



deti

universidade de aveiro
departamento de eletrónica,
telecomunicações e informática

Project 2 - SIO

Application Security Verification Standard (ASVS)

Bernardo Pinto – 105926

Diogo Falcão – 108712

João Santos – 110555

Matilde Teixeira – 108193

(Class P5)

Index

Introduction.....	3
Web application compliance audit.....	4
- Error handling and logging.....	6
- Authentication.....	6
- Input Validation.....	7
- Files and Resources.....	8
- Configuration.....	9
- Cryptography at rest.....	14
- Data protection.....	14
Mandatory Implementations.....	15
- Password strength evaluation.....	15
- Encrypted database storage.....	19
Conclusion.....	21
References.....	22

Introduction

In this project we were asked to comply to a certain level to the Application Security Verification Standard (ASVS), which is a comprehensive checklist outlining essential security requirements and tests for various stakeholders as it acts as a structured framework, offering a set of standards and measures to shape the creation, validation, and assurance of secure applications, particularly for modern web and mobile applications.

We used the same DETI memorabilia online shop that we developed in project 1 and made it comply in a certain way to the level 1 Application Security Verification Standard requirements. We'll showcase our assessment of the web application, followed by its examination. We'll discuss our insights, the key concerns, and we will explain our decision on integrating two additional software features.

Web application compliance audit

In the Excel document, we filled the verification requirements with the level 1, and by that, we classified zero requirements in the Architecture tab, twenty eight in the Authentication tab, twelve in the Session Management tab, nine in the Access control tab, twenty seven in the Input Validation tab, one in Cryptography at rest tab, three in Error handling and logging tab, six in Data protection tab, three in Malicious code tab, three in Communication Security tab, five in Business logic tab, eleven in the Files and resources tab, seven in Web Services tab and sixteen in the Configuration tab.

	Valid criteria	Total criteria	Validity Percentage
Architecture	0	42	0.00
Authentication	4	49	8.16
Session Management	6	18	33.33
Access Control	4	9	44.44
Input Validation	15	28	53.57
Cryptography at rest	0	15	0.00
Error Handling and Logging	2	13	15.38
Data Protection	3	16	18.75
Communication Security	0	7	0.00
Malicious Code	1	8	12.50
Business Logic	1	8	12.50
Files and Resources	5	15	33.33
Web Service	3	12	25.00
Configuration	3	23	13.04
Total	47	263	17.87

Image 1 - ASVS result

For each one of the verification requirements, we tried to explain in the best way possible with the comments and tools needed, as it is possible to see in the image below.

Verification Requirement	Valid	Source Code Reference	Comment	Tool Used
Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined). ([C6](https://owasp.org/www-project-proactive-controls/#div-numbering))	Non-valid	pattern="(?!.*d)(?!.*[a-z])(?!.*[A-Z]).{8,}"	In register.jsp, we have a pattern which doesn't allow for less than 8 characters and not 12	Regular expression

Image 2 - ASVS Checklist

Nevertheless, our application has a relatively low validity percentage, averaging 18,36% considering all ASVS levels but with a percentage of 43,69% if only considering the level 1 ASVS level. Therefore, our application from the previous project was not as secure as we thought, but this insight provides a clear direction for improvement. Understanding these percentages allows us to pinpoint areas where our security measures need enhancement.

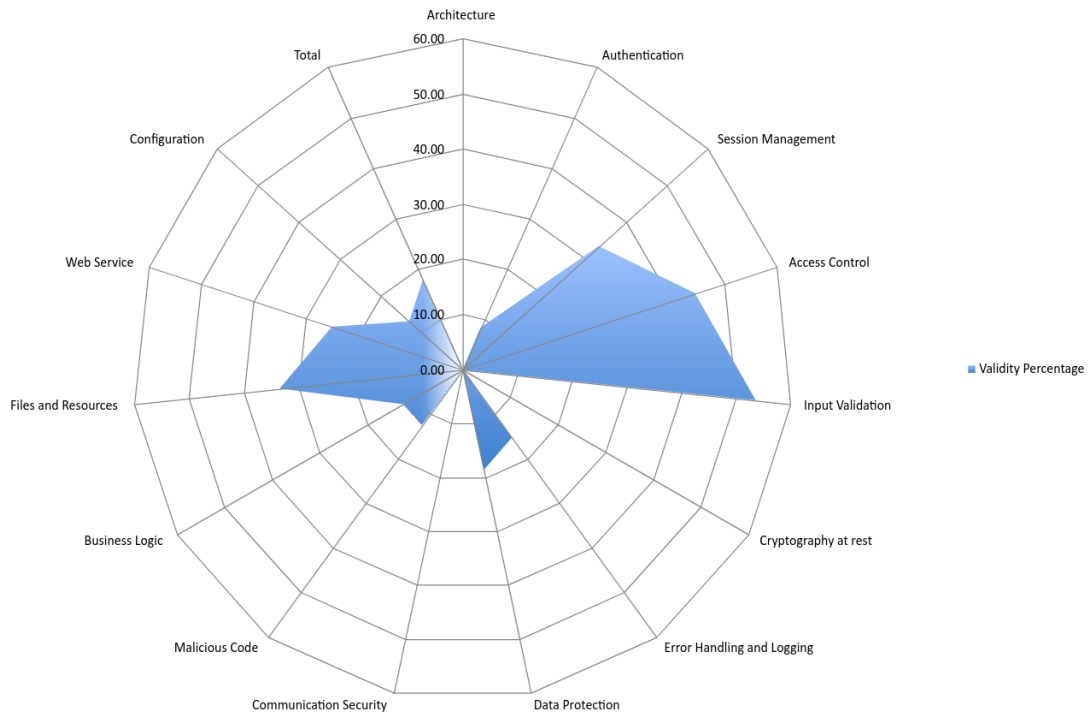


Image 3 - ASVS Validity Percentage

As we can see, only considering Level 1 ASVS, Configuration is a topic with low requirements checked, such as Files and Resources, Error handling and logging, Authentication, Configuration and Input validation. Thus, we will now enumerate for each topic, the points we think are important for the security of our web application.

- Error handling and logging

In this section, the issue that we choose is:

- 7.1.1 - Verify that the application does not log credentials or payment details. Session tokens should only be stored in logs in an irreversible, hashed form.

We selected this issue because it's crucial for security and privacy protection, as displaying or storing sensitive information like credentials and payment details greatly heightens the risk of identity theft and potential financial loss for our users. Additionally, this issue holds immense importance in ensuring compliance with legal requirements and regulations. By complying with data protection laws such as GDPR or industry-specific standards not only safeguards user information but also shields our web app from severe penalties, lawsuits, and reputational damage that non-compliance might incur, if it was public.

- Authentication

Here, we resolved the following requirement:

- 2.5.4 - Verify shared or default accounts are not present (e.g. "root", "admin", or "sa").

Resolving the presence of shared or default accounts "admin," or "test_user", like we had in our system, is crucial for several reasons. Firstly, it helps improve cybersecurity by getting rid of weak points that hackers often exploit. This reduces the chances of unauthorized access and potential breaches. Secondly, unique accounts make it easier to track what each person is doing on a system, which is vital for security. Lastly, by securing sensitive information and keeping it private, this action protects personal, financial, and important business data from being accessed by the wrong people.

Now, by default, the web application does not have any type of default accounts. We had to remove these accounts in the mysql_query.sql file to not add automatically these accounts when doing a docker compose. In the image below, we can see the initial script of the initial state of our application.

```
-- START TRANSACTION;
-- USE `shopping-cart`;
-- -- Insira um registro para o usuário "guest@gmail.com"
-- INSERT INTO `shopping-cart`.`user` (`email`, `name`, `mobile`, `address`, `pincode`, `password_hash`, `salt`)
-- VALUES ('guest@gmail.com', 'Guest User', 9876543234, 'K.P Road, Gaya, Bihar - India', 879767, 'hashed_password_for_guest', 'salt_for_guest');

-- -- Insira um registro para o usuário "admin@gmail.com"
-- INSERT INTO `shopping-cart`.`user` (`email`, `name`, `mobile`, `address`, `pincode`, `password_hash`, `salt`)
-- VALUES ('admin@gmail.com', 'Admin User', 9876543459, 'ABC Colony, Newtown, West Bengal', 786890, 'hashed_password_for_admin', 'salt_for_admin');
-- COMMIT;
```

Image 4 - Initial script

- Input Validation

In this section, the issues that we choose were:

- 5.2.1 - Verify that all untrusted HTML input from WYSIWYG editors or similar is properly sanitized with an HTML sanitizer library or framework feature.
- 5.2.2 - Verify that unstructured data is sanitized to enforce safety measures such as allowed characters and length.

The 5.2.1 requirement is very important once it acts as a security fence against injection attacks (HTML input can contain malicious scripts or code that can lead to XSS attacks), mitigates the risk of malware injections, maintains its data integrity and prevents other browser-side vulnerabilities. Although it was previously implemented in some features, with the erase of some features, a vulnerability was shown, in the footer section, where a user could submit messages to our server, which was a serious POST Reflected XSS (Cross site scripting attack), that was addressed and corrected throughout a data sanitation process using OWASP Policy Factor and OWASP HTML Policy Builder.

The 5.2.2 requirement is crucial mainly because of injection attacks like SQL Injection, which continue to be on the top 10 of Web Application Security Risks - OWASP. By this, this issue, if verified, can prevent corruption of the database system, ensuring that the data is consistent, accurate and usable. This also makes the application to not have system crashes or malfunctions, once excessively long inputs can lead to memory-related issues (also leading to more expenses). As said previously, the use of OWASP Policy Factor and OWASP HTML Policy Builder were essential in the data sanitization that then led to the resolution of 3 Cross Site Scripting Reflected Attacks (one for each field). It also provides a general message for the user if the data proves to have some type of script attack

```
private String sanitizeInput(String input) {  
    PolicyFactory sanitizer = new HtmlPolicyBuilder().toFactory();  
    String safeInpString = sanitizer.sanitize(input);  
    return safeInpString;  
}  
  
private boolean isValidEmail(String email) {  
    return email.contains("@");  
}
```

```
// sanitizar dados  
name = sanitizeInput(name);  
email = sanitizeInput(email);  
comments = sanitizeInput(comments);
```

Image 5- Code for sanitizing and validating data from Contact from fans.

```
▼ Cross Site Scripting (Reflected) (5)  
  GET: http://localhost:8080/shopping-cart/index.jsp?type=%3Cimg+src%3D%  
  GET: http://localhost:8080/shopping-cart/login.jsp?message=%3C%2Fp%3E  
  POST: http://localhost:8080/shopping-cart/fansMessage  
  POST: http://localhost:8080/shopping-cart/fansMessage  
  POST: http://localhost:8080/shopping-cart/fansMessage
```

Image 6 - Cross Site Scripting Reflected POST Methods from fansMessage (contact)

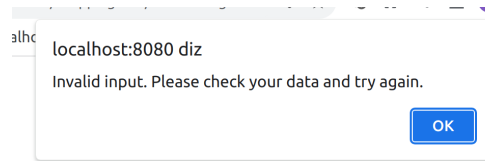


Image 7 - Result from Cross Site Scripting Reflected Attempt.

- Files and Resources

Here, we choose:

- 12.1.1 - Verify that the application will not accept large files that could fill up storage or cause a denial of service.
- 12.3.4 - Verify that the application protects against Reflective File Download (RFD) by validating or ignoring user-submitted filenames in a JSON, JSONP, or URL parameter, the response Content-Type header should be set to text/plain, and the Content-Disposition header should have a fixed filename.

One evident problem that this issue 12.1.1 could bring was related to resource management. Allowing large files could consume a large amount of storage, leading to resource constraints. The performance of the store could increase dramatically and this would lead to a worse system efficiency. Another huge problem was the high risk of DoS attacks - which is meant to shut down a machine or network, making it inaccessible to its intended users. This attack causes the system to be overwhelmed by large files and causes a denial of service.

Before this the system could be overloaded if an attacker wanted to, by reaching the maximum capacity of the Buffer.

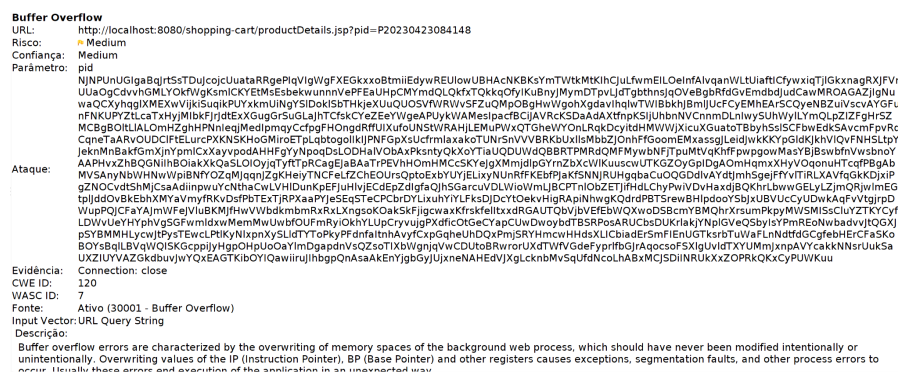


Image 8 - Buffer OverFlow attack, captured by OWASP ZAP

Because of this, we decided to add a header to our nginx configuration file. It defines the range of the files that could be sent, not allowing a pid that could overflow.

```
add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range';  
add_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';
```

Image 9 - Nginx header

RFD vulnerabilities enable attackers to manipulate user interactions, leading to the download of malicious files. By validating user-submitted filenames and setting specific response headers like Content-Type and Content-Disposition, the application prevents malicious file execution, reducing the risk of compromising sensitive data or unauthorized system access. Implementing this vulnerability also protects users from downloading inadvertently harmful content and thus the reputation and trust of our web app is maintained.

Before this was implemented, our website allowed an attacker to manipulate request parameters, such as a filename, to make the application return a malicious file to the user. This could result in undesired script execution or delivery of harmful content to the user, exploiting the user's trust in the application.

Now because of the “default_type text/plain” the website interprets the response as plain text, preventing the execution of malicious scripts that could be interpreted if the Content-Type was configured differently. By configuring the fixed filename, it helps to guarantee that the archive could be downloaded with a malicious manipulation made by the attacker.

```
if ($arg_download = "true") {  
    add_header Content-Disposition "attachment; filename=report.pdf";  
}  
default_type text/plain;
```

Image 10 - “Header” and “Default_type” example

404 Not Found

nginx

Image 11 - Response after trial of RFD attack

- Configuration

For this tab, we select the following issues:

- 14.2.1 - Verify that all components are up to date, preferably using a dependency checker during build or compile time.
- 14.2.2 - Verify that all unneeded features, documentation, samples, configurations are removed, such as sample applications, platform documentation, and default or example users.

- 14.4.3 - Verify that a Content Security Policy (CSP) response header is in place that helps mitigate impact for XSS attacks like HTML, DOM, JSON, and JavaScript injection vulnerabilities.
- 14.4.4 - Verify that all responses contain a X-Content-Type-Options: nosniff header.
- 14.4.7 - Verify that the content of a web application cannot be embedded in a third-party site by default and that embedding of the exact resources is only allowed where necessary by using suitable Content-Security-Policy: frame-ancestors and X-Frame-Options response headers.

Following the description of the issue 14.2.1, we think that it is very important that all components in the system are up to date. That are a number of issues that later can occur because of this issue not being implemented. Security, stability, reliability, compatibility, performance, compliance, community support and documentation, cost-efficiency and future development are only a few examples of the benefits of implementing this issue. The requirement 14.2.2 follows the same logic as the 14.2.1, simplifying our code and only using what we need to avoid confusion and future development and performance.

To validate issue 14.2.1, we utilized the Maven Version tool, which scans all project dependencies and checks for available updates. So, the following updates were recommended:



```
commons-codec:commons-codec - 1.15 -> 1.16.0
Jakarta.mail:jakarta.mail-api - 2.1.1 -> 2.1.2
javax.servlet:javax.servlet-api - 3.1.0 -> 4.0.1
org.apache.commons:commons-lang3 - 3.10 -> 3.14.0
```

Image 12 - recommended updates

So, considering that new versions of dependencies may address security vulnerabilities present in previous versions, we updated everything necessary in the POM.xml file.



```
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>javax.servlet-api</artifactId>
  <version>4.0.1</version>
</dependency>

<dependency>
  <groupId>jakarta.mail</groupId>
  <artifactId>jakarta.mail-api</artifactId>
  <version>2.1.2</version>
</dependency>

<dependency>
  <groupId>commons-codec</groupId>
  <artifactId>commons-codec</artifactId>
  <version>1.16.0</version>
</dependency>

<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.14.0</version>
</dependency>
```

Image 13 - Dependencies after updates

To validate issue 14.2.2, our focus was on refactoring the code to eliminate all functionalities that were implemented but not actually in use. The primary removal involved the email-sending functionality, which was scattered across various parts of the project. Consequently, the classes 'MailMessage' and 'TestMail' were removed, along with all the code responsible for their instantiation.

We've determined that confirming the presence of a Content Security Policy (CSP) response header - the level 1 requirement 14.4.3 - is effective in reducing the risks associated with XSS attacks, such as HTML, DOM, JSON, and JavaScript injection vulnerabilities. It can also protect sensitive data if implemented, because XSS attacks can lead to the leak of users' sensitive information.

```

# nginx.conf
1 server {
2     listen 80;
3     server_name localhost;
4     server_tokens off;
5
6     location /shopping-cart/ {
7         add_header Content-Security-Policy "default-src 'self'; style-src 'self' https://maxcdn.bootstrapcdn.com https://cdnjs.cloudflare.com 'unsafe-:
8         add_header X-Frame-Options "SAMEORIGIN" always;
9         add_header X-Content-Type-Options "nosniff" always;
10        add_header X-XSS-Protection "1; mode=block" always;
11        add_header Set-Cookie "name=value; Secure; HttpOnly; SameSite=Strict" always;
12        add_header 'Access-Control-Allow-Origin' '/shopping-cart/';
13        add_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';
14        add_header 'Access-Control-Allow-Headers' 'DNT,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range';
15        add_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';
16
17        proxy_pass http://tomcat:8080;
18        proxy_http_version 1.1;
19
20        default_type text/plain;
21    }

```

Image 14 - Code from a CSP Implementation using Nginx (from file nginx.config)

As a matter of fact before we had a Content Security Policy there were XSS Reflected Attacks, which were nefarious to our website, causing data to be explored by attackers. By implementing a CSP Header we guaranteed more protection to our website.

Cross Site Scripting (Reflected) (4)
 GET: http://localhost/shopping-cart/index.jsp?type=%3Cimg+src%3Dx+
 GET: http://localhost/shopping-cart/login.jsp?message=%3C%2Fp%3E%
 GET: http://localhost:8080/shopping-cart/index.jsp?type=%3Cimg+src%
 GET: http://localhost:8080/shopping-cart/login.jsp?message=%3C%2Fp%

Image 15 - Cross Site Scripting Attacks (OWASP ZAP)



Image 16 - Alert Caused from XSS Reflected Attack done through address(http://localhost:8080/shopping-cart/login.jsp?message=%3C%2Fp%3E%3Cscript%3Ealert%281%29%3B%3C%2Fscript%3E%3Cp%3E)

By implementing a CSP Protocol our website does not execute this nefarious code and shows an error in the console as well, showing that it does not conform to CSP (Content Security Policies).

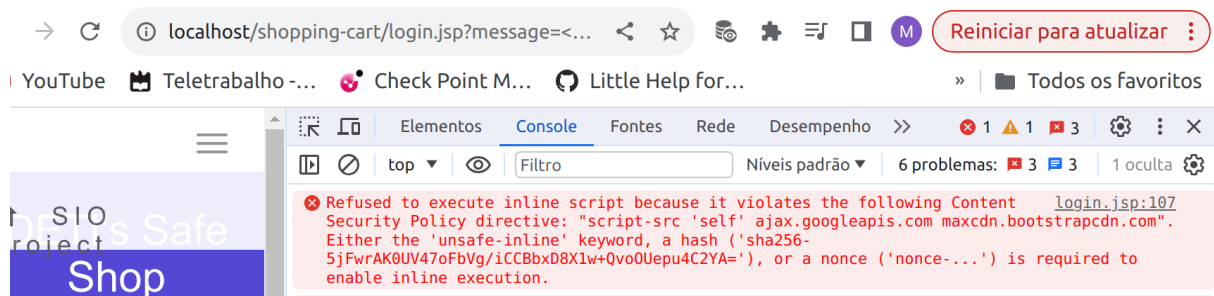


Image 17 - Error is shown in console and nothing happens when an attacker tries to do XSS Reflected.

Another key point for the security of our application was the issue 14.4.4, which implements the “nosniff” header, that instructs the browser to not perform MIME type sniffing and also mitigating XSS attacks.

Before this was implemented it allowed older versions of Internet Explorer and Chrome to perform MIME-sniffing on the response body, potentially causing the response body to be interpreted and displayed as a content type other than the declared content type.

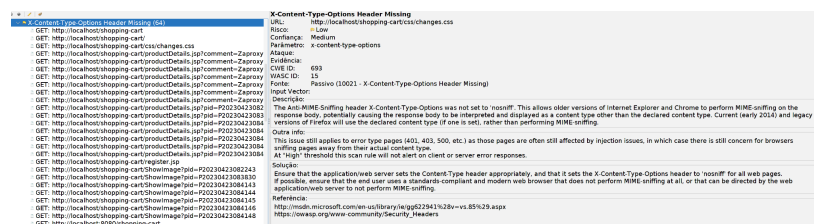


Image 18 - OWASP Analysis.

With the addition of this header to our nginx.conf file, our server file, we prevented this from happening.

```
add_header X-Content-Type-Options "nosniff" always;
```

Image 19 - nginx conf file Header X-Content-Type

- Cryptography at rest

The issues from this tab were mainly implemented due to the database encryption

- 6.1 - All this group is referent to Data Classification where we analyze the storage data and choose which one should be encrypted or not. We decided to only encrypt user related data.

email	salt	name	mobile	address	pincode	password_hash
joaocls123@hotmail.com	joaocls123@hotmail.com	tkJl+6Z0NjgIjzhwscB+Uw==	D/QUvk/kwb45ju4n7F3PKF58DFOX91LUwp+YtcbaXlQ=	73LkKo/h84XVeI4ep4jF9Q==	\$2a\$10\$Tz8mz6s97/jKCeAwP2uur.Ps6Tv4uyVVVB17jRASbBPhHrRQd119.	\$2a\$10\$Tz8mz6s97/jKCeAwP2uur.Ps6Tv4uyVVVB17jRASbBPhHrRQd119.

Image 20 - encrypted user data

We think that analyzing all the storage data is very important to guarantee that important information such as personal info should be protected even if someone can access our database. That way we can offer our users safety and protection.

- 6.2.2 - We chose an industry proven algorithm and not a random one so we know for sure that the encryption method is safe.

```
private static SecretKey generateAesKey() {
    try {
        KeyGenerator keyGenerator = KeyGenerator.getInstance("AES");
        keyGenerator.init(128);
        System.out.println("Using AES KeyGenerator");
        return keyGenerator.generateKey();
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

Image 21 - encryption algorithm example

- Data protection

For this topic:

- 8.3.2 - We set a delete account method, so the user can have control over is information, and if he doesn't want us to have is information any time he can delete is account and revoke all information given

- 8.3.3 / 8.3.4 - In this topic we ask a user to consent about the data collecting and what we do with their data so that they can be comfortable about what we do.

We prioritized these issues as we consider them crucial for ensuring user comfort and trust in our web service. By implementing a delete account feature (8.3.2), users gain control over their information, allowing them to revoke it at any time. Additionally, obtaining user consent for data collection and explaining how their data is handled (8.3.3 / 8.3.4) is essential to foster a sense of security and transparency in our service

Mandatory Implementations

- Password strength evaluation

For the mandatory implementations, we choose to implement the password strength evaluation: requiring a minimum of strength for passwords according to V2.1. This section comprises 12 key points, and we'll address each one.

- 2.1.1 - Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined).
- 2.1.2 - Verify that passwords 64 characters or longer are permitted but may be no longer than 128 characters.

These 2 issues were previously not totally in our secure version of the app. The input of the password was guided by a regular expression pattern in register.jsp frontend file (pattern = "(?=.*\d)(?=.*[a-z])(?=.*[A-Z]).{8,}") where, as we can see, it limits the password size to 8 characters or bigger. For doing this issue, we removed this pattern and instead changed the RegisterSrv.java class:

```
if (possibleSpacesPassword.length() >= 12 && possibleSpacesPassword.length() <= 128
```

Image 22 - new password check

- 2.1.3 - Verify that password truncation is not performed. However, consecutive multiple spaces may be replaced by a single space.

This issue was also not valid because if the user imputed in the password spaces, the password remained intact. Now, if a user inputs more than one space, the app truncates the spaces, as we can see in the image below:

```

String possibleSpacesPassword = request.getParameter("password");
String possibleSpacesConfirmPassword = request.getParameter("confirmPassword");
String consentCheckbox = request.getParameter("consentCheckbox");
String status = "";
String password = possibleSpacesPassword.trim().replaceAll(" ", " ");
String confirmPassword = possibleSpacesConfirmPassword.trim().replaceAll(" ", " ");

BreachedPasswords breachedPasswords = new BreachedPasswords();

if (password != null && password.equals(confirmPassword)) {

    if (possibleSpacesPassword.length() >= 12 && possibleSpacesPassword.length() <= 128
        && !possibleSpacesPassword.equals(possibleSpacesPassword.toLowerCase())
        && possibleSpacesPassword.matches(".*\\d+.*")) {

```

Image 23 - Truncates spaces in password

- 2.1.4 - Verify that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords.

As there's not anymore a password pattern that has to be followed, we can now include any type of characters in the passwords. Previously, we could not.

- 2.1.5 - Verify users can change their password.
- 2.1.6 - Verify that password change functionality requires the user's current and new password.

Users also can now change their passwords. In the last version of the web app, the button was there, but the action was not implemented. Now it is:

```

UserServiceImpI udao = new UserServiceImpI();

status = udao.isValidCredential(userName, oldPassword);

if (status.equalsIgnoreCase("valid")) {
    // valid user

    if (breachedPasswords.isPasswordBreached(newPassword)) {

        if (newPassword.length() < 12) {
            status = "password too small!";
        } else {

            UserBean user = udao.getUserDetails(userName, oldPassword);

            String salt = BCrypt.gensalt();
            String hashedPassword = BCrypt.hashpw(newPassword, salt);

            System.out.println("Salt: " + salt);
            System.out.println("Password: " + newPassword);
            System.out.println("Hashed Password: " + hashedPassword);

            UserBean user2 = new UserBean(user.getName(), user.getMobile(), user.getEmail(), user.getAddress(),
                user.getPinCode(), hashedPassword, salt);

            UserServiceImpI dao = new UserServiceImpI();

            udao.removeUser(user);

            status = dao.registerUser(user2);

            HttpSession session = request.getSession();

            session.setAttribute("username", userName);
            session.setAttribute("password", newPassword);
            session.setAttribute("usertype", "customer");
        }
    }
}

```


Image 24 - password change functionality

The user needs to input its old password for the process to continue:

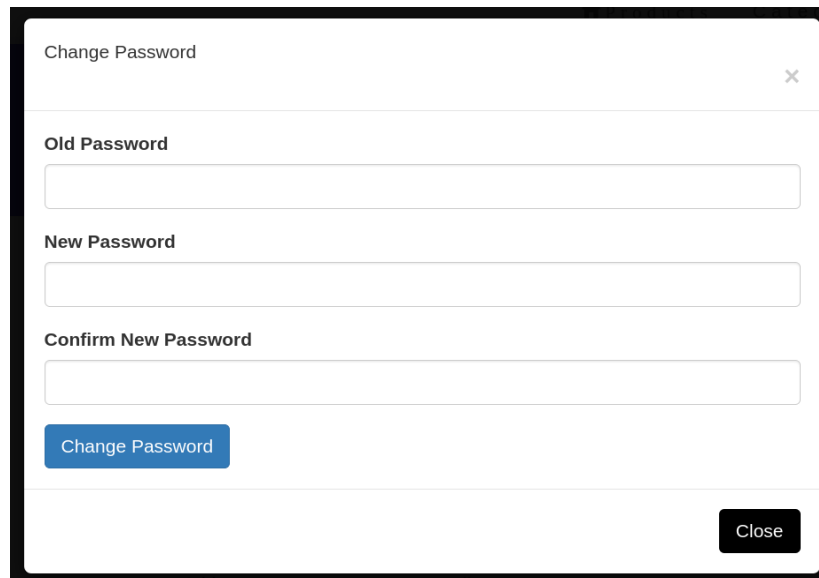


Image 25 - Change password form

- 2.1.7 - Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords.

What was once an impossibility, now can be shown in the image below:

You, 11 hours ago | 1 author (You)

```
public class BreachedPasswords {
    private List<String> commonBreachedPasswords;

    public BreachedPasswords() {
        commonBreachedPasswords = new ArrayList<>();
        readBreachedPasswordsFromFile(filename:"./src/com/shashi/utility/file.txt");
    }

    private void readBreachedPasswordsFromFile(String filename) {
        try (BufferedReader br = new BufferedReader(new FileReader(filename))) {
            String line;
            while ((line = br.readLine()) != null) {
                // Assuming each password is on a separate line in the file
                commonBreachedPasswords.add(line.trim());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println(commonBreachedPasswords);
    }

    public boolean isPasswordBreached(String password) {
        return commonBreachedPasswords.contains(password);
    }
}
```

Image 26 - BreachedPasswords class

Here we can see that it checks if an argument is in the 100 most used passwords in the world, a list found in this repository:

[\[https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000.txt\]](https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000.txt)

Now, we can create an object of the type BreachedPasswords and see if the password imputed is in the list:

```
BreachedPasswords breachedPasswords = new BreachedPasswords();

if (password != null && password.equals(confirmPassword)) {

    if (possibleSpacesPassword.length() >= 12 && possibleSpacesPassword.length() <= 128
        && !possibleSpacesPassword.equals(possibleSpacesPassword.toLowerCase())
        && possibleSpacesPassword.matches(".*\\d+.*")) {

        if (consentCheckbox == null || !consentCheckbox.equals("on")) {
            response.sendRedirect("register.jsp?message=Você deve concordar com os Termos e Condições.");
            return;
        }

        if (password.length() >= 12) {

            if (breachedPasswords.isPasswordBreached(confirmPassword)) {
                status = "Your password is part of the 1000 more used. Please choose a different password.";
            } else {
```

Image 27 - BreachedPasswords object creation and verification

- 2.1.8 - Verify that a password strength meter is provided to help users set a stronger password.

A password strength meter was also implemented using the “zxcvbn” library that gives a score to the password introduced using color code and words.

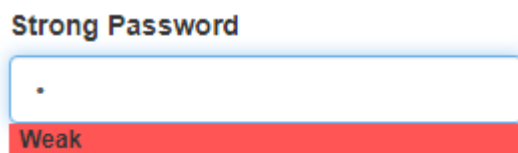


Image 28 - Weak password indication

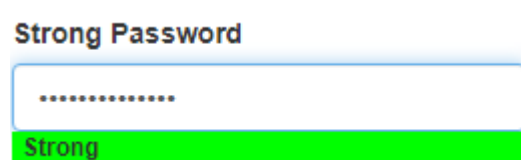


Image 29 - Strong password indication

- 2.1.9 - Verify that there are no password composition rules limiting the type of characters permitted.
- 2.1.10 - Verify that there are no periodic credential rotation or password history requirements.

All passwords require no composition rules limiting the type of characters permitted and no periodic credential rotation or password history requirements.

- 2.1.11 - Verify that "paste" functionality, browser password helpers, and external password managers are permitted.

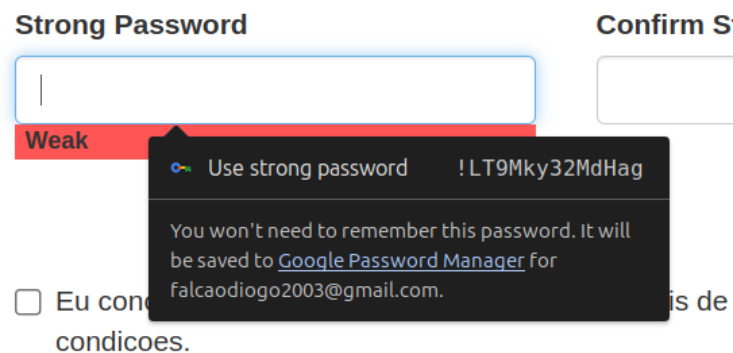


Image 30 - Password helper example

- 2.1.12 - Verify that the user can choose to either temporarily view the entire masked password, or temporarily view the last typed character of the password on platforms that do not have this as built-in functionality.

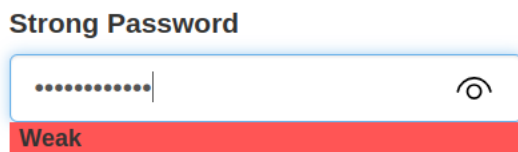


Image 31 - masked password

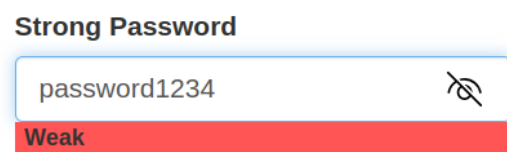


Image 32 - password with temporary view

- Encrypted database storage

For the second mandatory software feature, we chose encrypted database storage - to encrypt our database we started to select which content was important to encrypt and which wasn't and we concluded that all information of users must be encrypted, only the username and email are not encrypted.

For that matter we started by choosing which encryption method we were going to use and we decided to use the Advanced Encryption Standard (AES). We create a KeyGenerator that is responsible for generating the key and the methods to encrypt/decrypt the data. These methods are used by our other entities when they have some method that access database data. For the password we used BCrypt, a library that employs adaptive hashing, salting, and configurable iterations, making it resistant to brute force and rainbow table attacks

email	salt	name	mobile	address	pincode	password_hash
joac1s123@hotmail.com	joac1s123@hotmail.com	tkJ1l+6Z0NjgIjzHwscB+Uw==	D/QUvk/kwlb45ju4n7FJPKF58DF0X91LUmp+YtcbaxIQ=	73LkKo/h84XVeI4ep4jF9Q==	\$2a\$10\$Tz8mz6s97/jKCeAwP2uur.	P56Tv4uyVVVB17jRASb

Image 33 - encrypted information

We also acknowledge our users about the use of the collected data and provide a consent box.

Registration Form

Name

Email

Address

Mobile

Pin Code

Strong Password

Confirm Strong Password

Strength

Reset

Register

☐ Eu concordo com o uso dos meus dados pessoais de acordo com os termos e condicoes.

[Termos e Condicoes](#)

Termos e Condicoes

Terms and Conditions 1. Introduction Welcome to our online deti shop! These terms and conditions govern your use of our website and services offered. By accessing or using our website, you agree to comply with these terms and conditions. Please read them carefully.

2. Access and Account Creation 2.1. Our website allows the purchase of products either by adding them to the cart or by immediate purchase. 2.2. The cart functionality permits modification of product quantities (adding or removing products). 2.3. Account creation is available with the option to log in as an administrator or customer based on the account type. 3. Product Comments and Search 3.1. Users can post comments on each product page, including images, text, or both. 3.2. The website facilitates product search using the search bar and filters available in the Category section. 4. Technologies Used 4.1. The website is built using the following technologies: Front-End: HTML, CSS, JavaScript, and Bootstrap. Back-End: Java (JDK8+), JDBC (Java Database Connectivity), Servlet, and JSP (Jakarta Server Pages). Database: MySQL. Additional Tools: TomCat, Maven, docker-compose for efficient page loading. 5. Disclaimer 5.1. The vulnerabilities used for the development of this website were sourced from a repository authored by Shashi Rash. 5.2. While we strive to provide a secure platform, we do not guarantee absolute security. Users are responsible for their actions and data security. 6. Acceptable Use 6.1. Users must not misuse the website, violate laws, or infringe on others' rights. 6.2. Users must not attempt unauthorized access, interfere with the website's functionality, or distribute harmful content. 7. Limitation of Liability 7.1. We are not liable for any direct, indirect, or incidental damages resulting from the use of our website, including but not limited to loss of data, profits, or business interruption. 8. Modifications to Terms 8.1. We reserve the right to modify these terms and conditions at any time without prior notice. Continued use of the website constitutes acceptance of the modified terms. 9. Governing Law 9.1. These terms and conditions are governed by [Jurisdiction]'s laws, and any disputes shall be resolved in accordance with these laws. 10. Contact Information 10.1. For any queries or concerns regarding these terms, please contact us at [Contact Information]. By using our website, you agree to abide by these terms and conditions. If you do not agree with any part of these terms, please refrain from using our website. Thank you for choosing our deti shop!

Fechar

Image 34 - Registration form with terms and conditions check

Image 35 - details of terms and conditions

Conclusion

In conclusion, the website trial revealed numerous vulnerabilities within seemingly trivial features. Examining the provided Excel data showed the interconnected nature of security provisions, emphasizing the impact of minor adjustments on a website's overall security against potential malicious attacks.

The assessment with OWASP provided valuable insights into previously overlooked vulnerabilities, showcasing the practical application of theoretical and practical knowledge gained in previous classes.

This project has underscored the significance of integrating security measures and best practices into our future works, recognizing the critical role of security in an effective website management.

References

OWASP. (s.d.). OWASP Top Ten. OWASP.

Obtido de <https://owasp.org/www-project-top-ten/>

Content Security Policy. (s.d.). Hash.

Obtido de <https://content-security-policy.com/hash/>

Miessler, D. (n.d.). 10-million-password-list-top-1000.txt. GitHub.

Recuperado de

<https://github.com/danielmiessler/SecLists/blob/master/Passwords/Common-Credentials/10-million-password-list-top-1000.txt>