



universidade de aveiro

# Security of Information and Organizations

2022/2023

Professor João Paulo Barraca

Professor Alfredo Matos

Departamento de Eletrónica, Telecomunicações e Informática

## eHealth Corp

Ana Raquel Paradinha	102491
Paulo Pinto	103234
Miguel Matos	103341
Tiago Carvalho	104142

All students contributed equally to the project.

# Index

<b>Index</b>	<b>1</b>
<b>1. Introduction</b>	<b>2</b>
<b>2. System description</b>	<b>3</b>
<b>3. Explored vulnerabilities</b>	<b>4</b>
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	4
CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	5
CWE-522: Insufficiently Protected Credentials	6
CWE-521: Weak Password Requirements	7
CWE-352: Cross-Site Request Forgery (CSRF)	8
CWE-20: Improper Input Validation	9
CWE-434: Unrestricted Upload of File with Dangerous Type	10
CWE-200: Exposure of Sensitive Information to an Unauthorized Actor	10
<b>4. Conclusion</b>	<b>12</b>
<b>5. Bibliography</b>	<b>13</b>

# 1. Introduction

The scope of this assignment was the analysis and avoidance of existing vulnerabilities in a software project. To do this, we created eHealth, a web application focused on managing a health clinic.

Firstly, we created an app without security in mind, the unsecured app, that contained some vulnerabilities that were not clear to the users. After that, we created a secure version of the first app, where we corrected those vulnerabilities.

## 2. System description

Using our system, the user is able to register (sign up) himself in the system, as well as, login, once the account is created. Once the user is authenticated, he can start using some system features, such as, requesting an appointment, getting access to his exams results, accessing the blog, etc...

In terms of technologies, our applications were created using *Flask*<sup>1</sup> framework, a *Python* module that helped us develop the *API*'s, for the back-end; and *Bootstrap* components as well as the common web development kit (*HTML*, *CSS*, *JS*), for the front-end. Our database was created using *MySQL* and it's common for both applications. The project is contained in a *Docker-Compose*, in order to facilitate its execution.

Speaking of the file tree, both applications have a similar structure. Most of the front-end is located inside the templates folder, containing the *HTML* components, and the static folder, containing all the fonts, images, icons and *CSS* and *JS* files. The file *api.py* is basically the main file, in terms of app functionalities. It is responsible for handling the *HTTP Requests* on the *API* and returning a proper *Response View*. The file *db.py* is responsible for establishing the connection with the database.

1. Flask is a web framework, it's a Python module that lets you develop web applications easily. It has a small and easy-to-extend core: it's a microframework that doesn't include an ORM (Object Relational Manager) or such features. It does have many cool features like url routing, template engine. It is a WSGI web app framework.

### 3. Explored vulnerabilities

#### CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')

##### Description

The application does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.

##### Application

Our application features a blog, where users can freely comment on posts. These comments are then displayed in the web page for other users to see.

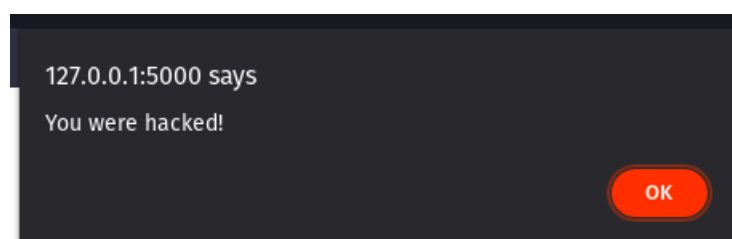
This means that commenting something like

```
<script> alert("this is a cross-site scripting attack") </script>
```

would make anyone that loads our comment in our web page see an alert on their screen. This can be used to load scripts from other websites, by url, and force the users accessing the web page to execute them.

##### Prevention

To prevent these types of attacks, all input needs to be sanitized before being displayed on the web page, or properly escaped so as to not be executed when loaded. In our app, all input sanitizing is done by *Flask* automatically. However, *Flask* allows us to disable this protection, which we did in our unsafe application counterpart. Commenting the example above will lead to the mentioned result, because the comment text is being loaded as an html script tag, and not as plain text.



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

### Description

The software constructs all or part of an *SQL* command using externally influenced input, but doesn't neutralize special elements that may modify the intended *SQL* command when sent to the database.

### Application

For example, you can login without having an account, you can make use of *SQL* injection. Entering

`" ' or 1=1; --"`

on the email camp, and leaving the password empty, the *SQL* command used to check if your account is registered is modified from

`"select * from patients where email = '[email field]' and password = '[password field]'"`

and turns it to

`"select * from patients where email='' or 1=1; --' and password='' "`

which means that the email and password checks are ignored, and because `1=1` is always true, you will successfully login.

This also allows for database manipulation. Manipulating the *SQL* command by inputting

`" ' or 1=1; update patients set password='password' where id=1; --"`

into the email field, effectively updates the password from the patient column with id 1 to `"password"`.

Email does not exist.

## Log In

Email

↑ or '1'='1'; --

---

Password

---

LOG IN

Don't have an account? [Sign Up](#)

## Prevention

To prevent *SQL Injection* attacks through input fields like these, we make sure to sanitize all input that comes from these. By allowing only certain characters, we can prevent malicious input from reaching the backend.

For example, an email can only consist of letters, numbers, dots (".") and underlines (\_), and the "@" character in the middle.

## CWE-522: Insufficiently Protected Credentials

### Description

The application transmits or stores authentication credentials, but uses an insecure method that is susceptible to unauthorized interception and/or retrieval.

```
mysql> select * from patients;
```

id	name	email	password	phone	profile_pic
1	Netty Wrefford	nwrefford1@tmall.com	KwKxuy	3355710502	NULL
2	Hali Donwell	hdonwell2@state.tx.us	DmBMnFoKEW	3788933004	NULL
3	Justinian Krebs	jkrebs3@bloglines.com	HiPyXTdQ4CrU	4282135927	NULL
4	Jorry Seiller	jseiller0@goo.net	UfzBESyz	4995358663	NULL
5	Leroy Esche	lesche4@theforest.net	SBo29PG	5197321309	NULL
6	Nady Jakubczyk	njakubczyk5@phoca.cz	7IjF4q1Kk	8614307489	NULL
7	Eugene Bernhardt	ebernhardt6@lund1.de	EPVSCJzyZZ	7701289577	NULL
8	Roda Shephard	rshephard7@aboutads.info	esZHeHNVUS	7599935717	NULL
9	Ilario Chritchley	ichritchley8@php.net	XmVoikD6	3552780449	NULL
10	Melony Grimmert	mgrimmert9@taobao.com	a3woSc6	3702735495	NULL

## Application

In our unsafe application counterpart, passwords were plainly stored in the database as a string. If our input on the password field were to be “superSafePassword”, this data would be stored as the same string in the database. This potentially exposes login credentials to anyone who can access the database through any means.

## Prevention

To solve this issue, we run the passwords through a hash (*SHA256*) before storing them. With this, our password from before would be stored as

`89946423345031ecc1a694f9ed98ea92bf602e1c1b29360468e1a11e6a49038b`

which adds another layer of security to our credential storage.

## CWE-521: Weak Password Requirements

### Description

The product does not require their users to use strong passwords, which makes it easier for attackers to compromise user accounts.

### Application

In our unsafe application counterpart, a password can be any string you want. For example, a string like “password” would be allowed, which means that a simple guess could compromise the account.

### Prevention

In our safe application counterpart, we have a wide range of password requirements, such as:

- minimum of 8 characters;
- at least one lowercase letter;
- at least one uppercase letter;
- at least one number;
- at least one special character (`_`, `@` or `$`).



```

def checkPass(password):
    flag = 0
    message = ""
    if len(password) < 8:
        flag = -1
        message = "Password should be bigger than 8 characters."
    elif re.search("[a-z]", password) == None:
        flag = -1
        message = "Password should have at least one lower case letter."
    elif re.search("[A-Z]", password) == None:
        flag = -1
        message = "Password should have at least one upper case letter."
    elif re.search("[0-9]", password) == None:
        flag = -1
        message = "Password should have at least one number."
    elif re.search("[_@$]", password) == None:
        flag = -1
        message = "Password should have at least one special character (_, @ or $)."
    elif re.search("\s", password) != None:
        flag = -1
        message = "Password shouldn't have whitespaces."

    return flag, message

```

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

### Description

The application does not, or can not, sufficiently verify whether a well-formed, valid, consistent request was provided by the user who submitted the request.

When a web server receives client requests without a mechanism to verify if those requests are made intentionally, an attacker can make unintentional requests through the user.

### Application

The application can be exploited in a way similar to CWE-79. By commenting something that will be interpreted as code and not plain text, one can request execution of external code when loading the web page with the malicious comment.

### Prevention

Once again, we use Flask to escape all data that comes from a source managed by the user, so that no input might be treated as code instead of text.

## CWE-20: Improper Input Validation

### Description

The application receives input or data, but it does not validate or incorrectly invalidates the input has the properties that are required to process the data safely and correctly.

When the software doesn't correctly validate user input, an attacker can create input that the application can't handle, because it is not prepared to do so. This can disrupt the normal application workflow, and can lead to resources being consumed and uncontrolled code to be executed.

### Application

When a user requests an appointment, they have to choose a date for it. However, in the unsafe application, no input validation is done. This allows, for example, to request an appointment for a past date, which would not only be pointless, but would occupy space in the database.

### Prevention

To prevent these problems, we make sure that all input is not only safe, but also context appropriate. For example, it's no longer possible to request an appointment for yesterday - it is automatically denied, and not stored in the database.

That date has already passed!

## Request Appointment

Select Department ▼

Select Doctor ▼

11/02/2022

I want an appointment for yesterday!

SUBMIT NOW

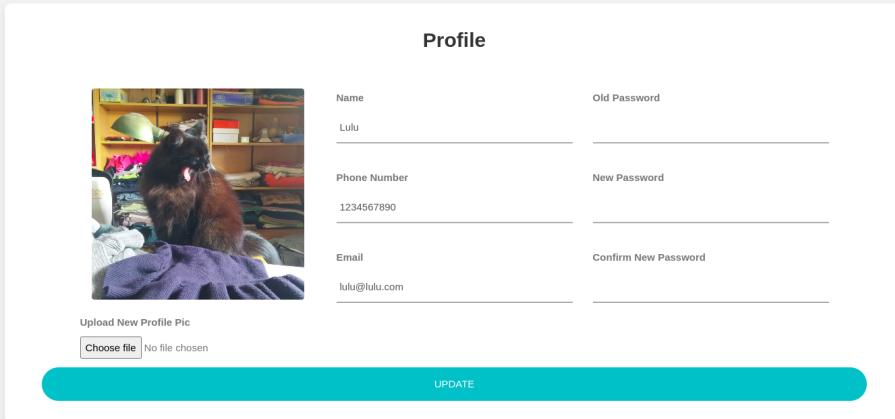
## CWE-434: Unrestricted Upload of File with Dangerous Type

### Description

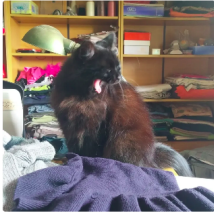
The application allows attackers to upload or transfer files of dangerous types that can be automatically processed within the application's context.

### Application

Our application allows users to upload a profile picture to their profile. In the unsafe application counterpart, no check was being run as to the nature of the file provided, meaning that an attacker could inject a file of malicious nature, or too big for the system to process.



**Profile**



Upload New Profile Pic  
 No file chosen

**Name**  
Lulu

**Old Password**

**Phone Number**  
1234567890

**New Password**

**Email**  
lulu@lulu.com

**Confirm New Password**

**UPDATE**

### Prevention

To prevent this issue, we only allow files of specific types to be submitted (namely *.png* and *.jpg*). Also, a maximum file size is set, taking into account the file nature that needs to be submitted.

## CWE-200: Exposure of Sensitive Information to an Unauthorized Actor

### Description

The application exposes sensitive information to an actor that is not explicitly authorized to have access to that information.

Exploiting this, an attacker can potentially gain access to confidential information about users or the application itself.

## Application

In the unsafe application counterpart, you may input a code (supposedly given to you by your doctor), and get access to your exam results through that code. However, if someone knows the code to your exam results, they may request and access them.

## Exam Results

Patient	Netty Wrefford
Result	You took an arrow to the knee!

## Prevention

In the safe application counterpart, we require the user to be logged in before requesting any exam results. We also check if the requested exam belongs to the logged in patient, and deny access if it doesn't.

That exam doesn't belong to you!

## Exam Code

Exam Code

---

REQUEST EXAM

## 4. Conclusion

Summarizing and concluding, the objective for this assignment was the analysis and avoidance of existing vulnerabilities in a software project and throughout this project, we applied the knowledge obtained in the theoretical classes about the vulnerabilities we corrected. We also resorted to online research, to increase our understanding about each vulnerability.

Besides that, we improved our coding abilities, using the *Flask* framework, our team working methodologies, as well as, our knowledge using *MySQL* and *Docker*.

So we are proud of our work and we think we accomplished the main objective for this work, making very clear the importance of a secure application.

## 5. Bibliography

For this project we used the following references:

- classes material;
- <https://cwe.mitre.org/index.html>;
- <https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator>;
- <https://flask.palletsprojects.com/en/2.2.x/>;