

Security of Information and Organizations - 2021/2022

Assignment No.1 Report

SIO- Vulnerabilities Report

Professor: João Paulo Barraca

Student: Filipe Adão nº80063
Student: Gonçalo Arieiro nº80130
Student: Bruno Castro nº80190
Student: Diogo Maduro nº80233

Class: P3 e P8

Index

1	Introduction	3
2	Methods and Technologies	4
3	Discussion	5
3.1	Stored XSS	5
3.2	Reflected XSS	7
3.2.1	Url Obfuscating	8
3.3	Weak Password Requirements	8
3.3.1	Python Brute Force	9
3.4	Unverified Password Change	10
3.5	Exposure of Sensitive Information to an Unauthorized Actor	11
3.6	SQL Injection	12
4	Conclusions	16
5	References	17

1

Introduction

This report is made for the first project of the course with a theme of Security Vulnerabilities.

Information changes the world, and the rapid evolution of technology has created a huge impact on the importance of protecting data, making it a multi-billionaire market each year.

Due to this rapid evolution of technology systems every year that passes security topics increase in importance as does the theme of this project.

In this project, we made a web application based on an album shop with two variants, one vulnerable-app, we purposely left out certain vulnerabilities discussed below, and one with these vulnerabilities corrected.

This had the purpose of showing ways of resolving security threats in real-world applications but with academic intents in mind.

We started this project with two primary objectives in mind. Firstly, we wanted to create a simple skeleton web-service to use as a base for our second and major objective, exploring security topics within our application. We started by developing the version of the web app that became the vulnerable one, and after created the safe application using the first one as a template, solving issues as needed.

2

Methods and Technologies

Since the focus of this work was on a security topic such as Vulnerabilities, we decided earlier on that we wanted a basic API, simple enough that would enable us to add/remove security features and play around with the app, trying to breach it and find exploits.

To achieve this goal, we choose node with express framework running on top as our web server. For managing data, we went with MySQL.

As a view engine, we, again, wanted something simple and fast to implement, so pug was the answer.

These factors allowed us to focus on the real goal of this work.

3

Discussion

3.1 Stored XSS

Since there's no such thing as a secure application, we focused on certain CWE definitions, adding them when needed to the insecure app and fixing them on the secure version. Starting off with Cross-site Scripting, CWE-79, our insecure app has two different kinds of XSS attacks.

“Stored XSS” (type 2), which consists in the application storing dangerous data in the database which will later be loaded by users dynamically, executing the malicious content.

In our application, this vulnerability affects the `add_review` functionality, when a user writes a review for an album, they can choose both a title and a body of the review. Purposely, we left out this vulnerability on the title field, allowing users to post a review with a malicious script on the title. Which will be executed by any user that visits the detailed view of an album page (`/shop/albumID`).

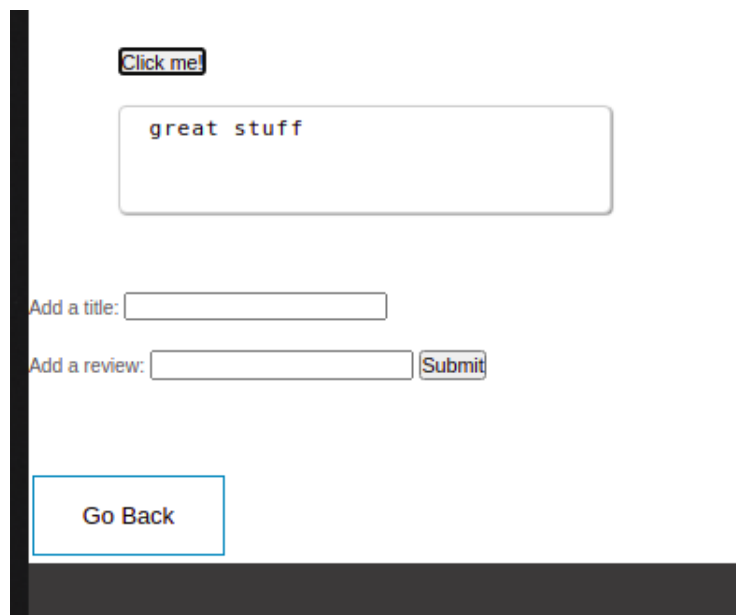


Figura 3.1: *unsafe-app-stored-xss*

```
<button onclick="alert(1)">Click me!</button>
```

Figura 3.2: *unsafe-app-stored-html*

As the image shows, we can exploit this vulnerability by adding a button tag with an onclick method to run a script or anything we want. We can't do the same on the content field, however, since we properly neutralize the input. Comparing both versions of the app when it comes to this source of attack, we can see that the difference resides on the frontend page that is rendered when Users visit this route.

In the insecure app we render the title field as raw html data, in pug we can do this with:

```
<h2>#{review.title}</h2>
```

Figura 3.3: *unsafe-app-frontend*

On the opposite spectrum when we want to escape some malicious title, we can use this pug syntax which will treat the field as text, thus removing any chances of it being a script:

```
<h2>#{review.title}</h2>
```

Figura 3.4: *safe-app-frontend*



Figura 3.5: *safe-app-neutralization-example*

We do this on both versions of the app in the review field (Content of the review) which does not allow for any scripting, as mentioned above.

wont work, it's neutralized

```
<script>alert(1)</script>
```

Add a title:

Add a review:

Figura 3.6: *both-apps-netralization-example*

3.2 Reflected XSS

The second type of XSS Vulnerability, on our insecure app is “Reflected XSS” (type 1), this attack exploits the fact that the server reads data directly from the HTTP request and uses it on HTTP response without any care for validation or neutralization.

The search functionality in our insecure app suffers from this, since it takes the query parameter directly from the request and sends it to our frontend which renders it without any neutralization, allowing for malicious code to be executed as soon as the user clicks the “evil” link.

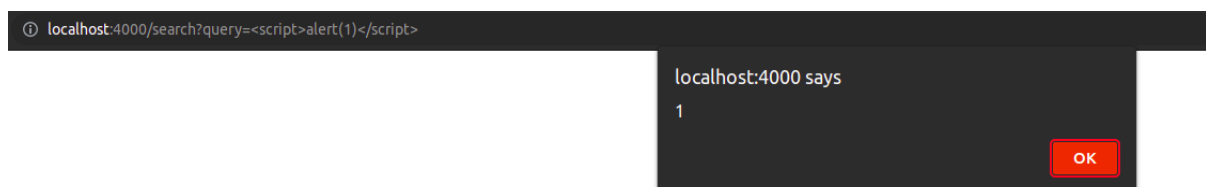


Figura 3.7: *unsafe-app-reflected-xss*

```
albums=result;  
res.render('search',{searched: req.query.query, albums : albums});
```

Figura 3.8: *unsafe-app-backend*

```
h1 You searched for : !{searched}
```

Figura 3.9: *unsafe-app-frontend*

In the safer version we also take this parameter directly from the request and send it to our frontend but this time we neutralize this input as text, removing the chance for a Cross-Scripting attack of this nature.

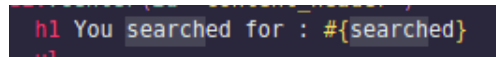


Figura 3.10: *safe-app-frontend*

3.2.1 Url Obfuscating

To increase chances of this type of attack working, obfuscating techniques are usually used to hide the malicious url from the User who's clicking it.

This is usually done by encoding the url in some format that is decoded automatically by the browser, a popular choice is encoding it in ASCII.

For a simple example, the malicious url:

`http://localhost:4000/search?query=<script>alert(1)</script>`

Can simply be encoded to:

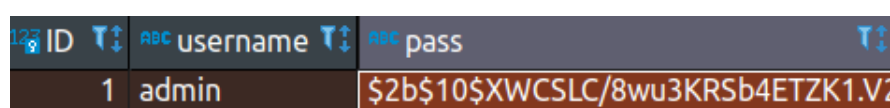
`http://localhost:4000/search?query=%3Cscript%3Ealert%281%29%3C/script%3E`

This second link hides some of the malicious part of the url, making it more likely that a naive user will click on it.

3.3 Weak Password Requirements

Another vulnerability we decided to focus on was Weak Password Requirements, CWE-521. This definition states that specific requirements for how complex and long a password is, need to be carefully chosen and implemented. This seems obvious these days, but in this project we did some experimentation with a “random” implementation of a brute force script to test how the length and complexity of a password would affect the app's security. We will talk about it below.

In addition to requiring extra complex and long passwords, the secure version of the app also does not store the password as plain text on the database but the hashed version of it, adding another layer of security. This also solves another vulnerability, CWE-256, Plaintext Storage of a Password, which our insecure app is guilty of.

A screenshot of a database table view. The table has three columns: 'ID', 'username', and 'pass'. The first row contains the values '1', 'admin', and a long, complex hash string: '\$2b\$10\$XWCSLC/8wu3KRSb4ETZK1.V2'. The 'pass' column is highlighted with a red border.

ID	username	pass
1	admin	\$2b\$10\$XWCSLC/8wu3KRSb4ETZK1.V2

Figura 3.11: *safe-app-usersTable-database-view*

3.3.1 Python Brute Force

To demonstrate the importance of minimum password requirements and also play around with the idea of applying brute force techniques to password cracking we created a little, rudimentary python script, located at `/analysis/bruteforce.py`.

This is a simple script that receives 4 inputs, the length of the password, the username to crack, type of alphabet to use and the url to exploit.

The idea is that the user of the script knows at least one account, probably admin exists, but he does not know the length of the admin's password.

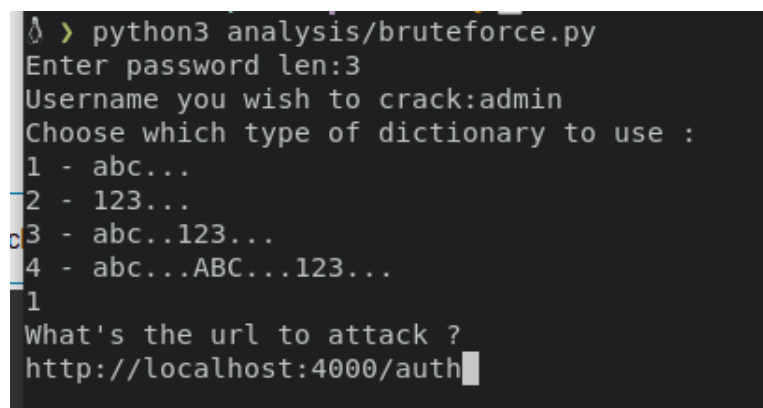
On the other hand he knows, on the insecure app, that for registering an user there are no minimum length or complexity requirements for a password, so he tries to crack the password for the admin account starting with the simplest alphabet and passwords of one digit, then two, etc.

This script is implemented with random access to the array containing all characters of the alphabet chosen.

This is done this way to prove that even with such "silly" and simple code, small and not complex passwords are easily cracked in reasonable time.

Here we can see a crack attempt of a 3 digits password and a very simple alphabet of just lower case characters. Even with completely random access this password was cracked in less than a minute.

Obviously this time increases dramatically if we increase the length of the password or the possible alphabet.



```
> python3 analysis/bruteforce.py
Enter password len:3
Username you wish to crack:admin
Choose which type of dictionary to use :
1 - abc...
2 - 123...
3 - abc..123...
4 - abc...ABC...123...
1
What's the url to attack ?
http://localhost:4000/auth
```

Figura 3.12: *script-view-menu*

```
cff
gas
sms
xuu
use
lka
pay
mvo
pte
gjz
ole
We got the password for admin, it is : ole
--- 0.29141454299290975 minutes ---
```

Figura 3.13: *script-view-attack*

```
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone tried to login with wrong data to admin account
someone just logged in sucefully to admin account
[ RowDataPacket { ID: 1, username: 'admin', pass: 'ole', admin: 1 } ]
```

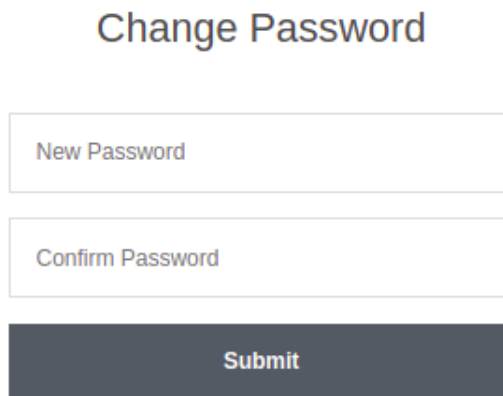
Figura 3.14: *server-view*

As mentioned above, this script is simple and basic just to prove how important strong passwords are. This could easily be a much more robust script that would not be random but sequential.

It's the norm to also be using a dictionary of commonly used words in the English Language (per Example) to increase cracking speed.

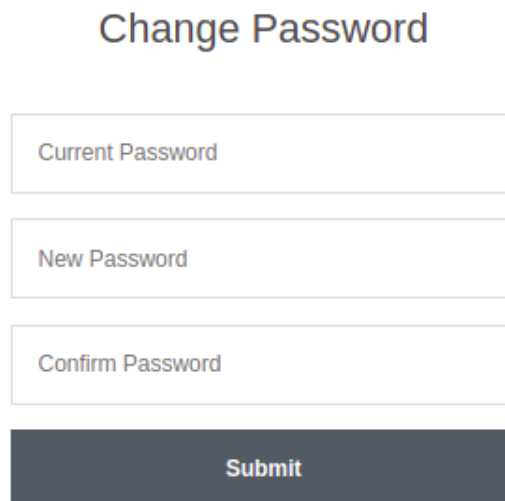
3.4 Unverified Password Change

Keeping with the password subject, let's look at CWE-620, Unverified Password Change. It states that when setting a new password for a user, there should be a confirmation of the current password or another form of authentication. We implemented this check for the current password on the secure version of our app.



A web form titled "Change Password" with two input fields: "New Password" and "Confirm Password", followed by a "Submit" button.

Figura 3.15: *unsafe-app-unsafePassword-example*



A web form titled "Change Password" with three input fields: "Current Password", "New Password", and "Confirm Password", followed by a "Submit" button.

Figura 3.16: *safe-app-safePassword-example*

3.5 Exposure of Sensitive Information to an Unauthorized Actor

We also looked at CWS-200 and found it relevant to our application. This definition states that we should not expose sensitive information to someone who's not explicitly authorized to have access to this data.

This can be personal information regarding certain users, internal state of the application or in our case metadata.

In the insecure version of our app, we define the database connection parameters and Express-Sessions Secret right on `index.js` which, obviously, it's the wrong approach, since it keeps private and critical information on the repository and source code of our web-app.

To combat this, in the secure version we made some changes. Firstly we use a package called `dotenv` to facilitate the setting of environment variables with values that we want to be set only by the host provider server at startup.

This allows us to create a .env file in our machine with all these secret variables that are then imported to the server.

For this project, this file is uploaded to the repository, this goes against protocol (we should not grant access to outside entities information on how we set our environment variables or their values), this is done strictly for development and academic purposes only.

```
var connection = mysql.createConnection({
  host: process.env.DBHOST,
  user: process.env.DBUSER,
  password: process.env.DBPASS,
  database: 'music_sec'
});
```

Figura 3.17: *safe-app-backend-1*

```
app.use(session({ resave: true ,
  secret: process.env.SESSION_SECRET,
  saveUninitialized: true}));
```

Figura 3.18: *safe-app-backend-2*

```
🔒 .env
1 DBHOST=localhost
2 DBUSER=root
3 DBPASS=your_new_password
4 SESSION_SECRET=123456
5
```

Figura 3.19: *safe-app-dotenv-file*

3.6 SQL Injection

Finally, we took a look at SQL injections, CWE-89. There exists the need to validate the data coming from user controlled inputs before sending it to a database query. This prevents some malicious inputs from being interpreted as SQL code and not data. This software flaw is easily exploitable but easily fixable as well. It's usually used to manipulate backend data and access sensitive information. Defusing this flaw could be achieved in various ways, we decided to implement parameterized queries as our sql syntax.

The difference between this approach and using a constant value, is that parameterized queries only take value when the query is executed. This way all the SQL code is defined first, leaving the parameter to always be interpreted as data, regardless of what input the user supplies.

```
FROM Users WHERE username = ?'
```

Figura 3.20: *safe-app-parameterized-query*

```
FROM Users WHERE username = '"+username+"'
```

Figura 3.21: *unsafe-constant-value-query*

In our app we have two vulnerable functions to exploit via sql injection, login and add album functionality. On the insecure version, we can close the query for the login field username, since there is no validation of user input and then apply a big range of techniques to manipulate data on the database. To demonstrate, let's show some very simple examples¹.

In this example, the malicious user knows or assumes there exists an account named admin so he gives it as username parameter but also escapes the query by adding ' . Then the user, who doesn't know the password for admin, just adds a line comment after admin' , telling sql server to ignore the rest of the statement. This will result in a successful query and login.

Login

Not a member? [Sign Up](#)

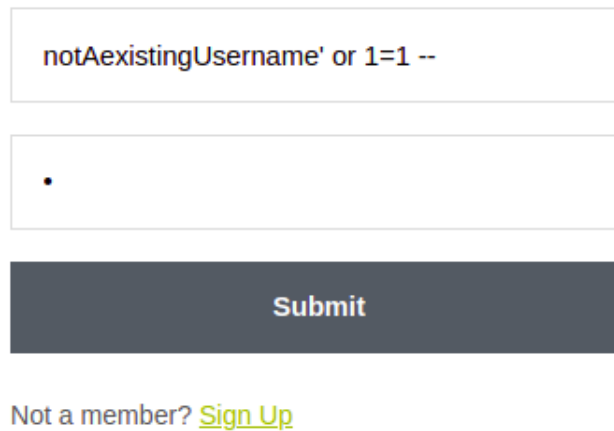
Figura 3.22: *unsafe-app-sqlInjection-login1*

We can also exploit the login without knowing any username from the database. We accomplish this by inputting a random string and escaping the input, which will result in a False evaluation from sql but we add OR <True statement> (e.g. 1=1, which will be evaluated as True) and finally a line comment. Since 0 or 1 always equals to 1, True, we will access and get a successful login.

In the case, shown below, the query actually returns all the users.

¹In all examples below, there needs to be a space after the double dash and a value for password, it can be a random value but it needs to be something for this to work.

Login



The screenshot shows a login interface. The top input field contains the SQL injection payload: `notAexistingUsername' or 1=1 --`. The bottom input field contains a single dot: `.`. Below the input fields is a dark grey button labeled "Submit". At the bottom of the form, there is a link that says "Not a member? [Sign Up](#)".

Figura 3.23: *unsafe-app-sqlInjection-login2*

```
"SELECT * FROM Users WHERE username = '"+username+"' AND pass = '"+pass+"'"
```

Figura 3.24: *unsafe-app-query*

The second via of attacks of this type is in the add album functionality. Here we are asked to provide a title, a genre and an artist name for the album. We can again escape any input field, add some more logic to close the brackets and then using stacked queries², we can, again, maliciously manipulate data from the backend.

To exemplify here are two very simple examples of sql injection we can do in the insecure app, add album functionality.

```
');delete from albums where artist = 'Frank Sinatra'; --
```

Figura 3.25: *sqlInjection-example1*

```
');insert into albums (title,artist,genre) select username,'null',pass from users; --
```

Figura 3.26: *sqlInjection-example2*

In the first example we can see a simple injection that will delete all albums with the artist value set as 'Frank Sinatra'. In the second one, we populate the Album table with data from the Users table, completely revealing all the sensitive account information.

²mysql does not support this by default, we added the `multiplestatements:true` parameter to the creation of our database to allow this

We could also drop the database or do numerous malicious actions, these two only serve as simple examples.

4

Conclusions

We consider that we have addressed all the goals requested for this assignment since we developed two functional applications, an application in which the existence of vulnerabilities can be exploited and one additional application that avoids these vulnerabilities. Several methods were implemented in the secure application to guarantee strong passwords and some type of verification when the passwords are changed. Methods to prevent cross-site scripting, SQL injection and the exposure of information to an unauthorized actor were also implemented. Both applications are working as expected.

Thus, this project served to deepen and condense the techniques, methods and systems learned during the classes.

5

References

- [1] CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-Site Scripting'), June 2006
<https://cwe.mitre.org/data/definitions/79.html>

- [2] CWE-521: Weak Password Requirements, July 2006
<https://cwe.mitre.org/data/definitions/521.html>

- [3] CWE-620: Unverified Password Change, May 2007
<https://cwe.mitre.org/data/definitions/620.html>

- [4] CWE-200: Exposure of Sensitive Information to an Unauthorized Actor, July 2006
<https://cwe.mitre.org/data/definitions/200.html>

- [5] CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection'), July 2006
<https://cwe.mitre.org/data/definitions/89.html>

- [6] CWE-256: Plaintext Storage of a Password, July 2006
<https://cwe.mitre.org/data/definitions/256.html>