



Application Security Verification Standards

Report

Gonçalo Lima - 108254
Tiago Fonseca - 107266
Beatriz Ferreira - 107214
Tomás Fonseca - 107245
Francisco Murcela - 108815
Grupo 10

Universidade de Aveiro
Segurança Informática e
Nas Organizações

Index

Introduction	3
Application Security Verification Standards	4
ASVS (Level 1) [CWE-212] – Sensitive Private Data (Data Protection)	4
ASVS (Level 1) [CWE-285] – Sensitive Private Data (Data Protection)	7
ASVS (Level 1) [CWE-235] – Input Validation Requirements (Input Validation).....	9
ASVS (Level 1) [CWE-287] – Out of Band Verifier Requirements (Authentication).....	11
ASVS (Level 1) [CWE-319] – Communications Security Requirements (Communication Security)	13
ASVS (Level 1) [CWE-326] – Communications Security Requirements (Communication Security)	14
ASVS (Level 1) [CWE-350] – Deployed Application Integrity Controls (Malicious Code)	15
ASVS (Level 1) [CWE-400] – File Upload Requirements (File and Resources)	16
ASVS (Level 1) [CWE-918] – SSRF Protection Requirements (File and Resources).....	21
ASVS (Level 1) [CWE-497] – Unintended Security Disclosure Requirements (Configuration).....	22
ASVS (Level 1) [CWE-548] – Other Access Control Considerations (Access Control)	23
ASVS (Level 1) [CWE-598] – Generic Web Service Security Verification Requirements (Web Services).....	24
ASVS (Level 1) [CWE-778] – Defenses Against Session Management Exploits (Session Management)	26
Implementation and Deployment	29
References	30

Introduction

This report details the outcomes of a comprehensive project centered on developing an online store specializing in memorabilia products from the DETI (Department of Electronics, Telecommunications, and Informatics) at the University of Aveiro. Beyond meeting the standard requirements of an e-commerce platform, this project involves an additional challenge: identifying and mitigating vulnerabilities that might not be immediately visible, potentially compromising the system's integrity and security, meeting the security standards in the industry.

Throughout this report, we meticulously implemented various fixes to meet the *Application Security Verification Standards (ASVS)*, with a specific focus on a designated set of *Common Weakness Enumeration (CWE)*, which includes categories such as *CWE-212*, *CWE-285*, *CWE-235*, *CWE-287*, *CWE-319*, *CWE-326*, *CWE-350*, *CWE-400*, *CWE-918*, *CWE-497*, *CWE-548*, *CWE-598* and *CWE-778*.

Our primary aim with this project is to expand our knowledge about vulnerabilities and enhance our capability to identify solutions for them. We strive for a deeper understanding of system weaknesses and the ability to implement effective measures for their protection. We believe that this project will contribute to refining our skills and fostering safer online environments.

The web application (secure version) has been fully implemented and can be found [here](#).



Application Security Verification Standards

ASVS (Level 1) [CWE-212] – Sensitive Private Data (Data Protection)

Previously, users were unable to export their data on demand, but now they have the freedom to do so whenever they choose.

Profile

In the Profile View, you'll find a button labelled “Retrieve My Data”. Clicking this button triggers the “Get User Data View”. This view generates an Excel file containing all the user data, enabling the user to download it.

Definition of the “Get User Data View”:

```
@views.route('/get_user_data/<id>', methods=['GET'])
def get_user_data(id):

    if id == None:
        return redirect(url_for("views.login"))

    current_directory = generate_excel_user_data(id)

    # Send the Excel file as a downloadable attachment
    return send_file(current_directory + f"\\database\\user_data\\{id}.xlsx",
as_attachment=True)
```

Moreover, within the “Get User Data View”, the aim is to utilize the function `generate_excel_user_data(id)` located in the `UserManagement.py` file. This function is responsible for creating the Excel file.



Definition of the function:

```
def generate_excel_user_data(id):

    data = get_user_data_by_id(id)

    # Convert data to DataFrame if the keys exist
    df_personal_info = pd.DataFrame(data.get('personal_info', []))
    df_reviews = pd.DataFrame(data.get('reviews', []))
    df_orders = pd.DataFrame(data.get('orders', []))

    # Read the HTML and CSS files
    if os.name == "nt":
        # Get the current working directory
        current_directory = os.path.dirname(os.path.abspath(__file__))
    else:
        # Get the current working directory
        current_directory = os.path.dirname(os.path.abspath(__file__))

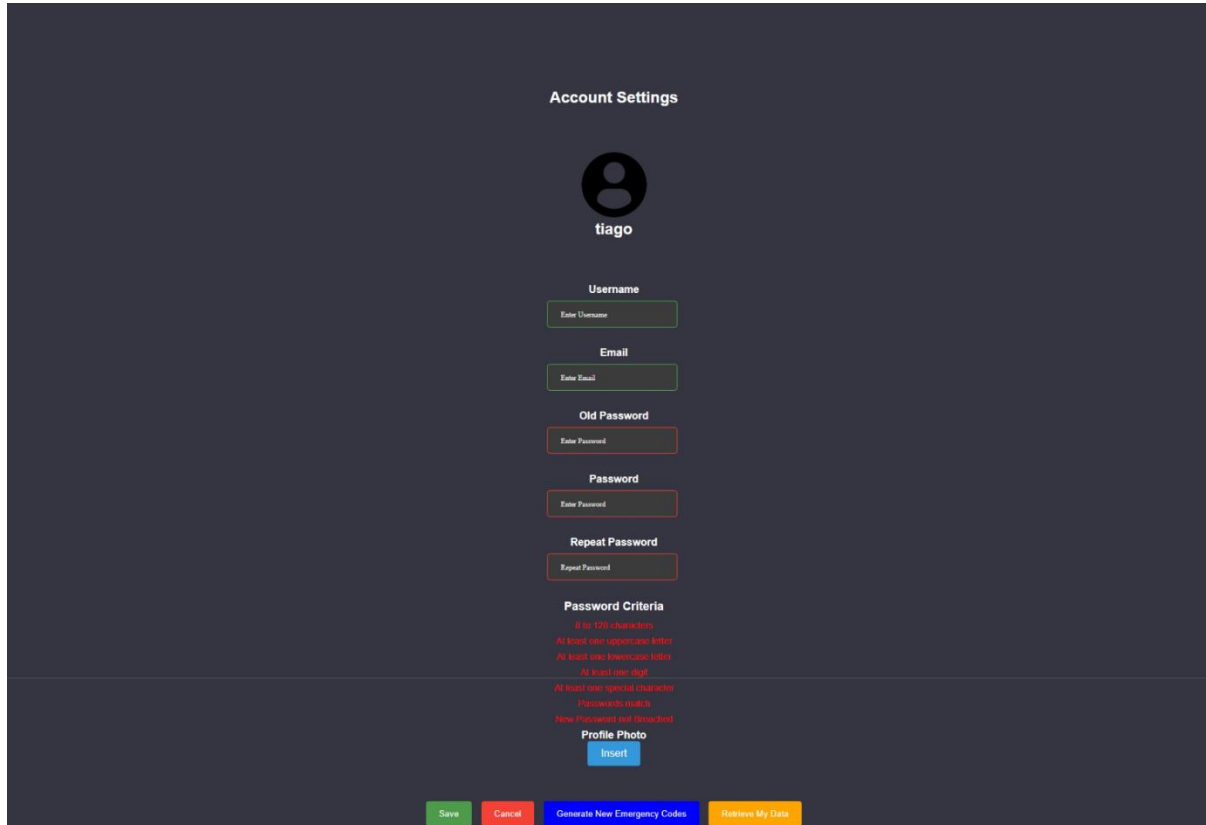
    # Check if the directory exists
    user_data_directory = os.path.join(current_directory, "database", "user_data")
    if not os.path.exists(user_data_directory):
        os.makedirs(user_data_directory)

    # Create an Excel writer using pandas with openpyxl engine
    with pd.ExcelWriter(os.path.join(user_data_directory, f"{id}.xlsx"),
engine='openpyxl') as writer:
        # Write each DataFrame to a separate sheet in the Excel file if they exist
        if not df_personal_info.empty:
            df_personal_info.to_excel(writer, sheet_name='Personal Info',
index=False)
        if not df_reviews.empty:
            df_reviews.to_excel(writer, sheet_name='Reviews', index=False)
        if not df_orders.empty:
            df_orders.to_excel(writer, sheet_name='Orders', index=False)

    return current_directory
```

In this manner, the user can ultimately export their data.

Demonstration:



The screenshot shows a dark-themed 'Account Settings' interface for a user named 'tiago'. The page includes input fields for Username, Email, Old Password, Password, and Repeat Password. Below these is a 'Password Criteria' section with red text indicating requirements: 8 to 128 characters, at least one uppercase letter, at least one lowercase letter, at least one digit, at least one special character, and password must not be the same as the previous one. There is also a 'Profile Photo' section with an 'Insert' button. At the bottom, there are four buttons: 'Save' (green), 'Cancel' (red), 'Generate New Emergency Codes' (blue), and 'Restore My Data' (orange).

[Excel File with User Data](#)



ASVS (Level 1) [CWE-285] – Sensitive Private Data (Data Protection)

Verify that users are provided clear language regarding collection and use of supplied personal information and that users have provided opt-in consent for the use of that data before it is used in any way.

Information Collected View

Previously, users were unaware of the data being collected, depriving them of any choice in the matter. However, following the completion of the signup form, they are now provided with the option to make informed decisions regarding their data.

In the “*Validate Consent View*” the user can choose to consent or not to the data collection used by the app, if he chooses not to consent, he will not be able to create an account.

Definition of the “*Validate Consent View*”:

```
@views.route('/validate_consent', methods=['POST'])
def validate_consent():
    consent = request.form.get('consent')
    if consent:
        temp_id = str(generate_random_id_totp_temp())

        username = session.get("signup_username")
        email = session.get("signup_email")
        hashed_password = session.get("signup_hashed_password")

        secret_key, secret_key_timestamp, qr_code_base64 =
generate_totp_attributes(username)

        store_totp_stage(temp_id, username, secret_key, secret_key_timestamp,
email, hashed_password)

        # Clean the session variables
        session.pop("signup_username", None)
        session.pop("signup_email", None)
        session.pop("signup_hashed_password", None)

        return render_template('totp_signup.html', secret_key=secret_key,
qr_code=qr_code_base64, id=temp_id)
    else:
        return redirect(url_for("views.signup"))
```



Demonstration:

Data Collection Information

We may collect the following information:

- ☐ Profile Photo
- ☐ Username
- ☐ Email
- ☐ Shipping Address
- ☐ Products that may be purchased

☐ I consent to the collection and use of the above information for specified purposes.

[Submit](#)



ASVS (Level 1) [CWE-235] – Input Validation Requirements (Input Validation)

Verify that the application has defences 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).

Applied to all the Views

The Views are restricted to access only the designated methods assigned to them, thereby distinguishing the source of request parameters.

The applied fix involves adding an additional parameter to the *Views* definition. While this parameter existed in some cases, it was missing in others prior to the update.



Definition of the parameters, with a random *View* serving has the example:

```
# This route is used to let the user reset their password
@views.route("/reset-password", methods=["GET", "POST"])
def reset_password():
    if request.method == "POST":
        email = request.form.get("email")
        if is_valid_input(email) == False:
            return render_template("reset-password.html", message="Invalid
email.")

        user = search_user_by_email(email)
        if user is None:
            # If the user doesn't exist, return the signup page
            return redirect(url_for("views.signup"))
        else:
            # Generate a unique reset token
            reset_token = generate_reset_token()
            # Store the reset token in the user's record in the database
            set_reset_token_for_user(user, reset_token)
            # Send a password reset email with the token
            send_password_reset_email(email, reset_token)
            return redirect(url_for("views.login"))
    else:
        return render_template("reset-password.html")
```

This fix restricts access to the *Views* solely for the methods they are assigned to, bolstering protection against potential *HTTP* parameter pollution attacks.



ASVS (Level 1) [CWE-287] – Out of Band Verifier Requirements (Authentication)

Verify that the out of band verifier expires out of band authentication requests, codes, or tokens after 10 minutes. Verify that the out of band verifier authentication requests, codes, or tokens are only usable once, and only for the original authentication request.

Reset Password

The password reset process incorporates tokens that must adhere to specific properties in accordance with the ASVS requirements.

To accomplish this, the “Reset Password View” utilized the `is_valid_reset_token(reset_token)` function within the `UserManagement.py` file to validate tokens. Additionally, the `clear_reset_token(user)` function was employed to eliminate tokens from the database upon expiration or usage.

Definition of the Views:

```
# This route is used to let the user reset their password
@views.route("/reset-password", methods=["GET", "POST"])
def reset_password():
    if request.method == "POST":
        email = request.form.get("email")
        if is_valid_input(email) == False:
            return render_template("reset-password.html", message="Invalid email.")

        user = search_user_by_email(email)
        if user is None:
            # If the user doesn't exist, return the signup page
            return redirect(url_for("views.signup"))
        else:
            # Generate a unique reset token
            reset_token = generate_reset_token()
            # Store the reset token in the user's record in the database
            set_reset_token_for_user(user, reset_token)
            # Send a password reset email with the token
            send_password_reset_email(email, reset_token)
            return redirect(url_for("views.login"))
    else:
        return render_template("reset-password.html")
```

Definition of the functions:

```
# This function checks if the reset token is valid and not expired.
def is_valid_reset_token(reset_token):
    user = get_user_by_reset_token(reset_token)

    if user:
        # Assuming 'reset_token_timestamp' is a field in your User model
        # to store the token creation timestamp.
        token_timestamp = user[4]

        # Define the token expiration time (10 minutes).
        token_expiration_time = timedelta(minutes=10)

        # Check if the token is not expired.
        if token_timestamp + token_expiration_time >= datetime.now():
            return True
    return False
```

```
def clear_reset_token(user):
    # Build the query to update the reset_token in the user's table
    # Secure Query
    query = "UPDATE users SET reset_token = NULL WHERE username = %s;"
    db_query(query, (user,))
    query = "UPDATE users SET reset_token_timestamp = NULL WHERE username"
    query = "%s;"
    db_query(query, (user,))
```

This ensures that the token properties align with the ASVS requirements.



ASVS (Level 1) [CWE-319] - Communications Security Requirements
(Communication Security)

Verify that secured *TLS* is used for all client connectivity and does not fall back to insecure or unencrypted protocols.

The ASVS requirements were addressed and met while implementing the system for deployment into the production environment. They will be explained further on the documentation in the *Implementation and Deployment* section.



*ASVS (Level 1) [CWE-326] – Communications Security Requirements
(Communication Security)*

Verify using online or up to date *TLS* testing tools that only strong algorithms, ciphers, and protocols are enabled, with the strongest algorithms and ciphers set as preferred.

The ASVS requirements were addressed and met while implementing the system for deployment into the production environment. They will be explained further on the documentation in the Implementation and Deployment section.



ASVS (Level 1) [CWE-350] – Deployed Application Integrity Controls (Malicious Code)

Verify that the application has protection from subdomain takeovers if the application relies upon *DNS* entries or *DNS* subdomains, such as expired domain names, out of date *DNS* pointers or *CNAMEs*, expired projects at public source code repos, or transient cloud *APIs*, serverless functions, or [storage buckets](#) or similar. Protections can include ensuring that *DNS* names used by applications are regularly checked for expiry or change.

The ASVS requirements were addressed and met while implementing the system for deployment into the production environment. They will be explained further on the documentation in the *Implementation and Deployment* section.



ASVS (Level 1) [CWE-400] – File Upload Requirements (Files and Resources)

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

Profile Photo and Product Photo

Before, there wasn't a size check for uploaded photos. Now, by implementing a simple file size check, we can prevent this security issue.

To fix the issue, we added a simple solution to check for the size of the file.

Definition of the functions:

```
@views.route('/update_account/<id>', methods=['POST'])
def update_account(id):

    if id == None and session.get("id") == None:
        return redirect(url_for("views.login"))

    try:

        if os.name == "nt":
            # Get the current working directory
            current_directory =
os.path.dirname(os.path.abspath(__file__)).split("\\handlers")[0]
        else:
            # Get the current working directory
            current_directory =
os.path.dirname(os.path.abspath(__file__)).split("/handlers")[0]

        accounts_directory = os.path.join(current_directory, "database",
"accounts")
        os.makedirs(accounts_directory, exist_ok=True) # Ensure the
directory exists

        file_path = os.path.join(accounts_directory,
f"{id}.png").replace("\\", "/")
```




```
# Get the new uploaded user's account image
profile_photo = request.files.get("profile_photo")

# If there is an image and the size is less than 5MB
if profile_photo and not profile_photo.content_length > 5120 *
5120:

    # Save the image to the user's account
    profile_photo.save(file_path)

# Get the username, email, and password from the user's session
username = request.form.get("username")
email = request.form.get("email")
password = request.form.get("psw")
old_password = request.form.get("psw-old")

# Check if the username field wasn't empty and occupied by another
user
    if username != "" and not check_username_exists(username) and
is_valid_input(username):

        # Update the username
        update_username(id, username)

        # Set the session's username
        session["username"] = username

    else:
        # If there is a problem with the username, get the username
based on the ID
        username = search_user_by_id(id)[1]

# Check if the email field wasn't empty and occupied by another
user
    if email != "" and not check_email_exists(email) and
is_valid_input(email):

        # Update the email
        update_email(id, email)

    else:
```



```
# If there is a problem with the email, get the email based on
the ID

email = search_user_by_id(id)[3]

# Check if the password wasn't empty
if password != "":
    # Update the password
    # Hash the password before storing it in the database

    if bcrypt.check_password_hash(search_user_by_id(id)[2],
old_password):
        hashed_password =
bcrypt.generate_password_hash(password).decode("utf-8")
        username = search_user_by_id(id)[1]
        update_password(username, hashed_password)
    else:
        return render_template("profile.html", message="Invalid
password.", username=username, id=id)

# Return the profile page
return redirect(url_for("views.catalog", id=id))

except Exception as e:
    print(e)
    return render_template("profile.html", message="Invalid input.",
username=username, id=id)
```



```
@views.route('/add_product/<id>', methods=['POST'])
def add_product(id):

    if id == None:
        return redirect(url_for("views.login"))

    if id is not None and is_valid_input(id) is not False:

        product_name = request.form.get("productName")
        product_description = request.form.get("productDescription")
        product_price = request.form.get("productPrice")
        product_category = request.form.get("productCategory")
        product_quantity = request.form.get("productUnits")
        product_photo = request.files.get("productImage")

        preconditions = is_valid_input(product_name) == False or \
            is_valid_input(product_description) == False or \
            is_valid_input(product_price) == False or \
            is_valid_input(product_category) == False or \
            is_valid_input(product_quantity) == False or \
            product_photo and product_photo.content_length >
5120 * 5120 # Verify that the size of the image is less than 5MB

        if preconditions:
            return redirect(url_for("views.catalog", id=id))
        else:
            create_product(product_name, product_description,
product_price, product_category, product_quantity, product_photo)

    return redirect(url_for("views.catalog", id=id))
```



Being the key definitions:

```
# If there is an image and the size is less than 5MB
    if profile_photo and not profile_photo.content_length > 5120 *
5120:
        # Save the image to the user's account
        profile_photo.save(file_path)
```

```
preconditions = is_valid_input(product_name) == False or \
    is_valid_input(product_description) == False or \
    is_valid_input(product_price) == False or \
    is_valid_input(product_category) == False or \
    is_valid_input(product_quantity) == False or \
    product_photo and product_photo.content_length >
5120 * 5120 # Verify that the size of the image is less than 5MB
```

By implementing size restrictions on file uploads, potential denial-of-service issues stemming from uncontrolled file sizes have been effectively mitigated.



ASVS (Level 1) [CWE-918] – SSRF Protection Requirements (Files and Resources)

Verify that the web or application server is configured with an allow list of resources or systems to which the server can send requests or load data/files from.

The ASVS requirements were addressed and met while implementing the system for deployment into the production environment. They will be explained further on the documentation in the *Implementation and Deployment* section.



ASVS (Level 1) [CWE-497] – Unintended Security Disclosure Requirements (Configuration)

Verify that web or application server and application framework debug modes are disabled in production to eliminate debug features, developer consoles, and unintended security disclosures.

The ASVS requirements were addressed and met while implementing the system for deployment into the production environment. They will be explained further on the documentation in the *Implementation and Deployment* section.



ASVS (Level 1) [CWE-548] – Other Access Control Considerations (Access Control)

Verify that directory browsing is disabled unless deliberately desired. Additionally, applications should not allow discovery or disclosure of file or directory metadata, such as *Thumbs.db*, *.DS_Store*, *.git* or *.svn* folders.

The ASVS requirements were addressed and met while implementing the system for deployment into the production environment. They will be explained further on the documentation in the *Implementation and Deployment* section.



ASVS (Level 1) [CWE-598] – Generic Web Service Security Verification Requirements (Web Services)

Verify *API URLs* do not expose sensitive information, such as the *API key*, session tokens etc.

Password Breach Check

When utilizing the *API*, the password is transmitted in a hashed format—specifically, only the first half is shared to safeguard sensitive details.

We employed the “Verify Password View” to ascertain whether a password had been compromised. This verification relied on the `check_password(password)` function contained within the *UserManagement.py* file.

Definition of the View:

```
@views.route('/verify-password', methods=['POST'])
def verify_password():
    data = request.get_json()
    password = data.get('password')

    if not password:
        return jsonify({'error': 'Password not provided'}), 400

    # Check if the password has been breached
    count = check_password(password)

    if count > 0:
        return jsonify({'breached': True, 'count': count})
    else:
        return jsonify({'breached': False})
```




Definition of the function:

```
def check_password(password):  
    # Hash the password using SHA-1  
    hashed_password = hashlib.sha1(password.encode('utf-  
8')).hexdigest().upper()  
    prefix, suffix = hashed_password[:5], hashed_password[5:]  
  
    # Make a GET request to the HIBP API  
    response =  
requests.get(f'https://api.pwnedpasswords.com/range/{prefix}')  
  
    # Check if the suffix of the hashed password exists in the response  
    hashes = (line.split(':') for line in response.text.splitlines())  
    for h, count in hashes:  
        if h == suffix:  
            return int(count)  
  
    return 0
```

This approach ensured that no sensitive data was exposed within the *API's URL* during transmission.



ASVS (Level 1) [CWE-778] – Defenses Against Session Management Exploits (Session Management)

Verify the application ensures a valid login session or requires re-authentication or secondary verification before allowing any sensitive transactions or account modifications.

Login

When implementing the app's updates, including new features and fixes, it became necessary to revalidate the login session to ensure its continued validity.

Definition of the "Login View":

```
# This route is used to perform the login
@views.route('/login', methods=['GET', 'POST'])
def login():
    if request.method == "POST":
        username = request.form.get("username").lower()
        password = request.form.get("password")

        if is_valid_input(username) == False:
            return render_template("login.html", message="Invalid
username.")

        user = search_user_by_username(username)
        id = get_id_by_username(username)

        if user and bcrypt.check_password_hash(user[2], password):
            return redirect(url_for("views.verify_totp_login", id=id))

        # Password is incorrect
        return render_template("login.html", message="Invalid login
credentials.")

    else:
        return render_template("login.html")
```



Definition of the *TOTP* stage:

```
@views.route('/verify_totp_login/<id>', methods=['POST', 'GET'])
def verify_totp_login(id):

    if id == None:
        return redirect(url_for("views.login"))

    if request.method == "GET":
        return render_template("totp_login.html", id=id)
    else:
        username = search_user_by_id(id)[1]
        token = request.get_json().get("token") if request.is_json else
None

        if username is None or token is None:
            return render_template("login.html", message="Invalid TOTP
code. Please try again.")

        verified = verify_totp_code(id, token, 'users', username=username)

        # Verify the TOTP code
        if verified == True:
            # Set session variables
            session["username"] = username
            session["id"] = id
            session["admin"] = get_user_role(session["id"])

            # Check for table existence
            check_database_table_exists(username.lower() + "_cart")
            check_database_table_exists(f"{username.lower()}_orders")

            return jsonify({'message': 'Login successful.'}), 200
        else:

            # Get all the emergency codes
            emergency_codes = get_user_emergency_codes(id)

            # Check if the token is an emergency code
```



```
        if token!=None and token in emergency_codes and  
emergency_codes[token] == True:  
            remove_valid_emergency_code(id, token)  
            # Set session variables  
            session["username"] = username  
            session["id"] = id  
            session["admin"] = get_user_role(session["id"])  
            return jsonify({'message': 'Login successful.'}), 200  
  
        # Return a JSON response indicating failure with status code  
500  
  
        return jsonify({'error': 'Invalid TOTP code. Please try  
again.'}), 500
```

This action ensured the maintenance of a valid login session.

Implementation and Deployment

For this project, we opted for a complete production setup, implementing, and deploying it on an *Apache WSGI* server. Acquiring a registered domain and an *SSL* certificate enabled us to establish a secure *HTTPS* connection. As a result, our *ASVS (Application Security Verification Standard)* checklist presented increased demands, introducing numerous new security standards that needed fulfilment. After achieving full implementation and deployment, we transitioned to analysing the setup. We utilized tools from reputable websites for this analysis.

1. **SSL Labs:** *SSL Labs* is an online service that assesses and analyses the security configuration of *SSL/TLS* implementations on web servers. It evaluates the *SSL/TLS* certificates, server configuration, and protocol support to provide a detailed security report. It helps identify potential vulnerabilities and weaknesses in *SSL/TLS* setups, aiding in improving the overall security posture of websites and online services. The report generated from this website can be found [here](#).
2. **DNS Spy:** *DNS Spy* is a tool that provides insights into *Domain Name System (DNS)* configurations. It allows users to analyse *DNS* records, name servers, and other related information for a given domain. *DNS Spy* helps identify issues, track changes, and ensure the accuracy and security of *DNS* settings. It's useful for monitoring *DNS* health, detecting misconfigurations, and ensuring the reliability of domain-related services on the internet. The report generated from this website can be found [here](#).

In addition to these tools, it was crucial to configure the server accurately. This involved establishing an allow list for necessary resource access, turning off any debug modes, and ensuring directory browsing was disabled.

[Apache App Config](#)



References

[CWE](#)

[CVE DETAILS](#)

[OWASP Application Security Verification Standard | OWASP Foundation](#)

[Have I Been Pwned API](#)

[OWASP ASVS checklist for audits](#)