universidade de aveiro
departamento de eletrónica, telecomunicações e informática

Second Project

Application Security Verification Standard

Segurança Informática e nas Organizações

Report

Alexandre Ribeiro Nº 108122

João Luís Nº 107403

Jose Gameiro Nº 108840

Vítor Santos Nº 107186

Index

1. Introduction

In the first practical project for the “Segurança Informática e nas Organizações” (SIO) class, our group developed an e-commerce shop to sell DETI-themed memorabilia using the *Python* web microframework *Flask* [10] and the DBMS *MySQL* [9], hosted in *Linux* containers [11].

As a follow-up to this first project, the second aims to explore the Application Security Verification Standard (ASVS) by applying its concepts to the previously developed web application. In addition to exploring and applying the ASVS, our group also explored and implemented two additional security-related features.

2. Application Security Verification Standard

2.1. Overview

The OWASP's **Application Security Verification Standard** (ASVS) [6] is a set of security requirements to take into consideration when developing web software applications, serving as a base for testing applicational and environmental security controls, as well as a guideline for secure development to be used by software engineers, architects and developers.

Requirements are grouped into three numerically identified security verification levels, with each being a superset of the previous level's requirements:

- ASVS L1 – completely penetration testable, composed of the bare minimum requirements that **all** applications should strive for.
- ASVS L2 – for applications containing sensitive data and conducting business-to-business transactions. Appropriate for most applications.
- ASVS L3 – for critical applications that perform high value transactions, such as the ones present in the areas of the military, health and safety, critical infrastructure, etc...

For the academic purposes of this project, our group conducted an audit of the previously developed e-commerce application, using a checklist based on the ASVS' L1 requirements.

If our application were to implement monetary transactions for product orders, sensitive credit card data and other payment information would need to be stored and handled, requiring the level of assurance covered by ASVS L2 instead of L1.

After identifying the points where our application fell short, the nine most critical issues were identified, with each covering one or more ASVS L1 requirements. All these issues were subsequently corrected.

2.2. Audit

Using the provided OWASP ASVS checklist for audits [1], we aggregated all checklist items pertaining to ASVS L1 requirements and verified each.

Audit results can be explored in detail at the file ***audit.xlsx*** in the ***reports*** directory.

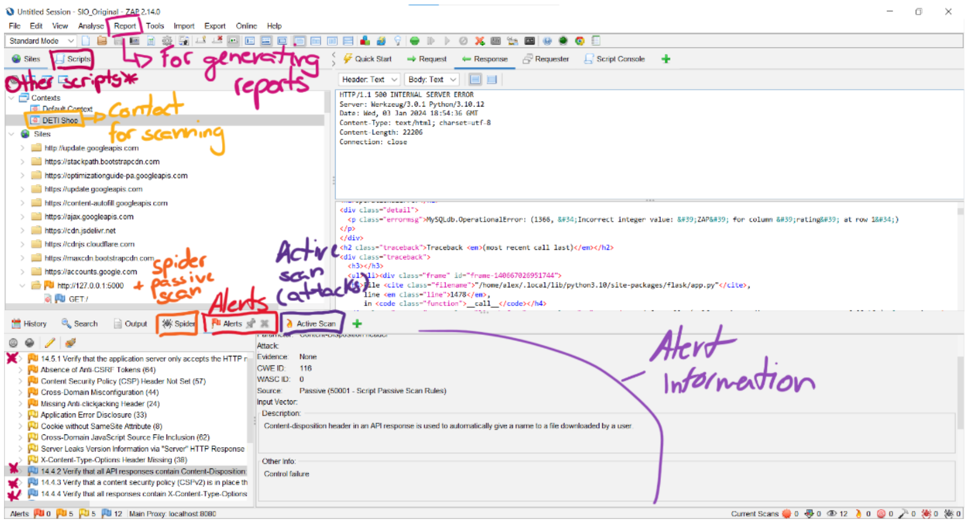
2.3. Audit tools

While researching how to better perform the audit, we came across a GitHub project [2] that provided tools and insight on how to test most L1 requirements.

Some of the tests used *Chrome Dev Tools* [4] or simply *curl* [5], but most of them used a tool called OWASP ZAP [3], which is a widely used web app scanner, capable of identifying a variety of weaknesses, most of them related to CWEs.

We utilized the testing guide and additional scripts from the GitHub project to conduct both passive and active scans. Our objective was to identify any alerts raised by ZAP pertaining to the requirements under examination. Subsequently, we assessed whether these alerts were false positives and documented our findings in the corresponding rows of the audit checklist.

We also analyzed the other alerts, unrelated to the specific ASVS requirements, to improve the security of our web app.


Untitled Session - SIO_Original - ZAP 2.14.0
File Edit View Analyse Report Tools Import Export Online Help
Standard Mode
Sites Scripts
↳ for generating reports
Other scripts
Contexts
Default Context
DETI Shop
→ Context for scanning
Sites
http://update.googleapis.com
https://stackpath.bootstrapcdn.com
https://optimizationguide-pa.googleapis.com
https://update.googleapis.com
https://content-autofill.googleapis.com
https://ajax.googleapis.com
https://cdn.jsdelivr.net
https://cdnjs.cloudflare.com
https://maxcdn.bootstrapcdn.com
https://accounts.google.com
http://127.0.0.1:5000
GET:/
spider + passive scan
Active scan (attacks)
Alerts
Quick Start Request Response Requester Script Console
Header: Text Body: Text
History Search Output Spider Alerts Active Scan
Alert Information
Alerts 0 5 5 12 Main Proxy: localhost:8080
Current Scans 0 0 12 0 0 0 0 0

```
HTTP/1.1 500 INTERNAL SERVER ERROR
Server: Werkzeug/3.0.1 Python/3.10.12
Date: Wed, 03 Jan 2024 18:54:36 GMT
Content-Type: text/html; charset=utf-8
Content-Length: 22206
Connection: close
```

```
<div class="detail">
  <p class="errormsg">MySQLdb.OperationalError: (1366, &#34;Incorrect integer value: &#39;ZAP&#39; for column &#39;rating&#39; at row 1&#34;)
</p>
</div>
<h2 class="traceback">Traceback <em>(most recent call last)</em></h2>
<div class="traceback">
  <h3></h3>
  <ul><li><div class="frame" id="frame-140667026951744">
  <h4>File <cite class="filename">"/home/alex/.local/lib/python3.10/site-packages/flask/app.py"</cite>,
      line <em class="line">1478</em>,
      in <code class="function">__call__</code></h4>
```

- 14.5.1 Verify that the application server only accepts the HTTP r
- Absence of Anti-CSRF Tokens (64)
- Content Security Policy (CSP) Header Not Set (57)
- Cross-Domain Misconfiguration (44)
- Missing Anti-clickjacking Header (24)
- Application Error Disclosure (33)
- Cookie without SameSite Attribute (8)
- Cross-Domain JavaScript Source File Inclusion (62)
- Server Leaks Version Information via "Server" HTTP Response
- X-Content-Type-Options Header Missing (38)
- 14.4.2 Verify that all API responses contain Content-Disposition
- 14.4.3 Verify that a content security policy (CSPv2) is in place th
- 14.4.4 Verify that all responses contain X-Content-Type-Options

Attack:
Evidence: None
CWE ID: 116
WASC ID: 0
Source: Passive (50001 - Script Passive Scan Rules)
Input Vector:
Description:
Content-disposition header in an API response is used to automatically give a name to a file downloaded by a user
Other Info:
Control failure

Fig.2.3 – ZAP layout.

3. Critical Issues Found

The following section will provide an exposition of each of the **nine** most severe issues identified using the previous audit, with a description of the issue, associated ASVS requirements, justification for the issue's severity, source code references involved in causing the issue, implemented solution and source code references for the implemented solution.

3.1. Weak Anti-CSRF Attacks Policies

Area of the ASVS: Operation Level Access Control.

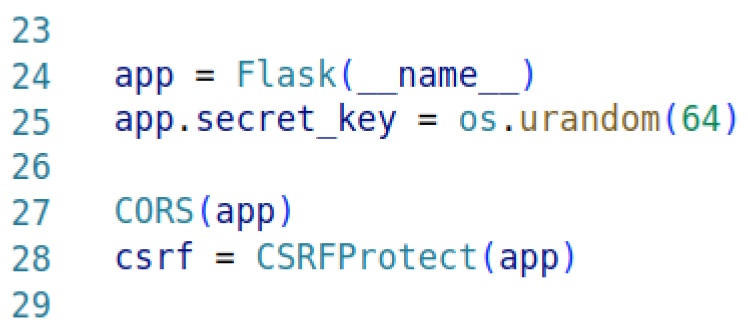
Associated ASVS requirements: 4.2.2.

Description: Verify that the application or framework enforces a strong anti-CSRF mechanism to protect authenticated functionality, and effective anti-automation or anti-CSRF protects unauthenticated functionality.

Impact: Without a robust anti-CSRF mechanism, attackers may exploit the absence of protection to trick authenticated users into unknowingly performing actions that they did not intend. This could result in unauthorized transactions, data manipulation, or other actions that compromise the confidentiality, integrity, and availability of the application's resources.

Solution: To address the CSRF vulnerability and fulfill the ASVS requirement, we implemented a solution by using the CSRF protection provided by the library flask_wtf.csrf.

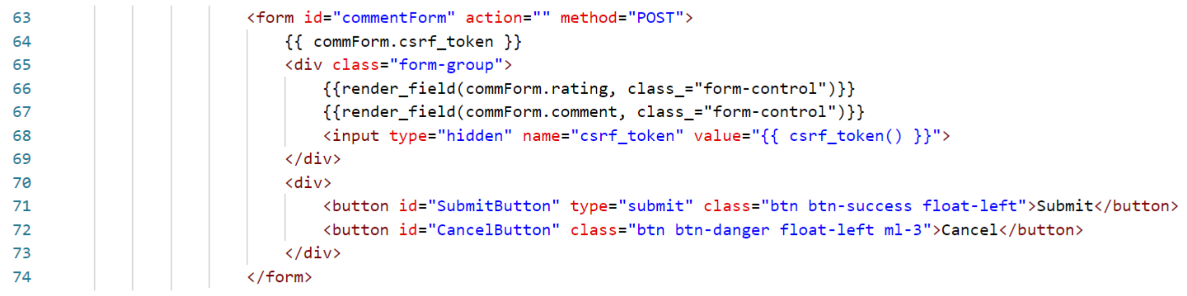
We used the CSRFProtect extension to generate and validate unique CSRF tokens for each user session. By incorporating this extension, we fortified our application against Cross-Site Request Forgery attacks.



```

app = Flask(__name__)
app.secret_key = os.urandom(64)

CORS(app)
csrf = CSRFProtect(app)

```

Fig.3.1a) *- Code developed to improve anti CSRF policies.*

In every form where user input is required, we also made certain to include a hidden input element containing the CSRF token. This token acts as a secure and unique identifier, ensuring that each form submission is associated with the correct user session.



```
<form id="commentForm" action="" method="POST">
    {{ commForm.csrf_token }}
    <div class="form-group">
        {{render_field(commForm.rating, class_="form-control")}}
        {{render_field(commForm.comment, class_="form-control")}}
        <input type="hidden" name="csrf_token" value="{{ csrf_token() }}">
    </div>
    <div>
        <button id="SubmitButton" type="submit" class="btn btn-success float-left">Submit</button>
        <button id="CancelButton" class="btn btn-danger float-left ml-3">Cancel</button>
    </div>
</form>
```

***Fig.3.1b)** – Input field added to one of the forms inside our application.*

3.2. Non-existence of limits or validation of business logic actions

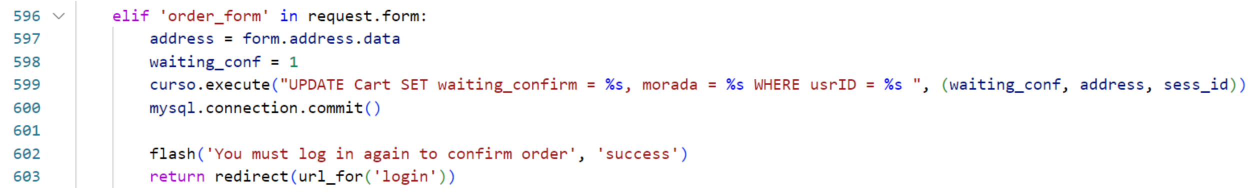
Area of the ASVS: Business Logic Security Requirements.

Associated ASVS requirements: 11.1.5.

Description: Verify the application has business logic limits or validation to protect against likely business risks or threats, identified using threat modeling or similar methodologies.

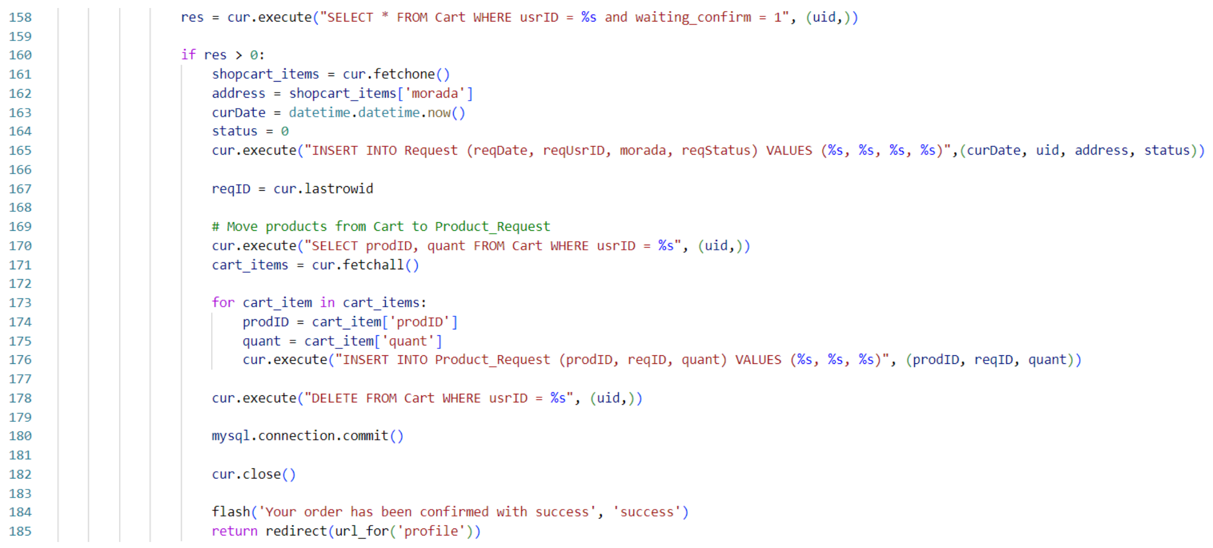
Impact: The application becomes susceptible to a range of treats that could compromise its integrity, availability, and confidentiality. For instance, attackers might exploit weaknesses in the business logic to manipulate transactions, bypass authorization protocols or execute unauthorized operations within the application.

Solution: We identified a security concern within specific functionality on our website, particularly concerning the confirmation process for orders within the shopping module. To enhance the security of this critical functionality, we implemented an additional authentication step. Users are now required to successfully log in again as part of the confirmation process, thereby introducing an extra layer of verification.



```
elif 'order_form' in request.form:
    address = form.address.data
    waiting_conf = 1
    curso.execute("UPDATE Cart SET waiting_confirm = %s, morada = %s WHERE usrID = %s ", (waiting_conf, address, sess_id))
    mysql.connection.commit()

    flash('You must log in again to confirm order', 'success')
    return redirect(url_for('login'))
```

***Fig.3.2a)** - Code developed in shopping cart endpoint for the user to be redirect to the log in page again.*



```
res = cur.execute("SELECT * FROM Cart WHERE usrID = %s and waiting_confirm = 1", (uid,))

if res > 0:
    shopcart_items = cur.fetchone()
    address = shopcart_items['morada']
    curDate = datetime.datetime.now()
    status = 0
    cur.execute("INSERT INTO Request (reqDate, reqUsrID, morada, reqStatus) VALUES (%s, %s, %s, %s)",(curDate, uid, address, status))

    reqID = cur.lastrowid

    # Move products from Cart to Product_Request
    cur.execute("SELECT prodID, quant FROM Cart WHERE usrID = %s", (uid,))
    cart_items = cur.fetchall()

    for cart_item in cart_items:
        prodID = cart_item['prodID']
        quant = cart_item['quant']
        cur.execute("INSERT INTO Product_Request (prodID, reqID, quant) VALUES (%s, %s, %s)", (prodID, reqID, quant))

    cur.execute("DELETE FROM Cart WHERE usrID = %s", (uid,))

    mysql.connection.commit()

    cur.close()

    flash('Your order has been confirmed with success', 'success')
    return redirect(url_for('profile'))
```

Fig.3.2b) *- Code developed in login endpoint to confirm the order if the log in is successful.*

3.3. Non-existence of data request limits

Area of the ASVS: Business Logic Security Requirements.

Associated ASVS requirements: 11.1.2, 11.1.3, 11.1.4.

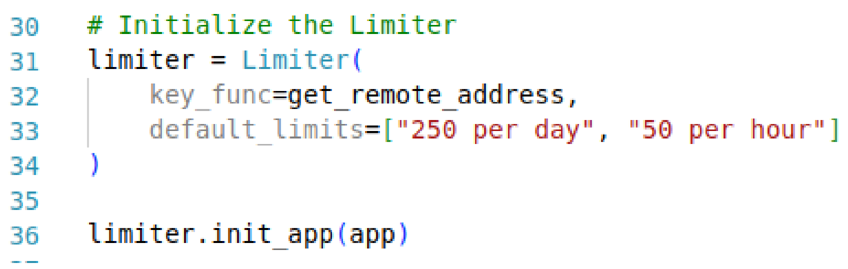
Description: Verify the application will only process business logic flows with all steps being processed in realistic human time, i.e. transactions are not submitted too quickly.

Verify the application has appropriate limits for specific business actions or transactions which are correctly enforced on a per user basis.

Verify the application has sufficient anti-automation controls to detect and protect against data exfiltration, excessive business logic requests, excessive file uploads or denial of service attacks.

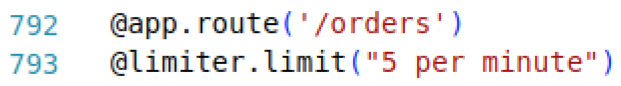
Impact: Without any restriction, any user could as many requests as wanted. This gap could be exploited by rogues to do a DoS attack against our application.

Solution: To address this issue, we limited the number of times than an endpoint can be called in a given period of time (per minute or per hour). We used a Limiter from a library called Flask_limiter, defining default limit values, and then, in specific endpoints, we applied an even lower limit of requests, with a number that we think for the normal user won't be noted, but for an evildoer, it would be restrictive.



```
# Initialize the Limiter
limiter = Limiter(
    key_func=get_remote_address,
    default_limits=["250 per day", "50 per hour"]
)

limiter.init_app(app)
```

Fig.3.3a) *Initialization of Limiter adding default values.*



```
@app.route('/orders')
@limiter.limit("5 per minute")
```

Fig.3.3b) *Limit for a specific endpoint.*

3.4. Missing Validation of Structured Data

Area of the ASVS: Input Validation.

Associated ASVS requirements: 5.1.4.

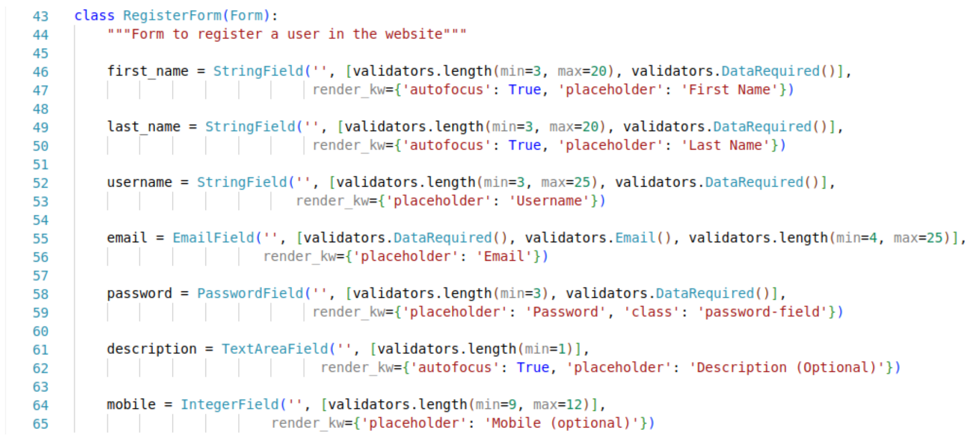
Description: Verify that structured data is strongly typed and validated against a defined schema including allowed characters, length, and pattern (e.g. credit card numbers or telephone, or validating that two related fields are reasonable, such as checking that suburb and zip/postcode match).

Impact: Improper validation of this type of data could lead to Database crashes, resulting in Server-Generated errors, leading to possible sensitive information leaking in Error Messages (this is CWE-550). In addition, any error of this type or crash should be, as obvious, prevented to not mess with the normal application flow.

Solution: In the first project, we had already used Wt_Forms, that already contains some features to avoid these issues. However, in the current project, we improved Forms we were using, adding Optional Fields, correcting some fields length, and restricting Fields to correct data type.

Lastly, we also made sure that every single piece of input was built using Wt_forms since this tool is great for validation.

Note: Every endpoint that uses a form has the Wt_forms called inside, i.e, going around the Frontend and calling API directly (with tools as Postman, for example), does not permit to have more “flexible” input types, which could lead to the app’s crash.



```python
class RegisterForm(Form):
    """Form to register a user in the website"""

    first_name = StringField('', [validators.length(min=3, max=20), validators.DataRequired()],
                             render_kw={'autofocus': True, 'placeholder': 'First Name'})

    last_name = StringField('', [validators.length(min=3, max=20), validators.DataRequired()],
                            render_kw={'autofocus': True, 'placeholder': 'Last Name'})

    username = StringField('', [validators.length(min=3, max=25), validators.DataRequired()],
                           render_kw={'placeholder': 'Username'})

    email = EmailField('', [validators.DataRequired(), validators.Email(), validators.length(min=4, max=25)],
                       render_kw={'placeholder': 'Email'})

    password = PasswordField('', [validators.length(min=3), validators.DataRequired()],
                             render_kw={'placeholder': 'Password', 'class': 'password-field'})

    description = TextAreaField('', [validators.length(min=1)],
                                render_kw={'autofocus': True, 'placeholder': 'Description (Optional)'})

    mobile = IntegerField('', [validators.length(min=9, max=12)],
                          render_kw={'placeholder': 'Mobile (optional)'})
```

Fig.3.4a) *Example: Register Form, that has various types of Data, and different Field Types.*

3.5. Unlimited input file size

Area of the ASVS: Files and Resources.

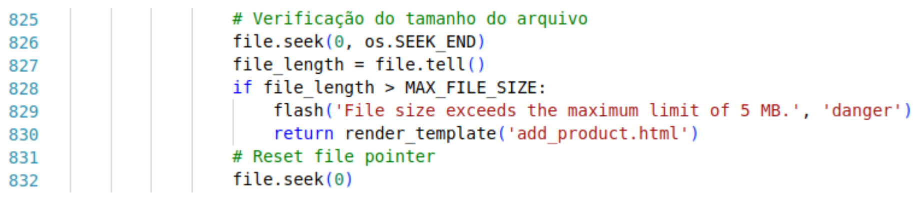
Associated ASVS requirements: 12.1.1.

Description: Verify that the application will not accept large files that could fill up storage or cause a denial of service.

Impact: Despite the input of files is only available for admin users and in an image context, it remains important to have a limit value for file size, to avoid a point of hypothetical denial of service.

Solution: To mitigate a DoS attack, this measure should be followed by a rule that limits the extensions permitted (in our case, we permit .jpg and .png) and limiting number of requests as already discussed above.

But, in concrete, for the issue in discussion now, what we did was to create a variable that defines the limit of a file size. This limit was manually set by us but can be changed by the owner of this code.



```python
                # Verificação do tamanho do arquivo
                file.seek(0, os.SEEK_END)
                file_length = file.tell()
                if file_length > MAX_FILE_SIZE:
                    flash('File size exceeds the maximum limit of 5 MB.', 'danger')
                    return render_template('add_product.html')
                # Reset file pointer
                file.seek(0)
```

Fig.3.5a) *Verification of the file size*

3.6. HTTP parameter pollution attacks

Area of the ASVS: Input Validation Requirements, Sensitive Private Information.

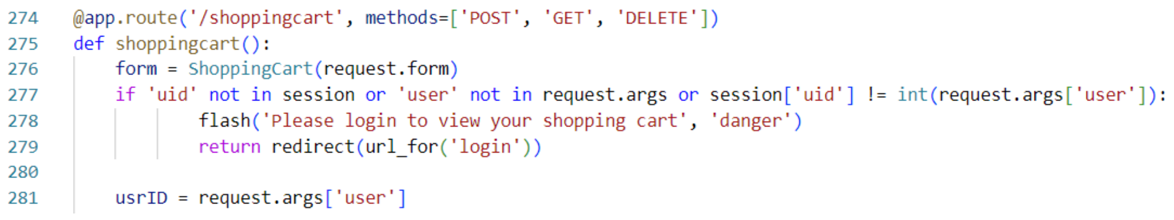
Associated ASVS requirements: 5.1.1, 8.3.1.

Description: Verify that the application has defenses against HTTP parameter pollution attacks, particularly if the application framework makes no distinction about the source of request parameters (GET, POST, cookies, headers, or environment variables).

Verify that sensitive data is sent to the server in the HTTP message body or headers, and that query string parameters from any HTTP verb do not contain sensitive data.

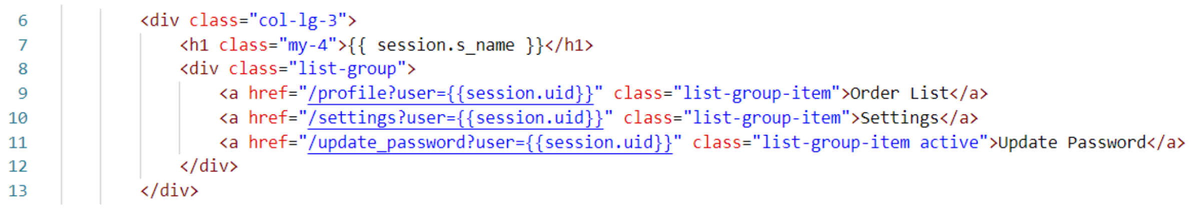
Impact: This vulnerability may allow malicious attackers to manipulate and inject parameters, leading to potential data corruption, unauthorized access, or the execution of unintended actions.

Solution: In our application we found sensitive information that was being passed through the URL, which was the user ID. This parameter was passed when a user accessed their profile page, settings page, shopping cart page and update password page. To fix this issue we removed the user ID from the URL and got the value from the session dictionary instead.



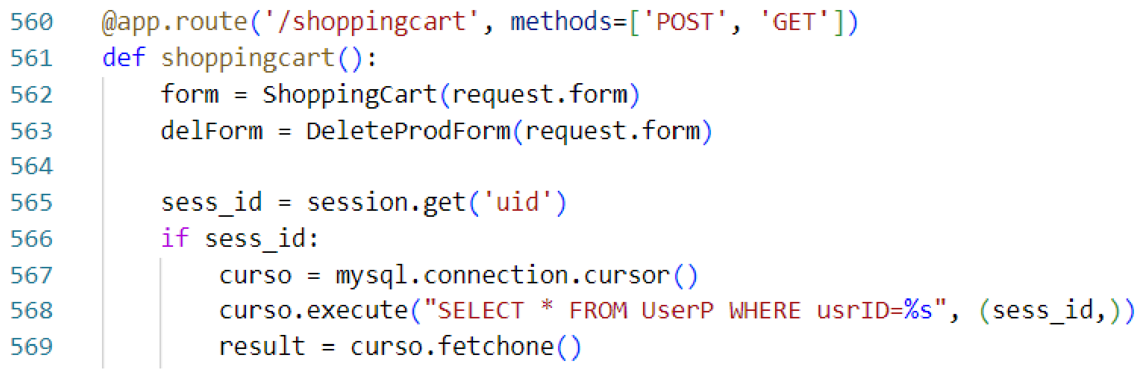
```
@app.route('/shoppingcart', methods=['POST', 'GET', 'DELETE'])
def shoppingcart():
    form = ShoppingCart(request.form)
    if 'uid' not in session or 'user' not in request.args or session['uid'] != int(request.args['user']):
            flash('Please login to view your shopping cart', 'danger')
            return redirect(url_for('login'))

    usrID = request.args['user']
```

Fig.3.6a) *Block of code vulnerable to URL pollution with sensitive information.*



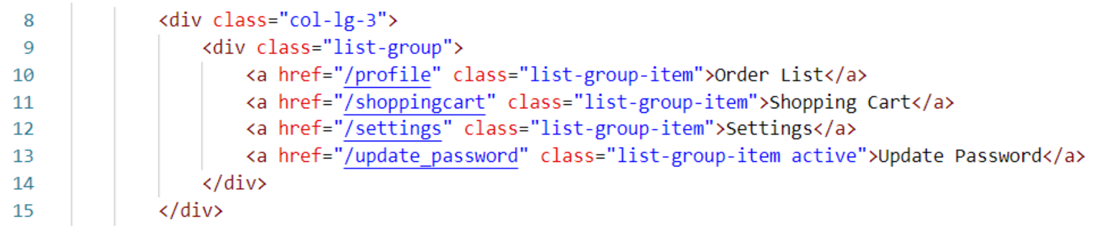
```
        <div class="col-lg-3">
            <h1 class="my-4">{{ session.s_name }}</h1>
            <div class="list-group">
                <a href="/profile?user={{session.uid}}" class="list-group-item">Order List</a>
                <a href="/settings?user={{session.uid}}" class="list-group-item">Settings</a>
                <a href="/update_password?user={{session.uid}}" class="list-group-item active">Update Password</a>
            </div>
        </div>
```

Fig.3.6b) *Links to the pages referred with the inclusion of the user ID.*



```
@app.route('/shoppingcart', methods=['POST', 'GET'])
def shoppingcart():
    form = ShoppingCart(request.form)
    delForm = DeleteProdForm(request.form)

    sess_id = session.get('uid')
    if sess_id:
        curso = mysql.connection.cursor()
        curso.execute("SELECT * FROM UserP WHERE usrID=%s", (sess_id,))
        result = curso.fetchone()
```

Fig.3.6c) *Obtaining the user ID through the session dictionary.*



```
<div class="col-lg-3">
    <div class="list-group">
        <a href="/profile" class="list-group-item">Order List</a>
        <a href="/shoppingcart" class="list-group-item">Shopping Cart</a>
        <a href="/settings" class="list-group-item">Settings</a>
        <a href="/update_password" class="list-group-item active">Update Password</a>
    </div>
</div>
```

Fig.3.6d) *Links to the pages referred without the inclusion of the user ID.*

3.7. Absence of secure encrypted client connections over TLS

Area of the ASVS: Communication Security.

Associated ASVS requirements: 9.1.1, 9.1.2, 9.1.3.

Description: Communications with the web browser end-user are made using clear text HTTP messages, i.e. no secure encrypted protocols are used to ensure sensitive data contained in the HTTP message body and headers, such as account credentials and authentication factors like an OTP secret token, are illegible to third parties. Furthermore, there are no message authentication mechanisms.

Impact: With relative ease, HTTP messages can be captured and read in clear text using publicly available tools such as *Wireshark* and *tcp-dump*. Sensitive data, such as account passwords and other authentication credentials can be stolen, compromising account authenticity.

A lack of a secure endpoint authentication mechanism also opens the possibility for man-in-the-middle (MITM) attacks.

Solution: To solve this problem, *Flask* supports the use of Transport Layer Security (TLS) [8]. TLS is a cryptographic communications protocol applicable to various forms of communication and their protocols but most often used over the HTTP protocol (HTTPS), which is the case here. It allows for endpoint message authentication and encrypted messaging, greatly mitigating MITM attacks and other attempts to compromise confidentiality, integrity, or authenticity.

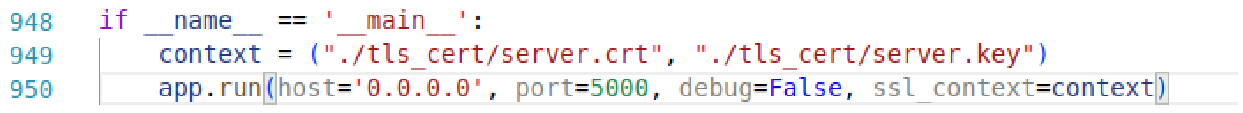
As part of the endpoint message authentication mechanism, certificates are exchanged between the client and web server to authenticate each. These end user certificates (in this context, end users represent both the client and server) are typically obtained from a trusted certification authority (CA), signed using the private key from a **root** CA (part of the chain-of-trust). End-user operating systems and browsers contain a set of trusted root CAs that can be relied upon to validate end user certificates.

In our case, we did not obtain a certificate from a trusted CA. Obtaining a trustworthy certificate involves obtaining a domain name for our website, which implies some monetary investment and budget. Due to a lack of monetary budget, we decided to create our own self-signed certificates with `openssl` [7]. This should NOT be endorsed in a real production scenario with real end-users as self-signed certificates don't ensure the legitimacy of a website.

BASH shell command used to generate self-signed certificate and respective private key:

```
$ openssl req -newkey rsa:2048 -keyout domain.key -x509 -days 365 -out domain.crt
```

The private key (`domain.key`) uses RSA with a size of 2048, both recommended parameters that balance performance and security. The X.509 format is also recommended for the public key certificate (`domain.crt`).



```
if __name__ == '__main__':
    context = ("./tls_cert/server.crt", "./tls_cert/server.key")
    app.run(host='0.0.0.0', port=5000, debug=False, ssl_context=context)
```

Fig.3.7a) Running Flask with a defined SSL context**.**

It should also be noted that using *Nginx* to proxy the *Flask* web server is preferable in a production environment as it has more security features and controls for the developer, mainly regarding communication security. For the purposes of this simple project, what *Flask* provides is enough.

For requirements 9.1.2 and 9.1.3, `openssl` was used to verify the TLS protocol version and *ciphersuite* being used:

```
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384

Server public key is 2048 bit
```

The *ciphersuite* and TLS protocol version used by *Flask* depend on the version of *OpenSSL* linked to the *Python* interpreter. Both the TLS protocol version and *ciphersuite* comply with the aforementioned ASVS requirements.

3.8. Unverified Passwords Against Discovered Sets

Area of the ASVS: Password Security Credentials.

Associated ASVS requirements: 2.1.7.

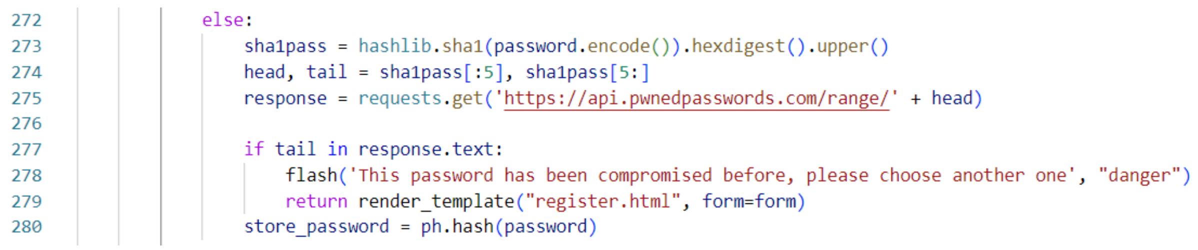
Description: Verify that passwords submitted during account registration, login, and password change are checked against a set of breached passwords either locally (such as the top 1,000 or 10,000 most common passwords which match the system's password policy) or using an external API. If using an API, a zero-knowledge proof or other mechanism should be used to ensure that the plain text password is not sent or used in verifying the breach status of the password. If the password is breached, the application must require the user to set a new non-breached password.

Impact: Minimize the use of “weak” passwords, which were also breached for this reason, and other passwords that may have been breached, to help a user choose a “good” password.

Solution: To address this issue, we decided to use the Have I Been Pawned API (commonly named HIBP), mainly because it is the service that has the biggest number of breached passwords (over 613M).

That said, to comply with the ASVS requirement, what is done is to encrypt the user's password with SHA1, since this is the Hash used to store all Hashes in HIBP API. Then, HIBP API is going to be called, passing the first five Hash characters, since HIBP uses a k-Anonymity model to avoid a 1:1 match, that could be intercepted by a man-in-the-middle attack. If the password was not breached, the password in clear text is going to be Hashed by Argon2, to then be stored in DB.

Lastly, clarify that the code that is going to be presented next is working on register form and in login page, because if the password was breached in the meantime, the user must set a new password.



```
else:
    sha1pass = hashlib.sha1(password.encode()).hexdigest().upper()
    head, tail = sha1pass[:5], sha1pass[5:]
    response = requests.get('https://api.pwnedpasswords.com/range/' + head)

    if tail in response.text:
        flash('This password has been compromised before, please choose another one', "danger")
        return render_template("register.html", form=form)
    store_password = ph.hash(password)
```

Fig.3.8a) *Block of code developed to verify if a password inputted by the user has been*

3.9. Missing CSP (Content Security Policy)

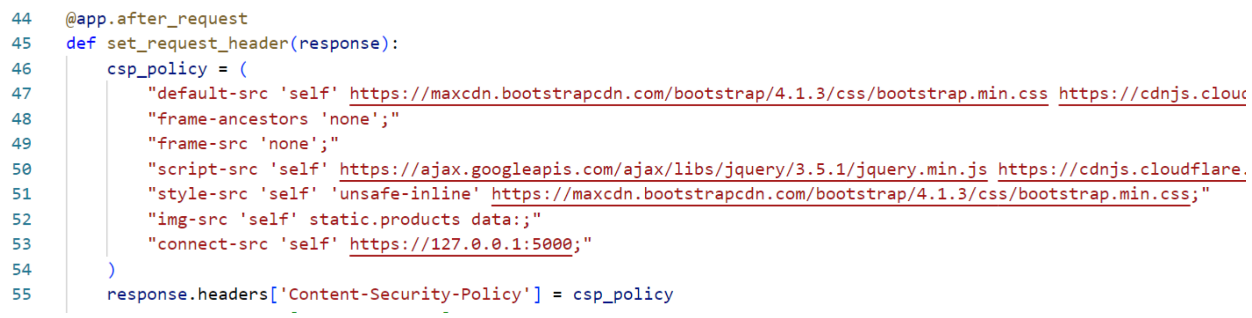
Area of the ASVS: HTTP Security Headers Requirements.

Associated ASVS requirements: 14.4.3.

Description: 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.

Impact: The CSP defined is one of the most important things in a website, because in addition to XSS, click-jacking is also resolved.

Solution: The CSP policy follows a whitelist approach. Therefore, we needed to permit the importing of scripts for CSS, jQuery, and Bootstrap to work properly. Moreover, we achieved having a "script-src" "self" policy, and for this, we needed to refactor all JS presented in our app. To apply this policy to "style-src", we would also need to do the same, but for lack of time, this specific part was not updated and CSS inline is presented in ou app code.



```
@app.after_request
def set_request_header(response):
    csp_policy = (
        "default-src 'self' https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css https://cdnjs.clouc
        "frame-ancestors 'none';"
        "frame-src 'none';"
        "script-src 'self' https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js https://cdnjs.cloudflare.
        "style-src 'self' 'unsafe-inline' https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css;"
        "img-src 'self' static.products data:;"
        "connect-src 'self' https://127.0.0.1:5000;"
    )
    response.headers['Content-Security-Policy'] = csp_policy
```

Fig.3.9a) *Content Security Policy developed.*

4. Two Additional Security Features

In addition to the ASVS audit and implemented fixes, two additional security features were implemented as well. One of these features (password strength evaluation) contributes to solving some of the identified issues in the audit.

4.1. Password strength evaluation

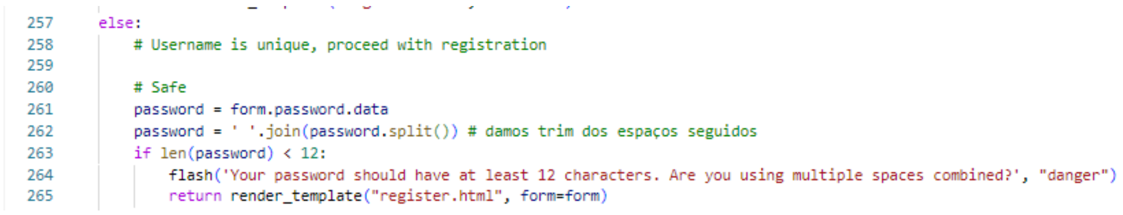
In the previous project, we developed some conditions that the user needed to follow to have a strong password and with these conditions we already fulfill the requirements of the following ASVS's:

- ASVS 2.1.4 (Verify that any printable Unicode character, including language neutral characters such as spaces and Emojis are permitted in passwords);
- ASVS 2.1.5 (Verify users can change their password);
- ASVS 2.1.6 (Verify that password change functionality requires the user's current and new password).

However, the number of valid ASVS were so low that we decided to improve our password strength, and to do so we implemented the following points:

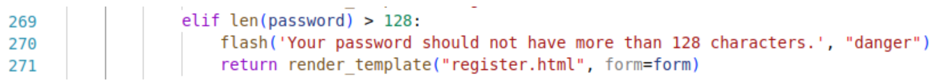
4.1.1. Password length

In the first project we defined that the minimum length of the password should be 8 characters but that wasn’t enough to fulfill the ASVS 2.1.1 (verify that user set passwords are at least 12 characters in length, after multiple spaces are combined), so we raised the minimum value to 12 characters and if the user inputted multiple consecutive white-spaces, they were replaced by a single one, complying with ASVS 2.1.3. To comply with ASVS 2.1.2, we added an *if* statement that does not allow a password with more than 128 characters. The if statements are separated to give the user a more specific error message if something is done wrong.



```
else:
    # Username is unique, proceed with registration

    # Safe
    password = form.password.data
    password = ' '.join(password.split()) # damos trim dos espaços seguidos
    if len(password) < 12:
        flash('Your password should have at least 12 characters. Are you using multiple spaces combined?', "danger")
        return render_template("register.html", form=form)
```

Fig.4.1.1a) *Remove the consecutive whitespaces and check length condition.*



```
elif len(password) > 128:
    flash('Your password should not have more than 128 characters.', "danger")
    return render_template("register.html", form=form)
```

Fig.3a) *Check length condition again*

4.1.2. Element to view temporarily entire masked password

We also decided to accomplish the requirements present in the ASVS 2.1.12, which describe the functionality of allowing a user to choose 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.

Without this feature, users may encounter difficulties in accurately inputting their passwords that may lead to frustration, additionally, in scenarios where users need to confirm or review their password entries, the absence of this element may result in repeated and potentially insecure attempts to input the password.

So, for this issue we developed a simple function in *JavaScript* and added an icon that a user could press, and it would show the entire password inputted, and to hide it the user needed to press the icon again. We included this functionality in the login, register and update password pages, because all of them have password fields.

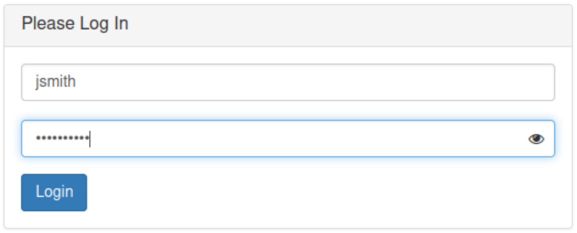


Fig.4.1.2a) *Initial Login Form.*

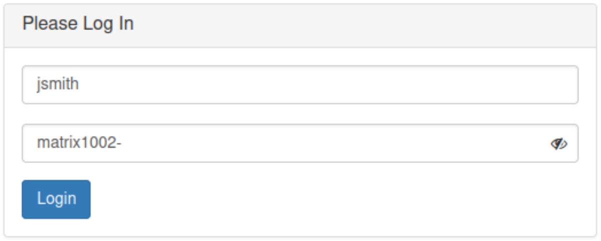


Fig.4.1.2b) *When the icon is pressed, password appears.*

4.1.3. Verify Passwords Against Discovered Sets

This requirement has been already described above in Section *3.8. Unverified Passwords Against Discovered Sets.*

4.1.4. Password Meter

Another feature that we created to improve the password strength was a password meter, that's described by the ASVS 2.1.8, **v**erify that a password strength meter is provided to help users set a stronger password.

A password strength meter serves as a valuable tool in guiding the users towards creating robust and resilient passwords by offering real-time feedback on the strength of their chosen passwords. The absence of this feature users may select weaker passwords that are more susceptible to exploitation by malicious attackers, also without this guidance the risk of unauthorized access through password-related attacks increases, such as brute force attacks or dictionary attacks. However, it's noted to that a poor password meter can guide a user to misleading, as stated in this paper, referenced by NIST [24].

With this ideas in mind, our meter implementation ended not complying neither with ASVS nor NIST, because we wanted to implement a solution that would take into account, the chars used and if that sequence of chars was related or not to some kind of domain (Example: username or company), reflecting in this way in the total length of the password (as discussed by some ASVS authors here). Besides all of this, we also found what we think that is the removal of ASVS 2.1.8 in future versions, here.

Therefore, to solve this issue, we created a password meter in the pages that the user needed to create a new password (register and update password page). This password meter isn't something mandatory that users must follow, we just included it to give them the choice to create a more secure and strong password, because. The meter contains 4 fields:

- The inclusion of upper-case letters;
- The inclusion of special characters such as ':', '_', '.', etc.;
- The length of the password should be greater than 12 characters;
- The inclusion of numbers.

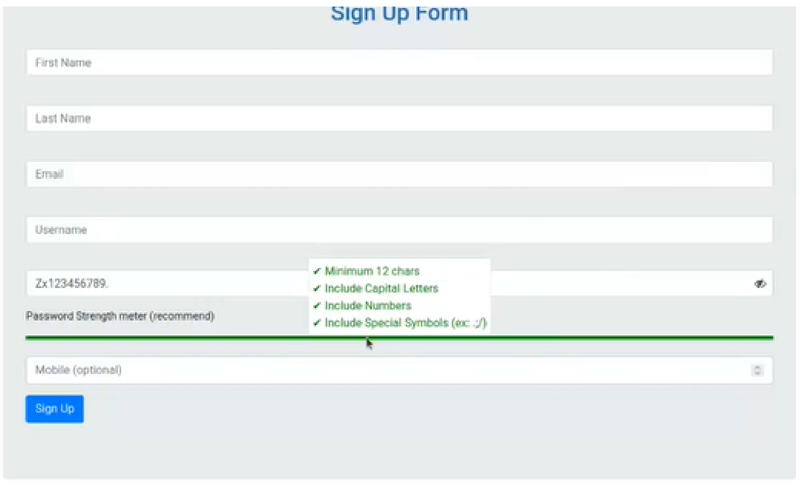
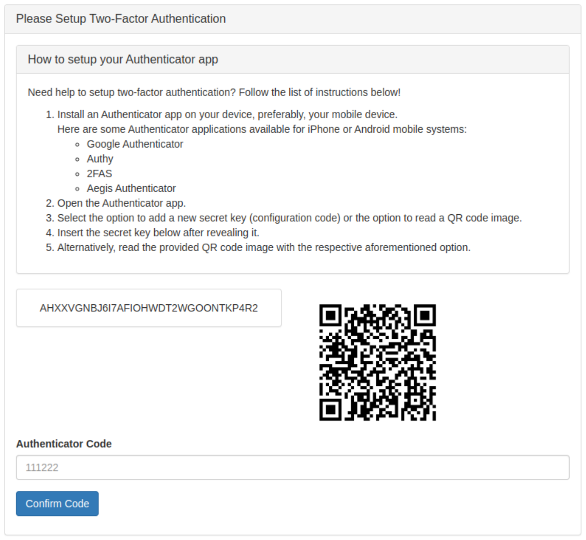


Fig.4.1.4a) *Example of a password that meets all meter requirements.*

4.2. Multi-factor Authentication (MFA) using TOTP

To implement multi-factor authentication (MFA) we opted to use time-based one-time passwords (TOTP). TOTP is a common alternative mean of authentication which uses one-time passwords (OTP), i.e. passwords (codes) that can only be used once and are then discarded. TOTP codes are generated based on a secret token/key and on the current time instant, being valid for a limited period.

Clients that have yet to set up TOTP are greeted with this page, prompting them to set up an authenticator app that calculates currently valid OTP codes, like *Google Authenticator*, being free to use the secret token directly or the encoded QR image representing the URI of the token:



Please Setup Two-Factor Authentication

How to setup your Authenticator app

Need help to setup two-factor authentication? Follow the list of instructions below!

1. Install an Authenticator app on your device, preferably, your mobile device.
   Here are some Authenticator applications available for iPhone or Android mobile systems:
   - Google Authenticator
   - Authy
   - 2FAS
   - Aegis Authenticator
2. Open the Authenticator app.
3. Select the option to add a new secret key (configuration code) or the option to read a QR code image.
4. Insert the secret key below after revealing it.
5. Alternatively, read the provided QR code image with the respective aforementioned option.

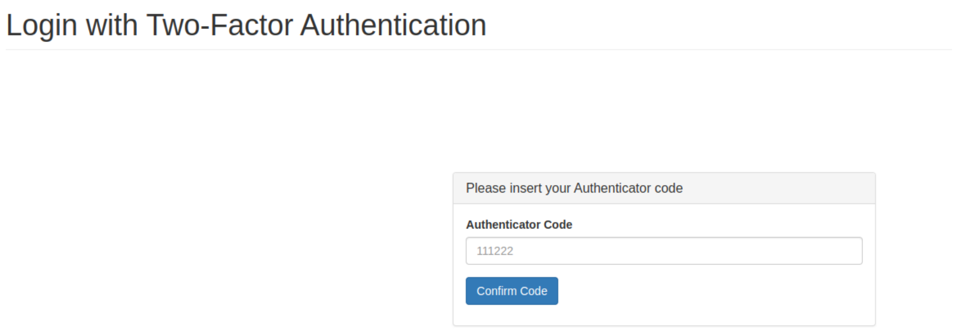
AHXXVGNBJ6I7AFIOHWDT2WGOONTKP4R2

Authenticator Code

111222

Fig.4.2a) *Page to configure Two-Factor Authentication.*

Clients that have already setup TOTP are greeted with this page after entering valid username-password credentials:



Login with Two-Factor Authentication

Please insert your Authenticator code

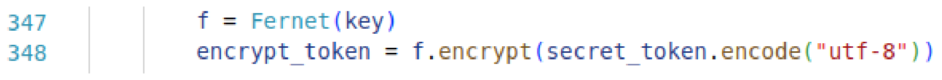
Authenticator Code

111222

Fig.4.2a) *TOTP login page.*

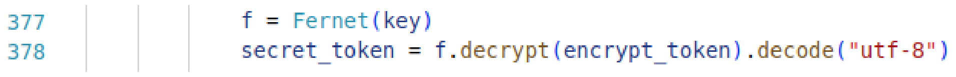
We used the *PyOTP* module to manage TOTP authentication in our application. The module is used to generate secret tokens and validate codes given by the end user.

In addition to this, the ASVS V2.8 requirements point towards a need to store OTP secrets in a secure manner, requiring encryption. We opted for symmetric encryption, storing symmetric keys in the Flask server's operating system as plaintext files in the application's filesystem. Symmetric encryption is done using the *Fernet* API of the Python cryptography module, which on its own uses AES-128 in CBC mode and HMAC message authentication, with the latter being a complement to the CBC cipher mode's padding problems.



```
f = Fernet(key)
encrypt_token = f.encrypt(secret_token.encode("utf-8"))
```

Fig.4.2c) *Fernet encryption of TOTP secret token.*



```
f = Fernet(key)
secret_token = f.decrypt(encrypt_token).decode("utf-8")
```

Fig.4.2d) *Fernet decryption of encrypted TOTP secret token*

In addition to this, the ASVS V2.8 requirements point towards a need to store OTP secrets in a secure manner, requiring encryption. We opted for symmetric encryption, storing symmetric keys in the Flask server's operating system secret store (D-Bus Secret Service API), while the secret token is stored as *ciphertext* in the user database table's `secretToken` column.

On top of secure secret storage, TOTPs should, obviously, only be usable once, so an additional column was added to the user database table, `lastOTP`, which keeps track of the last code used by the client, being compared against future TOTP code inputs to invalidate the reuse of OTPs. OTP codes used more than once are logged and an e-mail notification is sent to the address specified at the user's profile using the Simple Mail Transfer Protocol (SMTP) on top of SSL/TLS.

5. Extra Security Features

In addition to the aforementioned issues and guided by the results collected from the tests conducted using ZAP, we have identified further security vulnerabilities, subsequently we decided to address and solve them.

5.1. X-Content-Type-Options Header Missing

Area of the ASVS: HTTP Security Headers Requirements.

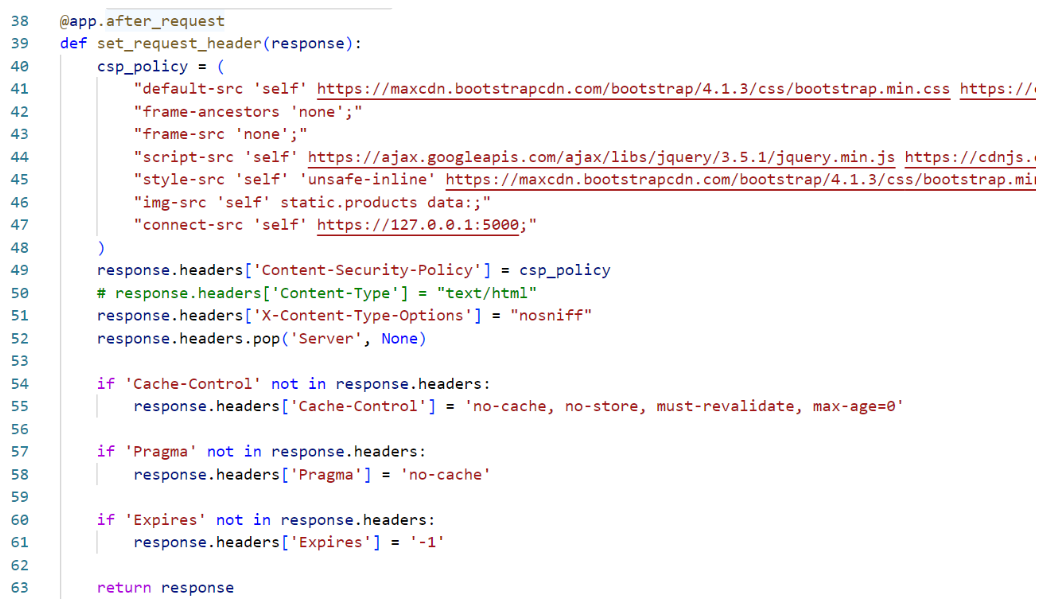
Associated ASVS requirements: 14.4.4.

Description: Verify that all responses contain a X-Content-Type-Options: “nosniff” header.

Impact: This header plays a crucial role in preventing MIME type sniffing, a technique where browsers may override the declared content type and interpret files as a different MIME type. Without the inclusion of this security header, attackers could exploit MIME type mismatches to execute Cross Site Scripting (XSS) attacks, leading to unauthorized execution of malicious scripts in the user’s browser.

Solution: To solve this issue, we defined the function set_request_header with the decorator @app.after_request, this allows us to call this function after each request, regardless the occurrence of an exception during the request.

In this function besides defining the CSP, has mentioned above, we added the X-Content-Type-Options and the Content-Type with the values “nosniff” and “text/html”, respectively, to the header of each request.



```
@app.after_request
def set_request_header(response):
    csp_policy = (
        "default-src 'self' https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css https://
        "frame-ancestors 'none';"
        "frame-src 'none';"
        "script-src 'self' https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js https://cdnjs.
        "style-src 'self' 'unsafe-inline' https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.mi
        "img-src 'self' static.products data:;"
        "connect-src 'self' https://127.0.0.1:5000;"
    )
    response.headers['Content-Security-Policy'] = csp_policy
    # response.headers['Content-Type'] = "text/html"
    response.headers['X-Content-Type-Options'] = "nosniff"
    response.headers.pop('Server', None)

    if 'Cache-Control' not in response.headers:
        response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate, max-age=0'

    if 'Pragma' not in response.headers:
        response.headers['Pragma'] = 'no-cache'

    if 'Expires' not in response.headers:
        response.headers['Expires'] = '-1'

    return response
```

Fig.5.1a) *Function set_request_header.*

5.2. Secure Session Cookie

Area of the ASVS: Cookie-Based Session Management, Session Binding Requirements.

Associated ASVS requirements: 3.2.3, 3.4.1, 3.4.3, 3.4.4.

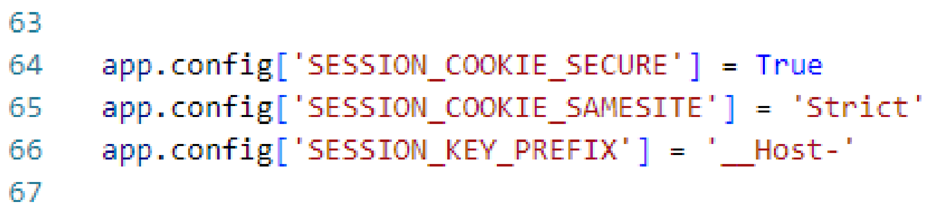
Description: Verify the application only stores session tokens in the browser using secure methods such as appropriately secured cookies or HTML 5 session storage.

Verify that cookie-based session tokens have the 'Secure' attribute set. Verify that cookie-based session tokens utilize the 'SameSite' attribute to limit exposure to cross-site request forgery attacks.

Verify that cookie-based session tokens use "__Host-" prefix (see references) to provide session cookie confidentiality.

Impact: If the session cookie does not have these attributes, our app becomes more susceptible to session hijacking, CSRF attacks, and potential exposure of sensitive information.

Solution: We added the corresponding config settings to the Flask application, guarantying that the session cookie is secure.



```
app.config['SESSION_COOKIE_SECURE'] = True
app.config['SESSION_COOKIE_SAMESITE'] = 'Strict'
app.config['SESSION_KEY_PREFIX'] = '__Host-'
```

Fig.5.2a) *Attributes added to solve this vulnerability.*

5.3. Incorrect session Logout

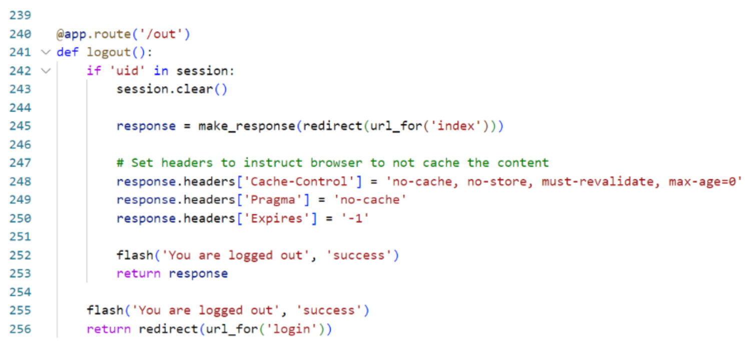
Area of the ASVS: Session Logout and Time Out Requirements.

Associated ASVS requirements: 3.3.1.

Description: Verify that logout and expiration invalidate the session token, such that the back button or a downstream relying party does not resume an authenticated session, including across relying parties.

Impact: Without proper session token invalidation, users may remain susceptible to session hijacking, allowing unauthorize access to their accounts even after purported logout or session timeout. Attackers exploiting this loophole could gain unauthorized access to sensitive information or perform actions on behalf of the user, compromising the integrity and confidentiality of the entire authentication system.

Solution: To fix this vulnerability we added the following configurations when the user logs out



```
@app.route('/out')
def logout():
    if 'uid' in session:
        session.clear()

        response = make_response(redirect(url_for('index')))

        # Set headers to instruct browser to not cache the content
        response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate, max-age=0'
        response.headers['Pragma'] = 'no-cache'
        response.headers['Expires'] = '-1'

        flash('You are logged out', 'success')
        return response

    flash('You are logged out', 'success')
    return redirect(url_for('login'))
```

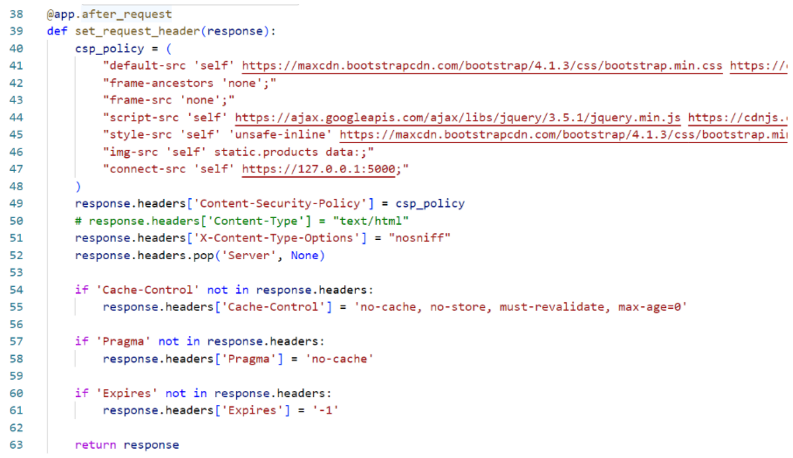
Fig.5.3a) *Log out endpoint with configurations to fix vulnerability.*

The ‘Cache-Control’ header with the value 'no-cache, no-store, must-revalidate, max-age=0' instructs the browsers not to cache the response, revalidate it on every access, and refrain from storing it.

The ‘Pragma Header’ with ‘no-cache’ further enforces caching prevention.

The ‘Expires’ header with the value ‘-1’ sets an expiration date in the past, ensuring immediate expiration.

We also added these configurations to our set_request_header to also ensure that there isn’t any leak of sensitive of information.



```python
@app.after_request
def set_request_header(response):
    csp_policy = (
        "default-src 'self' https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.min.css https://
        "frame-ancestors 'none';"
        "frame-src 'none';"
        "script-src 'self' https://ajax.googleapis.com/ajax/libs/jquery/3.5.1/jquery.min.js https://cdnjs.
        "style-src 'self' 'unsafe-inline' https://maxcdn.bootstrapcdn.com/bootstrap/4.1.3/css/bootstrap.mi
        "img-src 'self' static.products data:;"
        "connect-src 'self' https://127.0.0.1:5000;"
    )
    response.headers['Content-Security-Policy'] = csp_policy
    # response.headers['Content-Type'] = "text/html"
    response.headers['X-Content-Type-Options'] = "nosniff"
    response.headers.pop('Server', None)

    if 'Cache-Control' not in response.headers:
        response.headers['Cache-Control'] = 'no-cache, no-store, must-revalidate, max-age=0'

    if 'Pragma' not in response.headers:
        response.headers['Pragma'] = 'no-cache'

    if 'Expires' not in response.headers:
        response.headers['Expires'] = '-1'

    return response
```

Fig.5.3a) *Function set_request_header with configurations to fix vulnerability.*

6. References

1. OWASP ASVS checklist for audits
2. OWASP ASVS 4.0 testing guide
3. OWASP ZAP
4. Chrome Dev Tools
5. Curl
6. OWASP ASVS
7. OpenSSL
8. Transport Layer Security by Mozilla
9. MySQL
10. Flask
11. Linux Containers
12. TLS Basics by Internet Security
13. Creating self-signed certificates with OpenSSL
14. SSL/TSL Best Practices
15. X.509 Certificates
16. Flask Security Considerations
17. Flask TLS Setup Tutorial
18. QR image encoding utility function
19. TOTP in Python using PyOTP
20. PyOTP Documentation
21. Have I been Pwned API
22. OTP Overview by OneLogin
23. TOTP Overview by Twilio
24. NIST Digital Identity Guidelines
25. CSP Setup
26. Argon2
27. Fernet
28. Sending E-Mails in Python