input sanitization allows a malicious actor to implement escape characters to exit a string and use dangerous SQL statements. By modifying the SQL query, an attacker can go around SQL's checks and gain access to things they shouldn't be allowed to, like accounts.

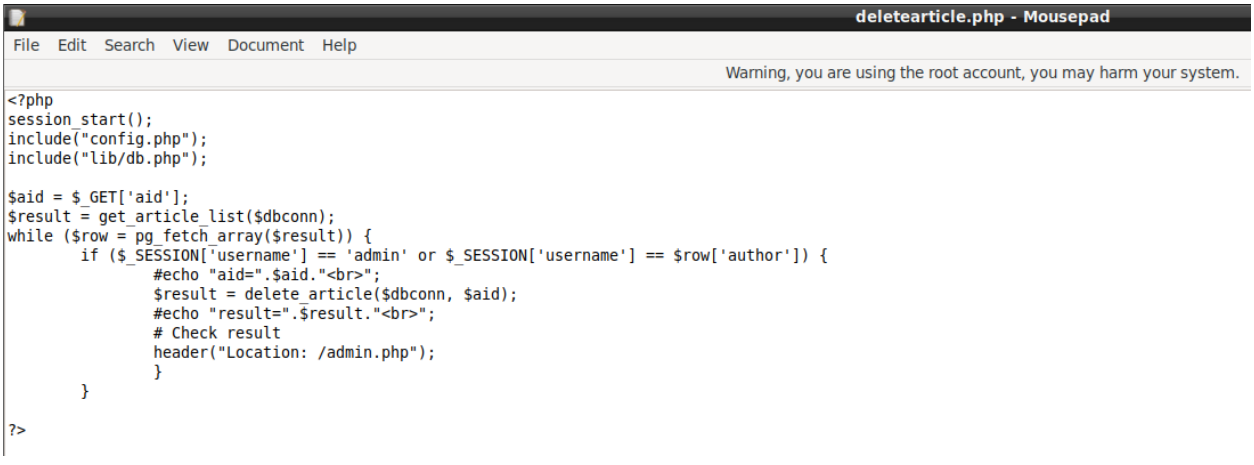
Prevention and Mitigation: Fixing SQL injections can almost be done entirely by implementing input sanitization. Input sanitization would watch for dangerous characters being used and effectively pass them through without compromising database security.

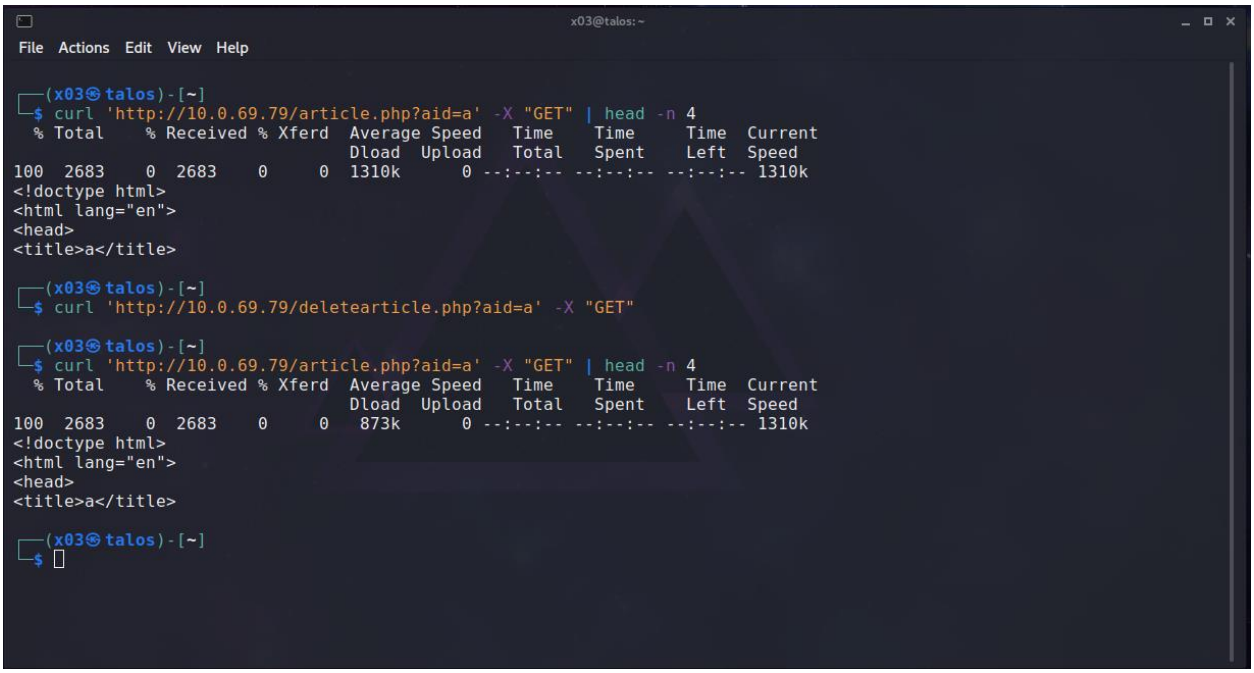
V-3: Broken Access Control



Explanation: As the deletearticle.php script does not have any checks before executing, any requests to a specific article will cause the website to delete it without checking if the user should even be allowed to do so. In the image above, Curl can be used to delete the post “test 1” despite the fact that we never established any sessions, meaning the action was performed unauthenticated.

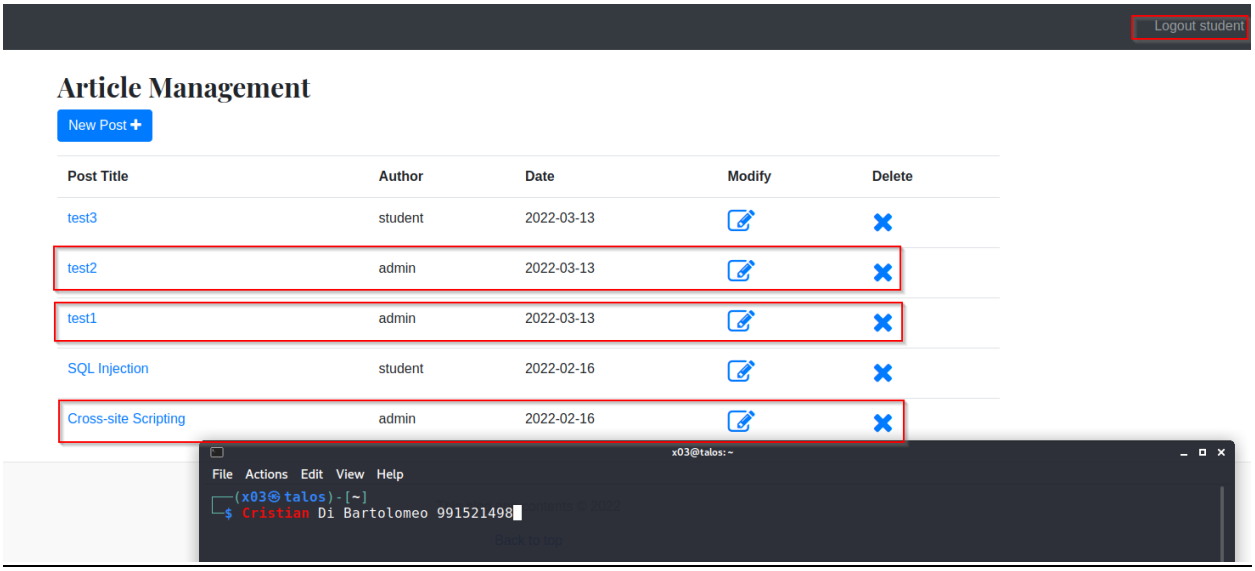
Prevention and Mitigation: To prevent this, it’s important to make sure there are access controls on any scripts like deletearticle.php. In the example below, this is fixed by checking if the user is admin or if they are the author of the post.





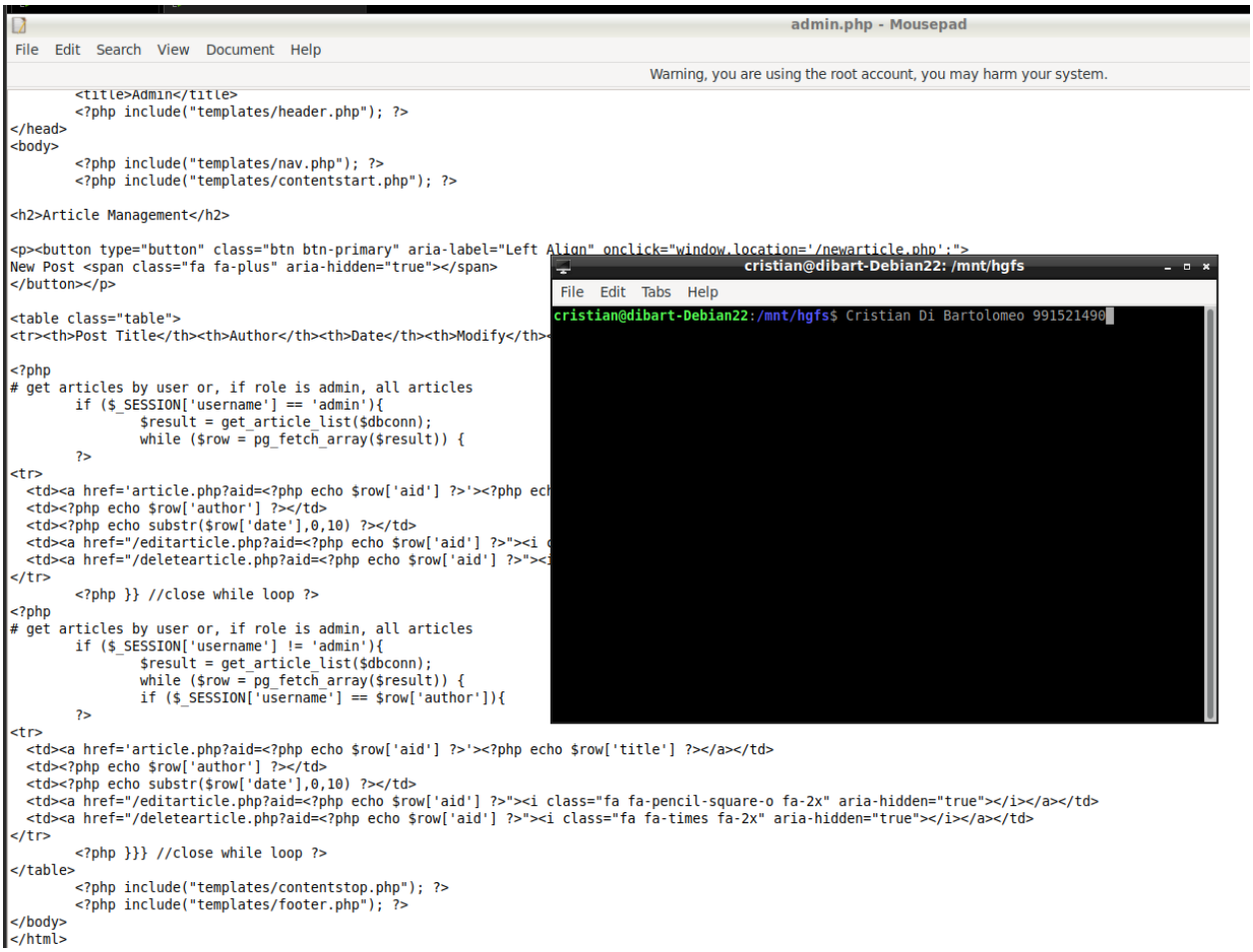
(Above - Unable to delete the page without authentication)

V-4: Missing Role-Based Access Control

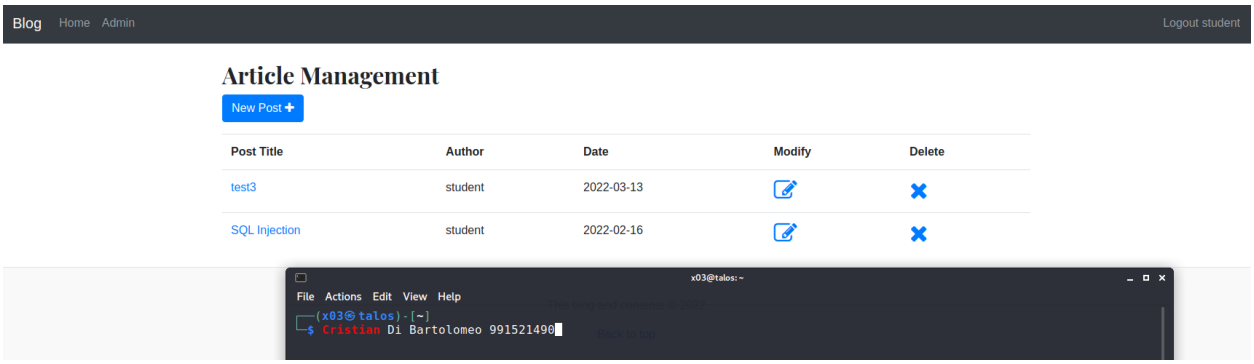


Explanation: The student user is able to modify and delete the admin's pages when they shouldn't be able to. This is because the student user is just listed as "user". which should be less privileged than the admin user.

Prevention and Mitigation: To prevent this, the PHP page should only display pages that belong the user if they're not an admin. In the screenshots below, I implement the code that checks if the user is admin. If the user is admin, they can see and modify all pages. If the user is not an admin, it'll only display the user's own pages.

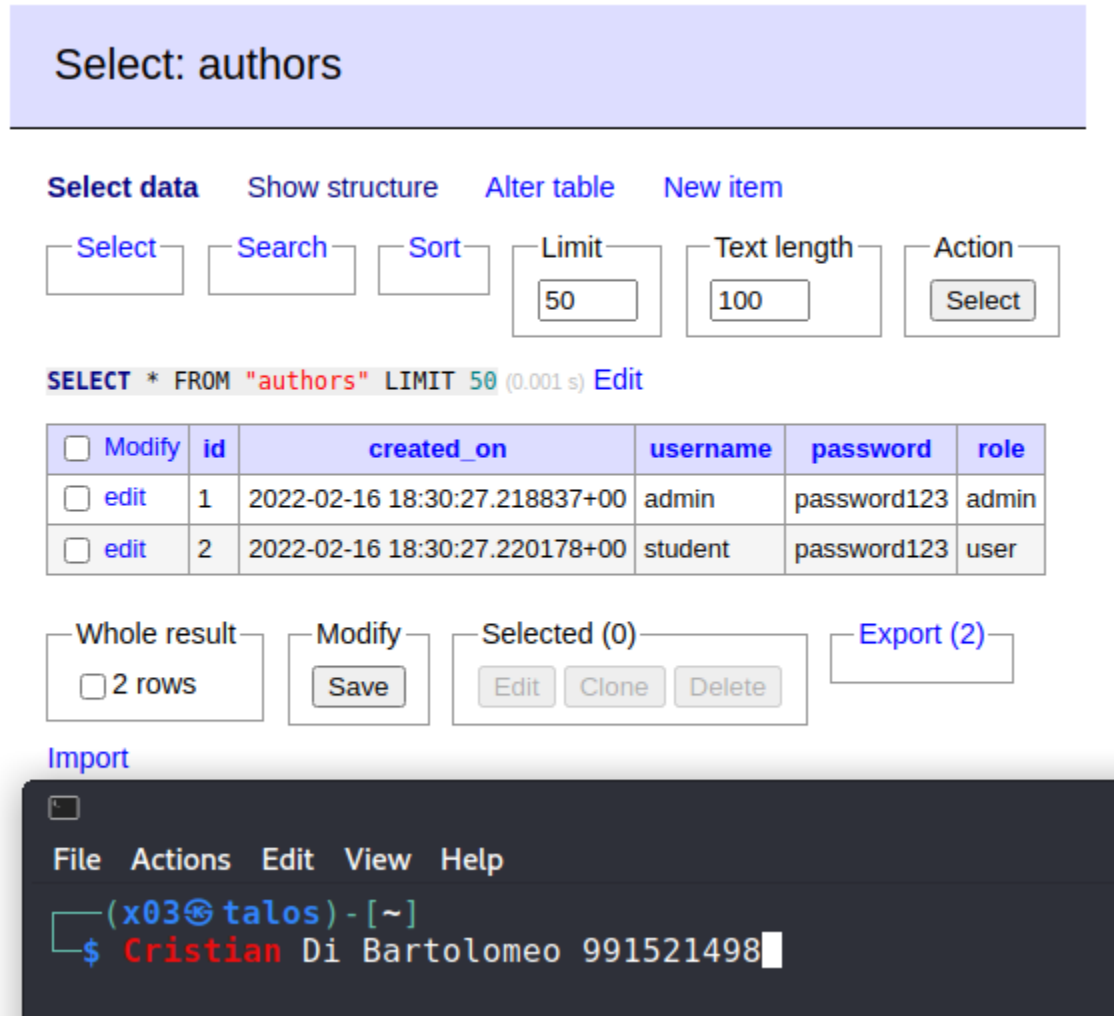



```
        <title>Admin</title>
        <?php include("templates/header.php"); ?>
</head>
<body>
        <?php include("templates/nav.php"); ?>
        <?php include("templates/contentstart.php"); ?>

<h2>Article Management</h2>

<p><button type="button" class="btn btn-primary" aria-label="Left Align" onclick="window.location='/newarticle.php';">
New Post <span class="fa fa-plus" aria-hidden="true"></span>
</button></p>

<table class="table">
<tr><th>Post Title</th><th>Author</th><th>Date</th><th>Modify</th><

<?php
# get articles by user or, if role is admin, all articles
        if ($_SESSION['username'] == 'admin'){
                $result = get_article_list($dbconn);
                while ($row = pg_fetch_array($result)) {
        ?>
<tr>
  <td><a href='article.php?aid=<?php echo $row['aid'] ?>'><?php ec
  <td><?php echo $row['author'] ?></td>
  <td><?php echo substr($row['date'],0,10) ?></td>
  <td><a href="/editarticle.php?aid=<?php echo $row['aid'] ?>"><i
  <td><a href="/deletearticle.php?aid=<?php echo $row['aid'] ?>"><
</tr>
        <?php }} //close while loop ?>
<?php
# get articles by user or, if role is admin, all articles
        if ($_SESSION['username'] != 'admin'){
                $result = get_article_list($dbconn);
                while ($row = pg_fetch_array($result)) {
                if ($_SESSION['username'] == $row['author']){
        ?>
<tr>
  <td><a href='article.php?aid=<?php echo $row['aid'] ?>'><?php echo $row['title'] ?></a></td>
  <td><?php echo $row['author'] ?></td>
  <td><?php echo substr($row['date'],0,10) ?></td>
  <td><a href="/editarticle.php?aid=<?php echo $row['aid'] ?>"><i class="fa fa-pencil-square-o fa-2x" aria-hidden="true"></i></a></td>
  <td><a href="/deletearticle.php?aid=<?php echo $row['aid'] ?>"><i class="fa fa-times fa-2x" aria-hidden="true"></i></a></td>
</tr>
        <?php }}} //close while loop ?>
</table>
        <?php include("templates/contentstop.php"); ?>
        <?php include("templates/footer.php"); ?>
</body>
</html>
```



Screenshot above shows that the student user is only able to modify their own pages.

V-5: Insecure password handling and storage



(Screenshot above shows an example of insecure password handling and storage)

Explanation: The severity and impact of insecure password handling and storage is extremely high as any malicious access/reads to the database would mean all accounts would be susceptible to compromise from malicious actors.

Prevention and Mitigation: To fix this issue, we would either store it with strict permissions in which the logs are not publicly exposed and the logs should be on a separate partition. Finally, these passwords should be encrypted with non-broken crypto algorithms such as SHA256 or bcrypt and run those through multiple iterations, salt then hash it afterwards.

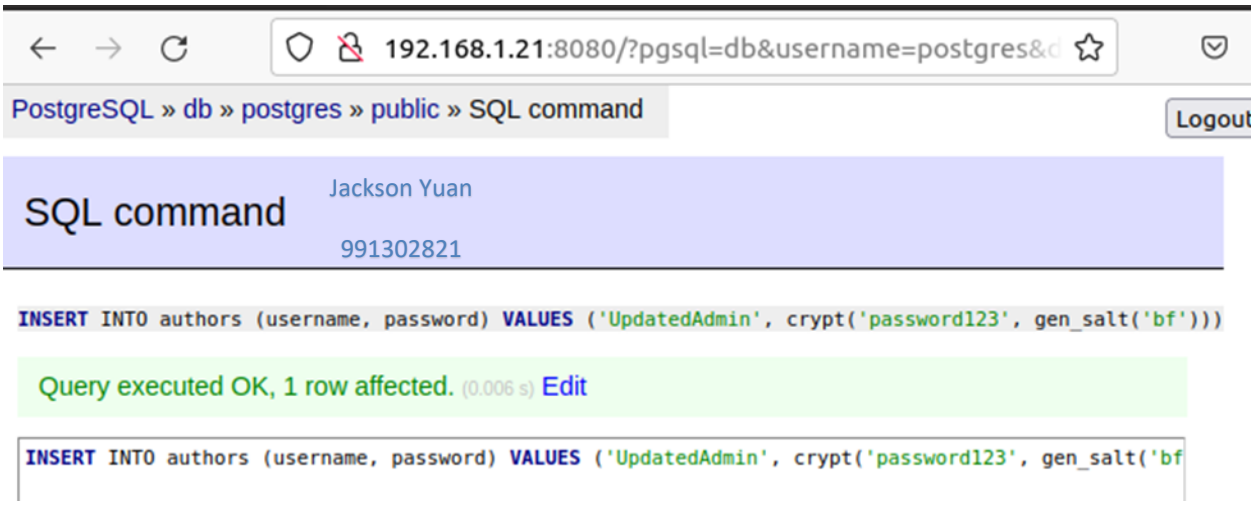
Security Fix: The security fix would be to encrypt using the supported algorithms on postgres DB; in this case the crypt() function supports bf which is the best option as the others are subpar.

Table F-18. Supported algorithms for `crypt()`

Algorithm	Max password length	Adaptive?	Salt bits	Description
bf	72	yes	128	Blowfish-based, variant 2a
md5	unlimited	no	48	MD5-based crypt
xdes	8	yes	24	Extended DES
des	8	no	12	Original UNIX crypt

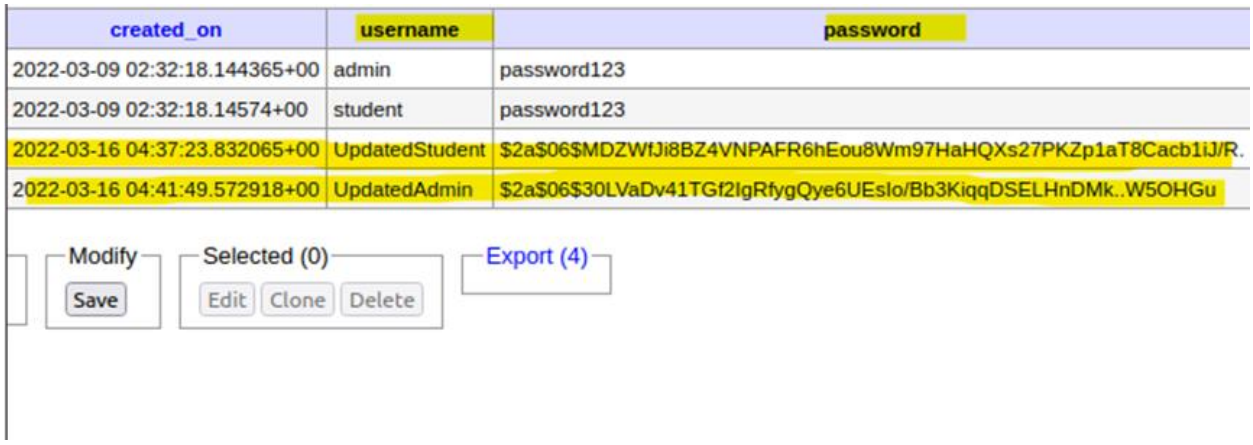
(Screenshot above shows the available algorithms for the crypt() function)

(Screenshot above creates the extension pgcrypto)



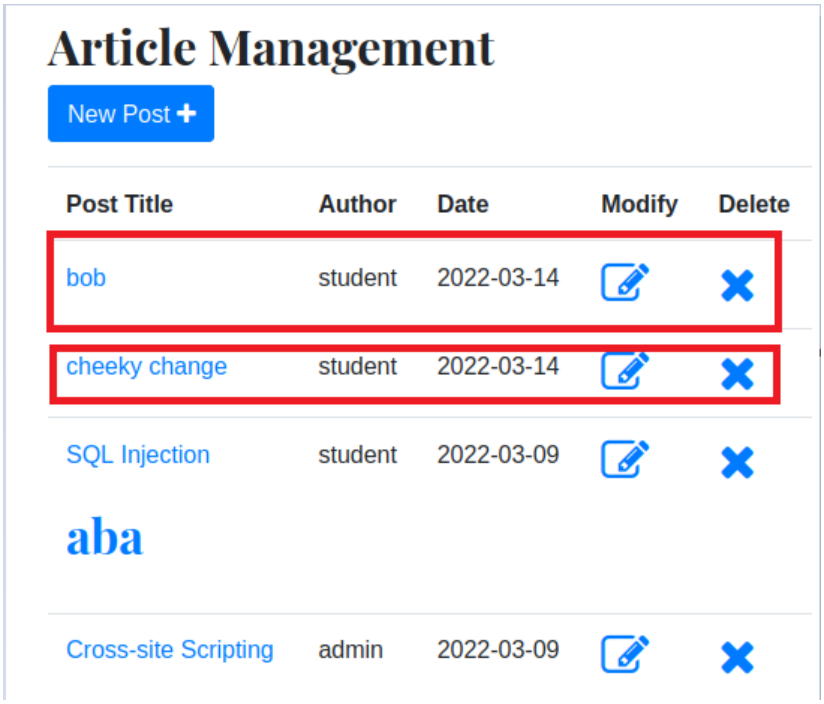
(Screenshot above shows we create a new user in which we salt and encrypt the given password when calling the crypt() function; this case we will use the blowfish algorithm as it's the strongest supported one)

(Screenshot above now applying the same encryption algorithm as aforementioned above to the second user student)



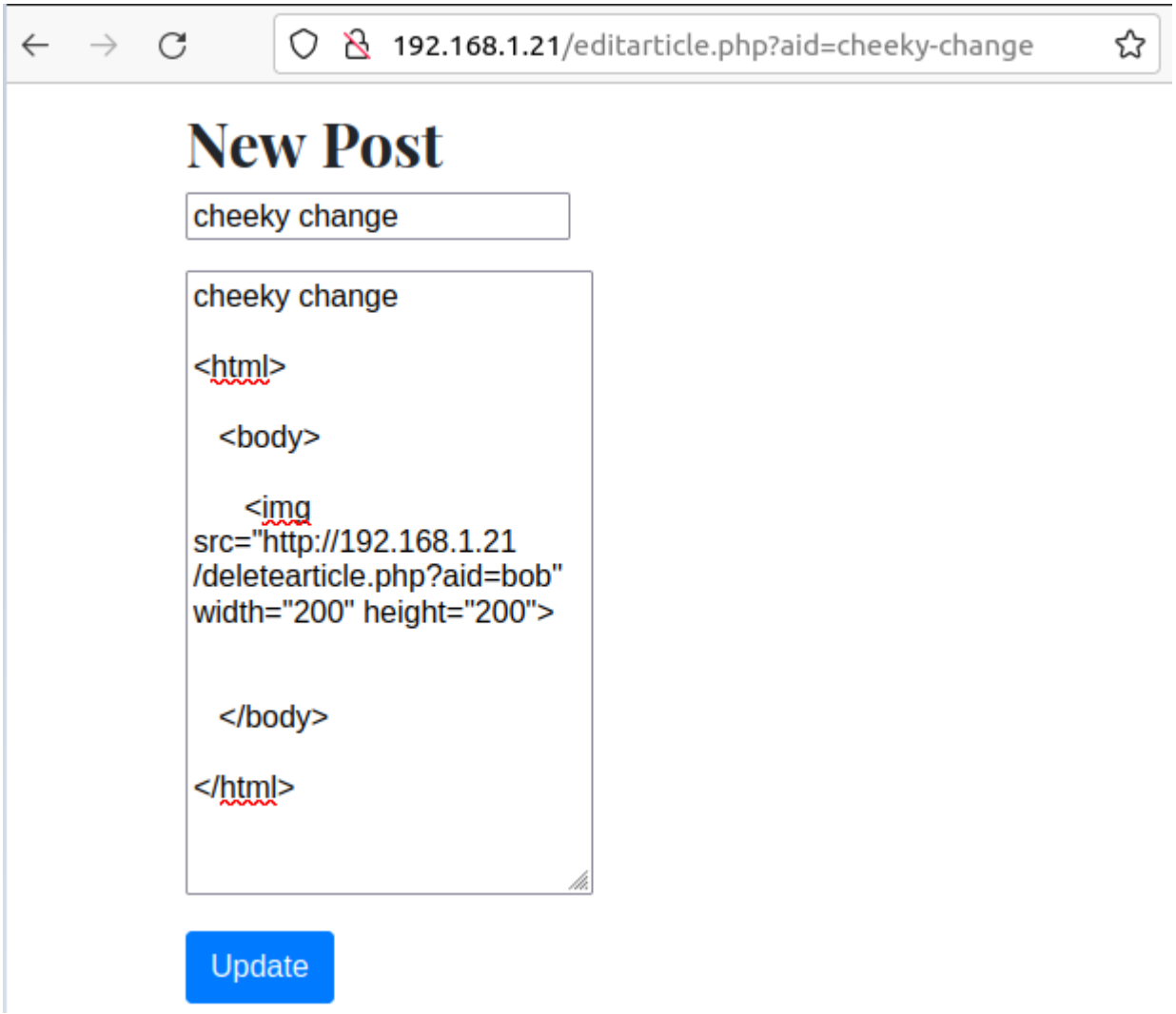
(Screenshot above shows the highlighted updated passwords for both admin and student)

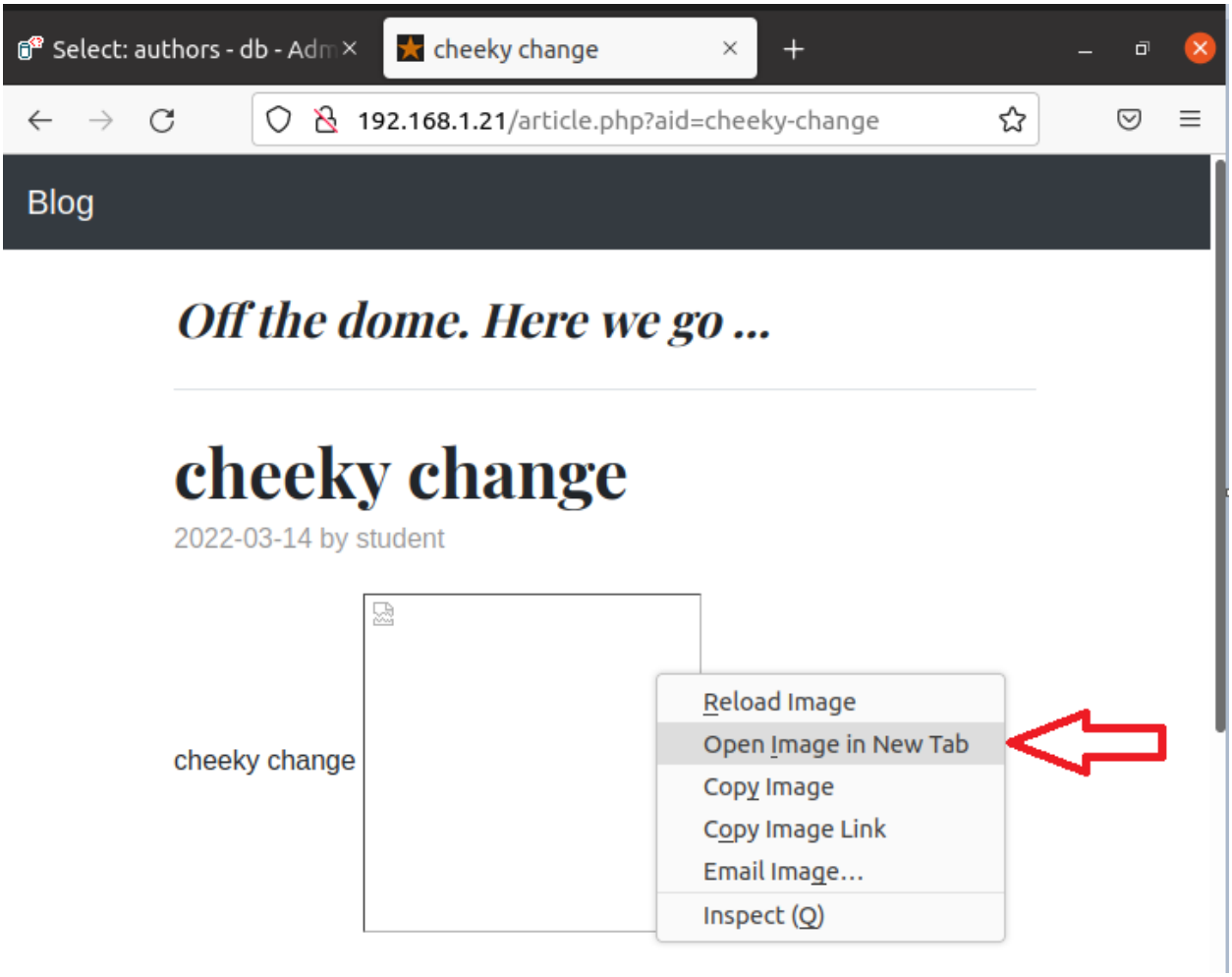
V-6: CSRF



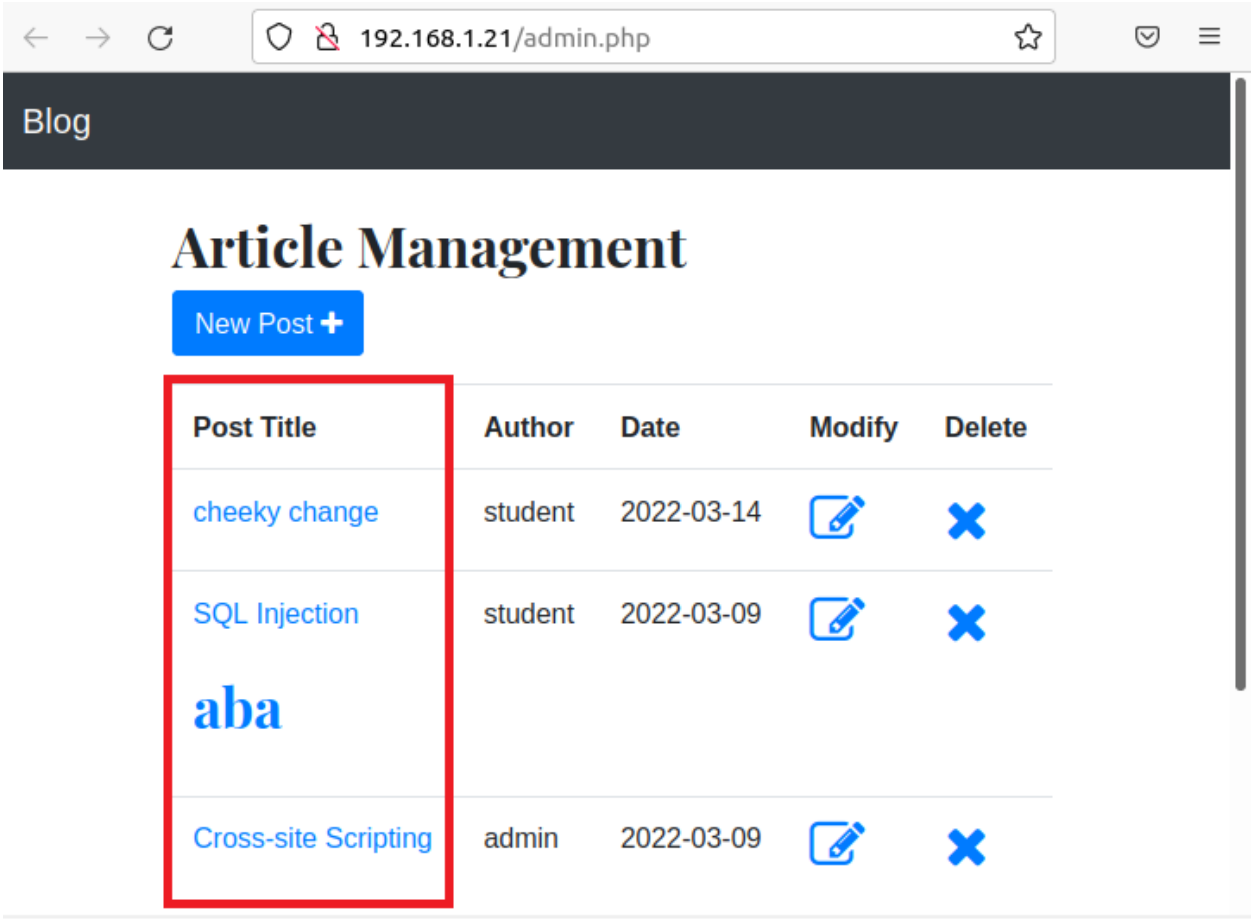
(Screenshot above shows two articles created, cheek change will be the CSRF exploit)

Since XSS is shown to work in the previous exploit, we embed HTML code in which we redirect to the article called “deletearticle.php” in which we will delete the article called “bob”. See screenshot below:





(Screenshot above shows the image link to the malicious code when opening in a new tab)



(Screenshot above shows that the article “bob” is deleted once user clicks on the image link to the malicious code as aforementioned above)

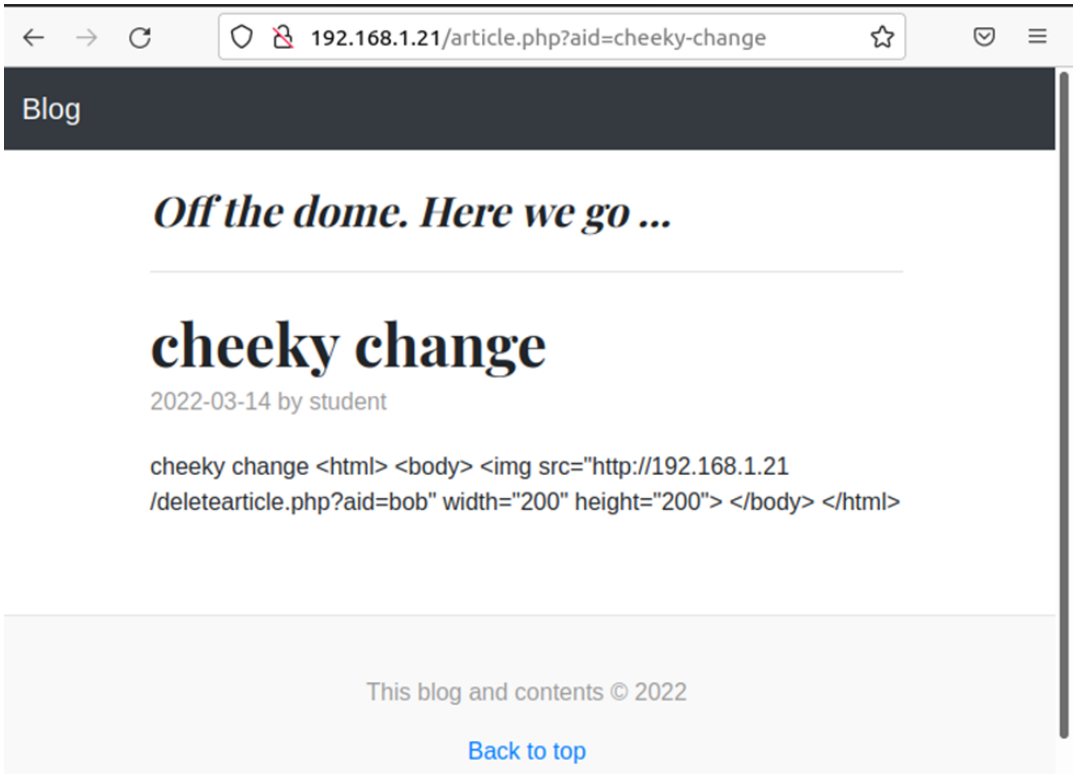
Explanation: The severity and impact of this vulnerability is that it allows the attacker to submit their own request to an authenticated application without the user’s acknowledgement. As you can see, students or anyone can delete articles by a simple XSS injection exploit as aforementioned.

Prevention and Mitigation: One simple way of doing this is by removing escape characters from being entered into the text field as aforementioned in V-1. This is done by using the htmlspecialchars() functions.

Security Fix: The security fix was to encode the htmlspecialchars() function into the text field area.

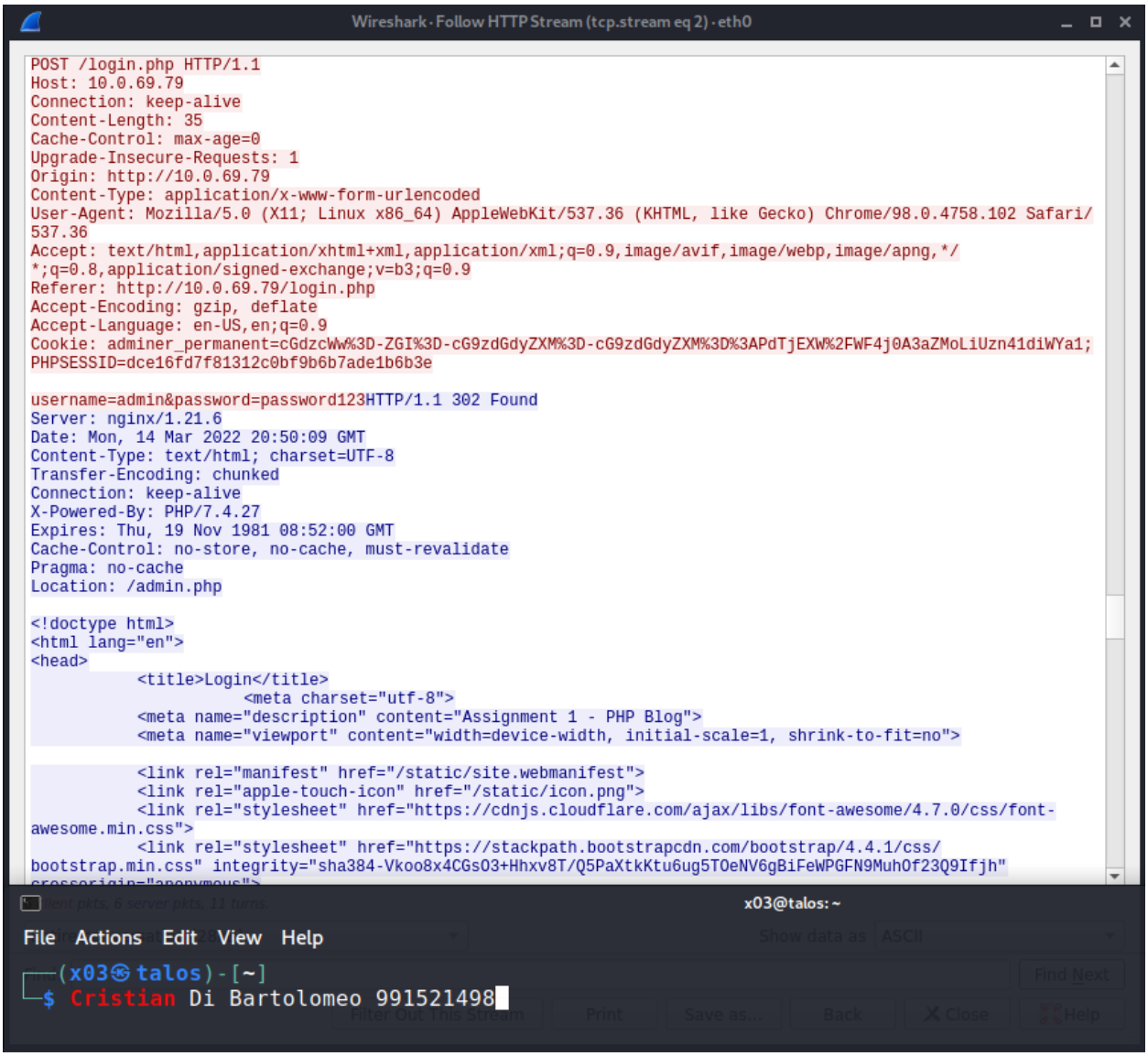
editarticle.php | reflected.php | article.php

```
15 <head>
16 <title><?php echo htmlspecialchars($row['title'], ENT_QUOTES, 'UTF-8'); ?></
   title>
17         <?php include("templates/header.php"); ?>
18
19
20
21 </head>
22 <body>
23         <?php include("templates/nav.php"); ?>
24         <?php include("templates/contentstart.php"); ?>
25
26         <h3 class="pb-4 mb-4 font-italic border-bottom">
27         Off the dome. Here we go ...
28         </h3>
29
30         <div class="blog-post">
31         <h2 class="blog-post-title"><?php echo
    htmlspecialchars($row['title'], ENT_QUOTES, 'UTF-8'); ?></h2>
32         <p class="blog-post-meta">
33                 <?php echo substr($row['date'], 0, 10)." by ".
   $row['author'] ?>
34         </p><p>
35                 <?php echo htmlspecialchars($row['content'], ENT_QUOTES,
    'UTF-8') ?>
```



(Screenshot above shows escape characters being printed out as a string)

V-7: The entire website, including the login page, is served over plaintext HTTP

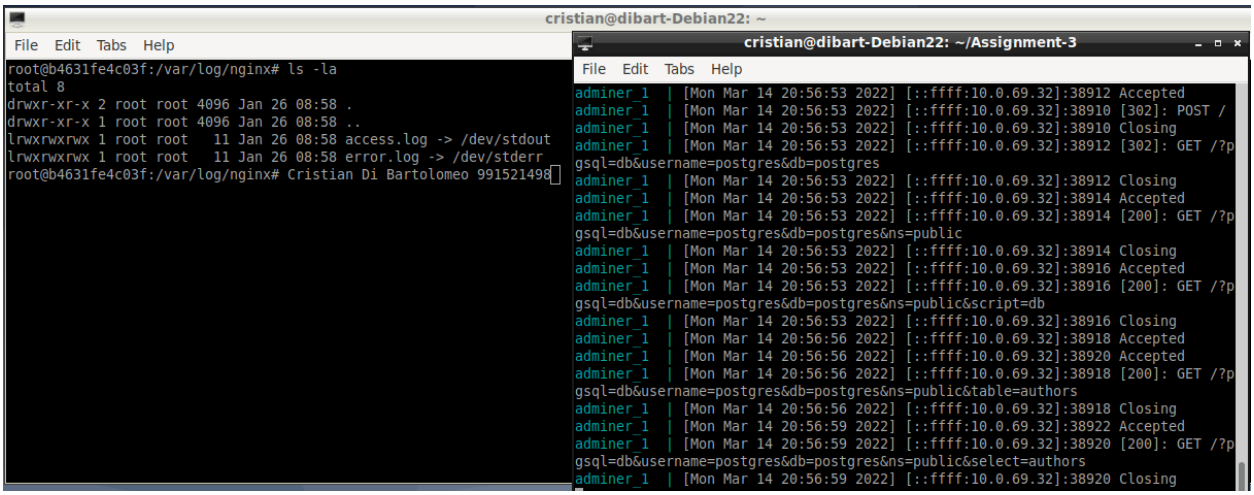


(Screenshot above shows evidence of password found through Wireshark when login)

Explanation: The severity and impact of this exploit is extremely high. As you can see in the above screenshot, the password and username are shown in plaintext through packet capture. That means that attackers can easily obtain credentials through packet capture with a very low skill and low effort attack.

Prevention and Mitigation: The protocol being used is HTTP, which is not encrypted. In order to prevent this, the administrator must upgrade the HTTP protocol to HTTPS, which uses SSL/TLS to encrypt traffic. As a result, the encrypted traffic prevents malicious actors from reading sensitive data. This can be done in multiple ways. Most modern web services (i.e. GoDaddy, NamesCheap) offer HTTPS certificates which can be used.

V-8: The web application has no logging except for the default logs generated by Nginx.

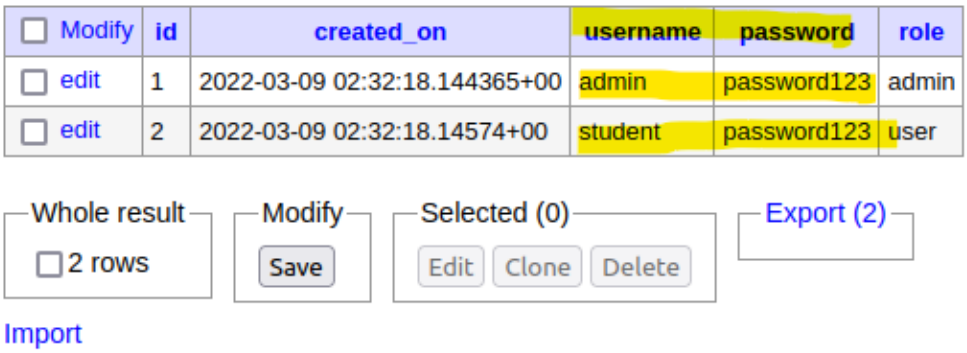


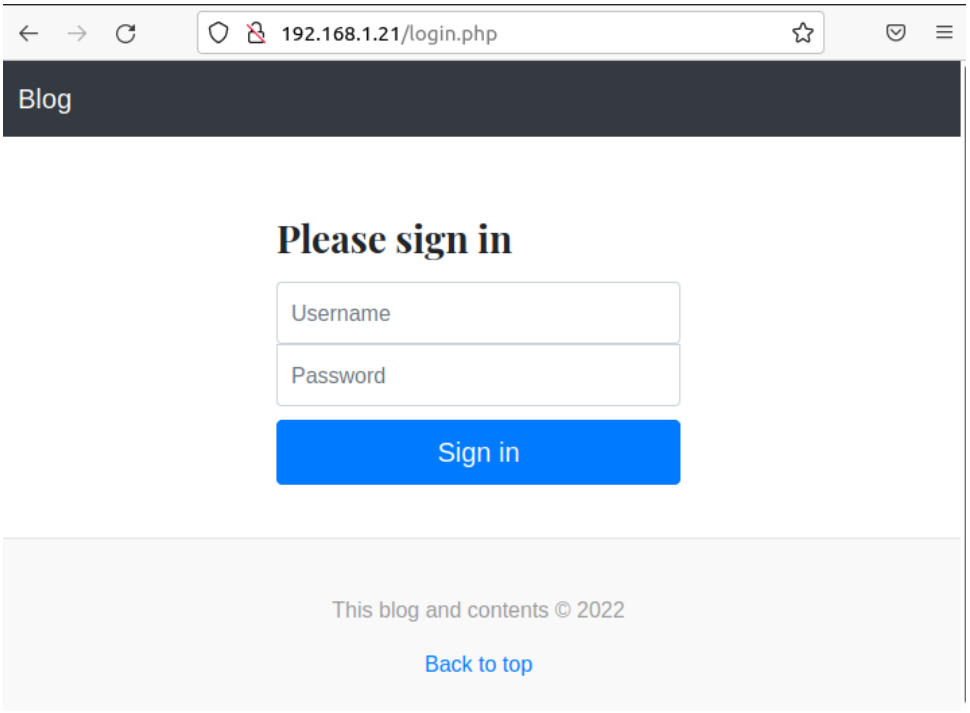
Explanation: The logs captured by the webpage are only the default Nginx logs (access.log and error.log) which appear to only be sent to stdout and stderr, meaning that they are never stored. As a result, capturing important events such as administrator changes or actions against the webpage are not easily accessible or may not even be possible. Backtracking something like an attack could be extremely hard or impossible if something like a restart happens, making this extremely severe.

Prevention and Mitigation: Adding logging functionality all depends on the service being used. In this case, Nginx is being used. To add custom logging to Nginx, you can edit Nginx's configuration file (/etc/nginx/nginx.conf) and use the "log_format" parameter. log_format will edit the default format of the logs, and you can choose the log file location for both access.log and error.log with "access_log" and "error_log" in the configuration file.

V-9: The website only uses single-factor authentication

Explanation: The website itself has only one way to authenticate the user which is by using the same password ("password123") for both admin and student.





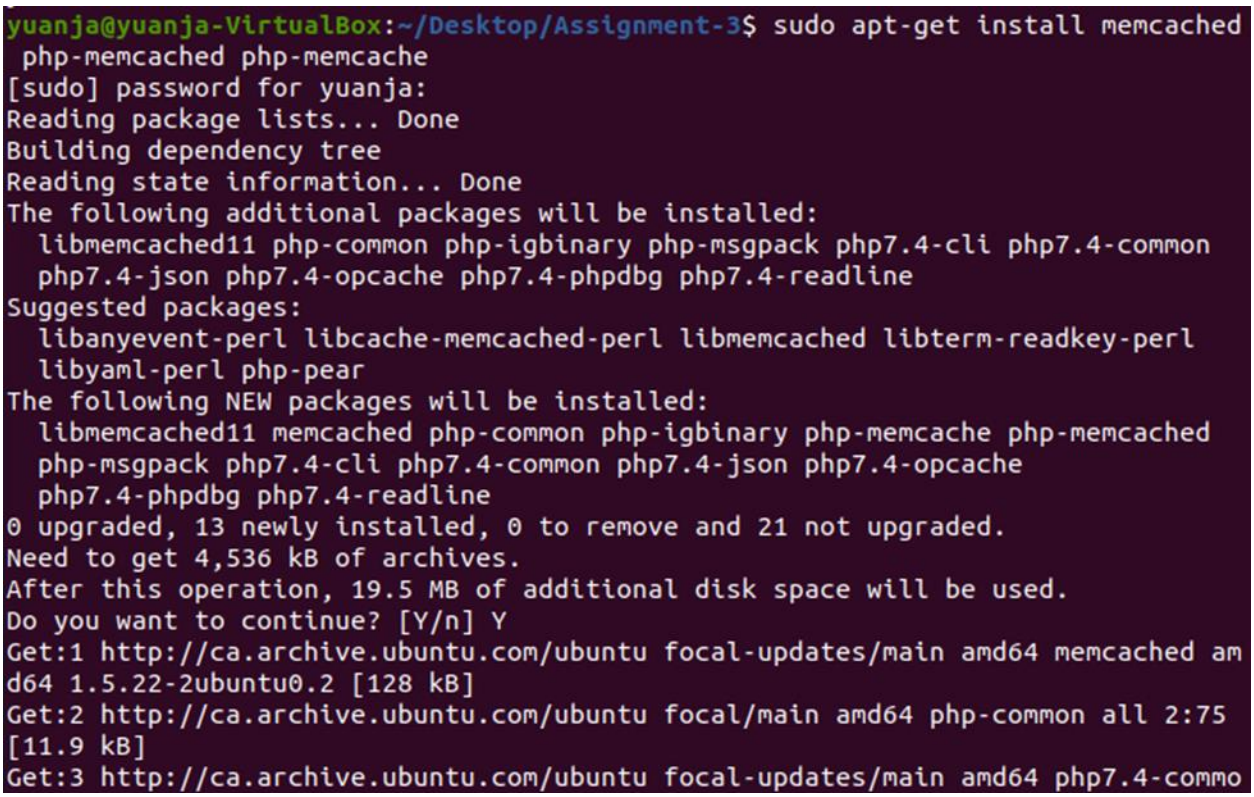
(Screenshot above shows no other option to authenticate the user)

Severity and Impact: The severity and impact is that its susceptible to 4 types of password attacks, 1) brute force attacks (guess all possibility) 2) dictionary attacks (try all words in English language first) 3) hybrid attacks (different variations combined) 4) rainbow table attack (uses precomputed hashes to attack).

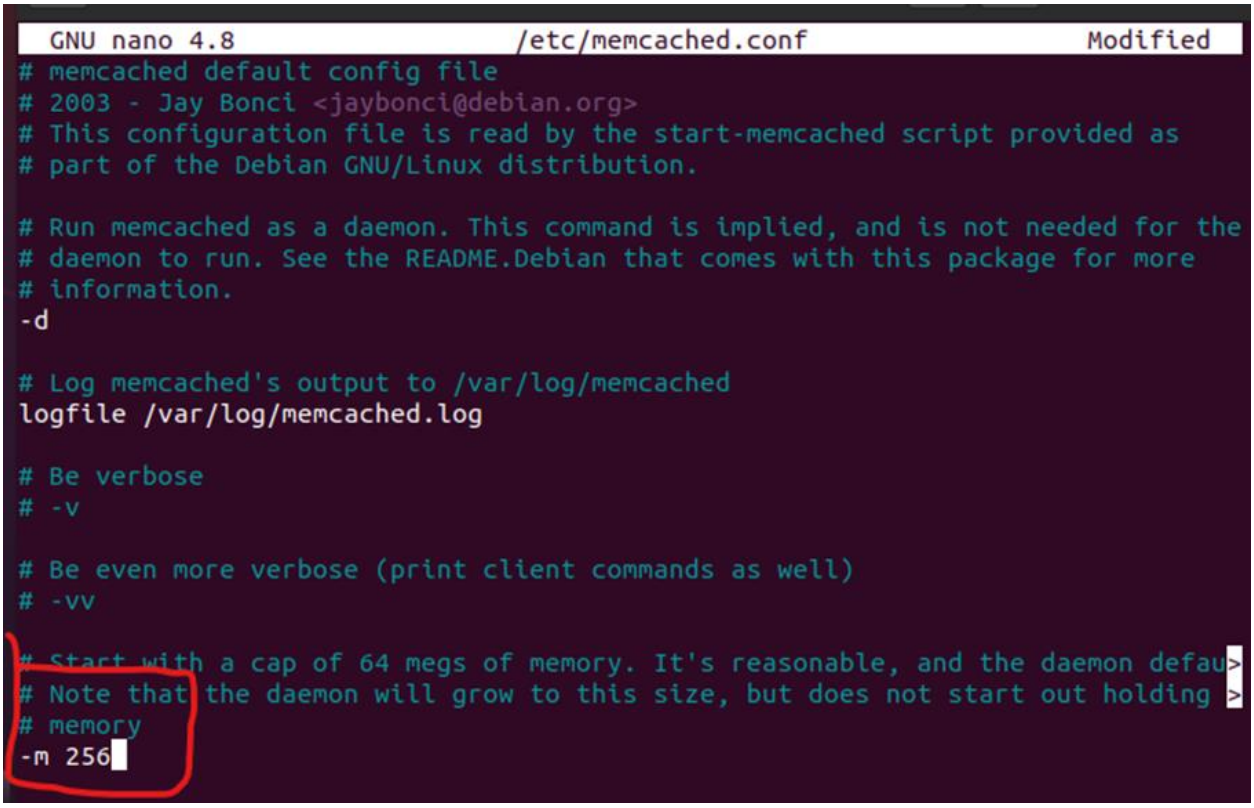
Prevention and Mitigation: To prevent such a thing, we should use MFA (Multi-factor authentication) such as implementing something the user has (keys, debit/credit cards, security tokens, etc.), something the user knows (passwords/PIN's) or something the user is (biometrics).

Security Fix: The fix for this would be to redesign the website in such a way where we would add additional buttons in which we can further verify the user.

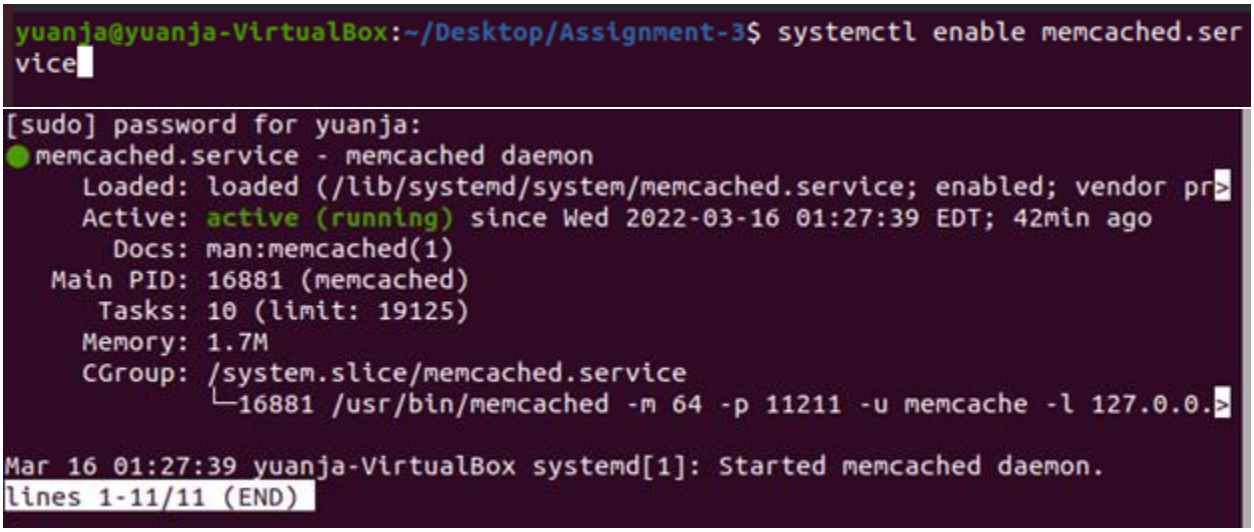
Bonus Mark #1 - Implement database caching using Memcached



(Screenshot above shows installing memcached and also php memcached for php)



(screenshot above shows editing the memcached.conf file to increase the cached from 64mb to 256 mb



(Screenshot above, we enable memcached.services)

Bonus Mark #3 - Allow Articles to be written using simple HTML tags & prevent the use of malicious JavaScript

First of all, the way to allow HTML tags to be written without exposing us to the vulnerabilities is to first, remove the JavaScript escape characters by calling JSON.stringify() function, what this essentially does is it will serialize and deserialize JavaScript objects. Second, we will escape the “<” by implementing it with

a Unicode after the serialization process is achieved. We can also apply this escaping to HTML character. However, the problem we encounter with this is that it will spoil our data, to fix this we store the information within a variable in which we call the getElementById and getAttribute to retrieve the information. Another way we can stop malicious code is to add a nonce attribute to prevent execution of scripts without that nonce attribute. Finally, the last fix would be the usage of the <safescript> tag. What <safescript> does is it will take any JavaScript code in between the <safescript> and carry its values as a string object.