Security in Informatics and Organizations

Application Security Verification Standard (ASVS)

Daniel Madureira, 107603

Diana Miranda, 107457

Miguel Pinto, 107449

Pedro Ramos, 107348

DETI

University of Aveiro

January 3, 2024

Index

Introduction

The first iteration of this project involved the development of a merchandising website for DETI (Department of Electronics, Telecommunications, and Informatics) related memorabilia, specializing in the sale of stickers, mugs, mousepads, wearables like t-shirts, and pen-drives.

For that iteration we created two full-stack applications with the same functionality and visual appearance, but one being more secure to all the basic attacks that a malicious user might try to exploit, with the intent to break or miss-use system functionalities or even steal client or administrative data.

For this current delivery we were tasked with auditing our own more secure application and identify and fix the eight most critical vulnerabilities or issues that we found, as well as implementing some extra security features to improve the overall safety and protection of our application.

Firstly, we will conduct a moderately in-depth ASVS Security Audit, specifically the OWASP ASVS checklist for audits, and identify high relevance issues from the last project's application.

After this audit, we must choose and select the eight most critical issues and implement fixes for them, explaining in depth why we chose the specific issue and how the final fix was implemented.

After these eight issues are dealt with, we must implement two of the following features:

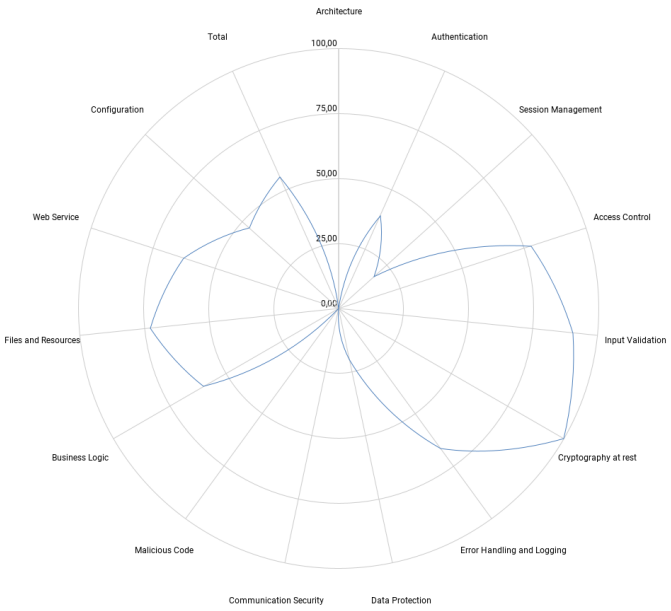
- Password strength requirements;
- Two-Factor authentication using either a OAuth2 system, a TOTP one time password system or a FIDO2 challenge response system;
- Critical data encryption in the Database layer of the original application.

Security Audit and Found Issues

After a comprehensive analysis of all Level 1 issues in the OWASP ASVS checklist for audits on our application, we could use the result to determine which domains of our application's security are in the most need of improvement.

After obtaining the following graph and listing all the most relevant issues we found while conducting the analysis, we determined that the following eight areas were the most critical:

- Better password security and requirements (Authentication);
- Implementing credential recovery (Authentication);
- Out of band verifier requirements for better token implementation (Authentication);
- Better session management and client-side storage (Session Management);
- Better input validation (Input Validation);
- Validate and sanitize untrusted data and HTTP metadata to protect against SSRF attacks (Sanitization and Sandboxing);
- Implement ways for users to export or delete their in-app data (Data Protection);
- Implement more in-depth request header, origin, CORS and anti-CSRF checks to achieve better trust and security in the RESTfull implementation that the application works on (Web Services and Configuration).



After determining which eight areas we needed to work on more, we have identified the following issues as the most critical and with the highest priority for resolution.

Some of these issues will be grouped later on in the project due to the fact that many of them are simply not effective if implemented alone and could be easily implemented together, for example, checking multiple headers of a request can be easily implemented simultaneously and is much more secure than just selecting one header to check, so we will treat them as one issue later on.

The highest priority issues that were not valid in our last iteration are:

Area	#	Description
Password Security Credentials	2.1.1	Verify that user set passwords are at least 12 characters in length (after multiple spaces are combined). (C6)
	2.1.2	Verify that passwords 64 characters or longer are permitted but may be no longer than 128 characters. (C6)
	2.1.7	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. (C6)
	2.1.8	Verify that a password strength meter is provided to help users set a stronger password.
	2.1.9	Verify that there are no password composition rules limiting the type of characters permitted. There should be no requirement for upper or lower case or numbers or special characters. (C6)
Credential Recovery Requirements	2.5.3	Verify password credential recovery does not reveal the current password in any way. (C6)
	2.5.6	Verify forgotten password, and other recovery paths use a secure recovery mechanism, such as time-based OTP (TOTP) or other soft token, mobile push, or another offline recovery mechanism. (C6)
Out of Band Verifier Requirements	2.7.2	Verify that the out of band verifier expires out of band authentication requests, codes, or tokens after 10 minutes.
Fundamental Session Management Requirements	3.1.1	Verify the application never reveals session tokens in URL parameters or error messages.
Input Validation Requirements	5.1.4	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). (C5)

Sanitization and Sandboxing Requirements	5.2.6	Verify that the application protects against SSRF attacks, by validating or sanitizing untrusted data or HTTP file metadata, such as filenames and URL input fields, and uses allow lists of protocols, domains, paths, and ports.
Sensitive Private Data	8.3.2	Verify that users have a method to remove or export their data on demand.
RESTful Web Service Verification Requirements	13.2.3	Verify that RESTful web services that use cookies are protected from Cross-Site Request Forgery via at least one or more of the following: double submit cookie pattern, CSRF nonces, or Origin request header checks.
HTTP Security Headers Requirements	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 an X-Content-Type-Options: nosniff header.
	14.4.5	Verify that a Strict-Transport-Security header is included on all responses and for all subdomains.
	14.4.6	Verify that a suitable "Referrer-Policy" header is included, such as "no-referrer" or "same-origin".
	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.

In conclusion, we determined that these issues were the most critical to fix, and we grouped them according to the table below:

Area	Issues
Better password security and requirements	2.1.1, 2.1.2, 2.1.7, 2.1.8 and 2.1.9
Implementing credential recovery	2.5.3 and 2.5.6
Better token implementation	2.7.2
Better session management and client-side storage	3.1.1
Better input validation	5.1.4
Validate and sanitize untrusted data and HTTP metadata	5.2.6
Implement ways for users to export or delete their in-app data	8.3.2
Implement more in-depth request header, origin, CORS and anti-CSRF checks	13.2.3, 14.4.1, 14.4.3, 14.4.4, 14.4.5, 14.4.6 and 14.4.7

Implementation Steps for the Identified Issues

Better Password Security and Requirements

We have selected five issues in this category because one of the significant vulnerabilities in security systems is related to individuals creating weak passwords. Even when passwords are complex, there is a risk of them being recorded in easily discoverable locations by third parties. Considering this, we deem it crucial to address all these issues to ensure greater robustness in the security of our application.

→ 2.1.1

Initially, our application required a minimum of 8 characters for passwords. Recognizing this as a simple, yet overlooked vulnerability, the resolution was to increase the minimum password character requirement to 12.

→ 2.1.2

In our application, we had not set a maximum character limit, which is not aligned with best security practices. In response to this gap, we implemented a maximum limit of 128 characters. With the update mentioned, passwords in our system are now constrained to a range of 12 to 128 characters, ensuring compliance with recommended security standards.

```
const lengthRegex = /^.{12,128}$/u;

if (!lengthRegex.test(password) || !charRegex.test(password)) {
  setShowAlertPass(true);
} else {
  setShowAlertPass(false);
}
```

→ 2.1.7

Previously, we did not implement any verification to determine whether passwords entered by users, either when creating a new account or updating an existing password, were listed in compromised password databases. To enhance the security of our users' accounts, we introduced this verification.

The verification process operates as follows: first, we generate a SHA-1 hash of the provided password. Subsequently, we convert this byte array into a hexadecimal string. Next, using the prefix of this string, we make a request to the "Have I Been Pwned" API, which returns

all compromised passwords sharing the same prefix. Finally, we check whether the suffix of the provided password is contained in any of the passwords returned by the API. If it is, a notification is sent to the user, alerting them that the password they are attempting to use is compromised, thereby requiring the input of a new password.

```
public boolean check_password(String password) {
  try {
    MessageDigest md = MessageDigest.getInstance("SHA-1");
    byte[] digest = md.digest(password.getBytes());
    StringBuilder sb = new StringBuilder();
    for (byte b : digest) {
        sb.append(String.format("%02x", b));
    }
    String prefix = sb.substring(0, 5);
    String suffix = sb.substring(5);

    HttpClient client = HttpClient.newHttpClient();
    HttpRequest request = HttpRequest.newBuilder()
        .uri(new URI("https://api.pwnedpasswords.com/range/" + prefix))
        .GET()
        .build();
    HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
    return response.body().contains(suffix.toUpperCase());
  } catch (Exception e) {
    e.printStackTrace();
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
  }
}
```

```
if (check_password(newPassword)) {
  throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "The password was found in a data breach! Please choose a different password.");
}
```

→ 2.1.8

Previously, we had not incorporated any "strength" meter to assess the quality of the password entered by the user. In order to assist the user in understanding whether their password is robust and, consequently, more secure. We decided to implement a meter that displays the increasing strength of the password as specific criteria are met. These criteria include the presence of uppercase letters, lowercase letters, numbers, special characters, and a length exceeding 12 characters. It is worth noting that the only mandatory requirement is a minimum of 12 characters.

```
const handlePasswordChange = (event) => {
  setPassword(event.target.value.replace(/\s+/g, ' '));
  const password = event.target.value.replace(/\s+/g, ' ');
  const checks = [
    { regex: /^.{12,128}$/u, points: 1 },
    { regex: /[a-z]/, points: 1 },
    { regex: /[A-Z]/, points: 1 },
    { regex: /[0-9]/, points: 1 },
    { regex: /[\p{S}\p{P}]/u, points: 1 },
    { regex: /[^a-zA-Z0-9]/, points: 1 },
  ];

  let s = 0;
  checks.forEach((check) => {
    if (check.regex.test(password)) {
      s += check.points;
    }
  });

  setScore(s);
};
```

```
{password.length > 0 && score < 3 && (
  <div>
    <span className="text-error">Password is too weak!</span>
    <progress
      className="progress progress-error "
      value={score}
      max="6"
    ></progress>
  </div>
)}
{password.length > 0 && score >= 3 && score < 5 && (
  <div>
    <span className="text-warning">Password is medium!</span>
    <progress
      className="progress progress-warning "
      value={score}
      max="6"
    ></progress>
  </div>
)}
{password.length > 0 && score >= 5 && (
  <div>
    <span className="text-success">Password is strong!</span>
    <progress
      className="progress progress-success "
      value={score}
      max="6"
    ></progress>
  </div>
)}
```

→ 2.1.9

Initially, we had mandatory requirements that included the need for the password to contain at least one uppercase letter, one lowercase letter, one number, and one special character. However, we recognized that this approach posed a vulnerability, as enforcing complex passwords could lead individuals to record them in less secure locations to avoid forgetting them. In light of this, we opted to remove these restrictions, retaining only the requirement that the password have a character count between 12 and 128, as mentioned earlier.

Implementing Credential Recovery

→ 2.5.3

In the app_org, no password recovery method was initially implemented. However, as part of the application restructuring, we decided that this functionality is crucial to help users recover forgotten, stolen or compromised passwords, so now the application offers a password recovery option. Our implementation does not reveal the previous password in any way. Instead, recovery is accomplished through a token sent via email.

→ 2.5.6

Initially, as mentioned in the point above, there was no method implemented for password recovery. Therefore, a secure and effective password recovery feature was introduced to ensure the security of our users' accounts. This feature operates as follows:

Firstly, the user needs to provide the email address associated with the account they wish to recover. This helps us verify the user's identity and ensure that the email corresponds to the existing account on the platform.

After a successful email verification, we generate a temporary token and associate it with the user's account. This token is embedded in a link that is sent to the user via email. The token is unique, random, and has a short lifespan (in this case, 30 minutes) to ensure that if an attacker gets access to this token after it expired, the attacker cannot use it to gain access to the account.

The user receives an email containing a link to reset their password. Upon clicking the link, they are redirected to a page where they can enter a new password. Token validation takes place at this stage. If the token is valid, the user can set a new password.

This implementation ensures that only the legitimate owner of the account can reset their password, making it very difficult for an attacker to take over the account, even if they obtain the user's email. Additionally, this method is convenient for the user as it does not require answering security questions or entering additional information that may be forgotten or compromised.

```
@GetMapping(path = "/forgotPassword")
public @ResponseBody String forgotPassword(@RequestParam String email) {
  App_User user;

  // Check if a User with this login information exists or nor
  try {
    user = app_userRepository.findapp_userByEmail(email);
  } catch (Exception e) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
  }

  if (user == null) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "User email has no account associated!");
  }

  removeExpiredTokens();
  PasswordResetToken token = new PasswordResetToken();
  token.setToken(UUID.randomUUID().toString());
  token.setUser(user);
  token.setExpiryDate(expiryDate:30);
  passwordTokenRepository.save(token);

  mailSender.send(constructResetTokenEmail(token.getToken(), user));

  return "Email sent";
}

private SimpleMailMessage constructResetTokenEmail(String token, App_User user) {
  String url = "http://localhost:5173/resetPassword?token=" + token;
  return constructEmail(subject:"Reset Password", "Dear, " + user.getName() + "\n" +
          "\n" +
          "To reset your password, please click the link below and follow the instructions. Please note that this link is valid for 30 minutes:\n" +
          "\n" +
          url +
          "\r\n" +
          "If you didn't request this reset, please ignore.\n" +
          "\n" +
          "Best regards,\n" +
          "Deti Merch Store" + " \r\n", user);
}

private SimpleMailMessage constructEmail(String subject, String body, App_User user) {
  SimpleMailMessage email = new SimpleMailMessage();
  email.setSubject(subject);
  email.setText(body);
  email.setTo(user.getEmail());
  email.setFrom("detistore.sio@outlook.com");
  return email;
}
```

```java
@PostMapping(path = "/resetPassword")
public @ResponseBody String resetPassword(@RequestParam String token, @RequestParam String password) {

  removeExpiredTokens();
  try {
    PasswordResetToken passToken = passwordTokenRepository.findByToken(token);
    if (passToken == null) {
      throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "Invalid token!");
    }

    App_User user = passToken.getUser();
    if (user == null) {
      throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "Invalid token!");
    }

    if (check_password(password)) {
      throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "The password was found in a data breach! Please choose a different password.");
    }

    Encoder encoder = Base64.getUrlEncoder().withoutPadding();
    KeySpec spec = new PBEKeySpec(password.toCharArray(), user.getSalt().getBytes(), 65536, 128);
    try {
      SecretKeyFactory factory = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA1");
      byte[] hash = factory.generateSecret(spec).getEncoded();
      String newHashedPass = encoder.encodeToString(hash);
      user.setPassword(newHashedPass);
      app_userRepository.save(user);
    } catch (Exception e) {
      throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
    }

  } catch (Exception e) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
  }

  return "Password changed";
}

public void removeExpiredTokens() {
  Date now = new Date();
  List<PasswordResetToken> expiredTokens = passwordTokenRepository.findAllByExpiryDateBefore(now);
  passwordTokenRepository.deleteAll(expiredTokens);
}
```

Better Token Implementation

→ 2.7.2

In the previous iteration, we created a token system for authenticating the user without the need for storing credentials in the user-side application or having to ask the user for authentication every time he requests an endpoint that requires an active account to access.

One of the biggest issues with our implementation was that tokens were not set to expire, that is, the token would live until a new one was generated for the same account, creating a very dangerous environment where once an attacker gained access to a user's token, he could use it indefinitely without the need for further authentication.

To resolve this, all tokens generated by our application are set to expire after 10 minutes.

The user login can be refreshed by requesting a new token while the old one is still active, giving the user 10 more minutes until the new token expires.

If the token expires, the user will always be asked to login again.

```java
@GetMapping(path = "/reloadToken")
public @ResponseBody String reloadToken(@RequestParam String email, @RequestParam String oldToken) {
  App_User user;

  //  Check if a User with this login information exists or nor
  try {
    user = app_userRepository.findapp_userByEmail(email);
  } catch (Exception e) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, reason: "Internal processing error!");
  }

  if (user == null) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, reason: "User email has no account associated!");
  }

  if (!user.getActive_Token().equals(oldToken)) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, reason: "Token does not match the given user ID!");
  }

  if (user.getToken_Expiration() < (System.currentTimeMillis() / 1000)) {
    throw new ResponseStatusException(HttpStatus.UNAUTHORIZED, reason: "The provided token has expired! Please log in again.");
  }

  //  Generate the token
  Encoder encoder = Base64.getUrlEncoder().withoutPadding();
  SecureRandom rng = new SecureRandom();
  byte bytes[] = new byte[64];
  rng.nextBytes(bytes);
  String token = encoder.encodeToString(bytes);

  user.setActive_Token(token);
  //  Set token to expire after 10 minutes
  user.setToken_Expiration((int)(System.currentTimeMillis() / 1000) + 600);
  app_userRepository.save(user);

  //  Generate the output object for the frontend
  JSONObject out = new JSONObject();
  out.put(name: "new_token", token);

  return out.toString(indentSpaces: 1);
}
```

Better Session Management and Client-side Storage

→ 3.1.1

Another of the biggest identified problems with our system was with the easy spoofing and modification of internal system messages.

Our frontend communicates with the backend via a set of RESTful HTTP requests, where the data was sent via the URL parameters of the request.

Initially this was done as most parameters were unimportant IDs and non-critical data, but after adding other functionalities, soon passwords and tokens were being sent inside the same parameters, making the task of analyzing, stealing and interfering with system requests very easy.

To remedy this, we moved all critical variables (emails, passwords, tokens and credit information) from the URL string to the body of the request, providing an extra layer of protection against man-in-the-middle attacks and other similar exploits harder to conduct.

This was done by replacing the “RequestParam” in the backend to “ResponseBody”, which maps the information to a JSON like string, which is then converted by Spring to a Map

object. The fix also had to be applied to the front end of the application, to change where the request data was being stored.

```
@GetMapping(path = "/checkLogin", consumes = MediaType.MULTIPART_FORM_DATA_VALUE)
public @ResponseBody String checkLoginInfo(@RequestPart Map<String, String> json) {

  String email = json.get("email");
  String password = json.get("password");

  App_User user;
```

Better Input Validation

→ 5.1.4

In order to avoid typing errors or misunderstandings, we conducted a comprehensive review of all input checks and requirements, particularly on the checkout page where these checks were most lacking. The verification process for most fields was adjusted in an equivalent manner, with the exception of the validation process for address-related fields, which is more complex and was addressed as follows:

When filling in all the fields on the checkout page and clicking the button to place the order, a thorough check of the fields is performed to ensure they meet the requirements. Specifically, we conduct a detailed analysis of the data related to the delivery address.

Upon clicking to complete the order, a request is sent to the Google API (geocode), where the postal code, city, address, and country entered by the user are provided as parameters. If the Google API confirms that all these data are valid and that the address exists, the address-related data is marked as valid in the checkout process.

This approach is essential to prevent incorrect data, as it is common for users to make inadvertent typing errors.

```
const handleChangeCVC = (e) => {
  const cvcRegex = /^\d{3}$/;
  if (e.target.value.length < 1 || !cvcRegex.test(e.target.value)) {
    setCvcAlert(true);
  } else {
    setNewCard({
      ...newCard,
      cvv: e.target.value,
    });
    setCvcAlert(false);
  }
};

const handleChangeCardName = (e) => {
  const cardNameRegex = /^[A-Za-z\s]+$/;
  if (e.target.value.length < 1 || !cardNameRegex.test(e.target.value)) {
    setCardNameAlert(true);
  } else {
    setNewCard({
      ...newCard,
      card_name: e.target.value,
    });
    setCardNameAlert(false);
  }
};

const handleChangeCardNumber = (e) => {
  const cardNumberRegex = /^\d{16}$/;
  if (e.target.value.length < 1 || !cardNumberRegex.test(e.target.value)) {
    setCardNumberAlert(true);
  } else {
    setNewCard({
      ...newCard,
      card_number: e.target.value,
    });
    setCardNumberAlert(false);
  }
};
```

```
const handleChangeExpirationDate = (date) => {
  const expirationDateRegex = /^\d{2}\/\d{4}$/;

  if (
    date.format('MM/YYYY').length < 1 ||
    !expirationDateRegex.test(date.format('MM/YYYY'))
  ) {
    setExpirationDateAlert(true);
  } else {
    const [month, year] = date.format('MM/YYYY').split('/');

    if (parseInt(month) > 12) {
      setExpirationDateAlert(true);
    } else {
      const inputDate = new Date(year, month - 1);
      const currentDate = new Date();

      if (inputDate > currentDate) {
        setNewCard({
          ...newCard,
          expiration_date: date.format('MM/YYYY'),
        });
        setExpirationDateAlert(false);
      } else {
        setExpirationDateAlert(true);
      }
    }
  }
};
```

```
const handleDeliveryDay = (date) => {
  const dateRegex = /^\d{2}\/\d{2}\/\d{4}$/;
  if (
    date.format('DD/MM/YYYY').length < 1 ||
    !dateRegex.test(date.format('DD/MM/YYYY'))
  ) {
    setDelivery_dayAlert(true);
  } else {
    setForm({
      ...form,
      delivery_day: date.format('DD/MM/YYYY'),
    });
    setDelivery_dayAlert(false);
  }
};

const handleDeliveryTime = (date) => {
  const timeRegex = /^([01]\d|2[0-3]):([0-5]\d)$/;

  if (date.format('HH:mm') < 1 || !timeRegex.test(date.format('HH:mm'))) {
    setDelivery_timeAlert(true);
  } else {
    setForm({
      ...form,
      delivery_time: date.format('HH:mm'),
    });
    setDelivery_timeAlert(false);
  }
};

const handleAddress = (e) => {
  const addressRegex = /^[\s\S]*$/u;
  const poBoxRegex = /^PO Box \d+$/;

  if (
    e.target.value.length < 1 ||
    !addressRegex.test(e.target.value) ||
    poBoxRegex.test(e.target.value)
  ) {
    setAddressAlert(true);
  } else {
    setForm({
      ...form,
      address: e.target.value,
    });
    setAddressAlert(false);
  }
};

const handleAddress2 = (e) => {
  const addressPartRegex = /^[A-Za-z0-9'\.\-\s\,]+$/;

  if (e.target.value.length < 1 || !addressPartRegex.test(e.target.value)) {
    setAddress2Alert(true);
  } else {
    setForm({
      ...form,
      address2: e.target.value,
    });
    setAddress2Alert(false);
  }
```

```
const verifyLocation = async () => {
  let zc = form.zip_code;
  if (form.zip_code.includes('-')) {
    zc = form.zip_code.split('-')[0];
  }
  const cit = form.city;
  const c = form.country;
  const add = form.address;
  const address = `${zc} ${cit} ${add} ${c}`;
  const apiKey = 'AIzaSyBFkLDRwb7tal8NXKkU397FDRFQFtXTaM0';

  let control = [];
  let valid = true;
  try {
    const response = await fetch(
      `https://maps.googleapis.com/maps/api/geocode/json?address=${address}&key=${apiKey}`
    );
    const data = await response.json();

    if (data.status === 'OK' && data.results.length > 0) {
      for (let i = 0; i < data.results[0].address_components.length; i++) {
        if (
          data.results[0].address_components[i].types.includes('postal_code')
        ) {
          const postalCode = data.results[0].address_components[i].long_name;
          if (postalCode.includes('-')) {
            if (postalCode.split('-')[0] != zc) {
              valid = false;
            }
          } else
          {
            if (postalCode != zc) {
              valid = false;
            }
          }
        }
        if (data.results[0].address_components[i].types.includes('country')) {
          if (
            data.results[0].address_components[i].long_name.toLowerCase() !=
            c.toLowerCase()
          ) {
            valid = false;
          }
        }
        if (
          data.results[0].address_components[i].types.includes('locality')
        ) {
          if (
            data.results[0].address_components[i].long_name.toLowerCase() !=
            cit.toLowerCase()
          ) {
            valid = false;
          }
        }
        control.push(
          ...data.results[0].address_components[i].types.map((type) => type)
        );
      }

      if (
        !control.includes('postal_code') ||
        !control.includes('country') ||
        !control.includes('locality')
      ) {
        valid = false;
      }
      setAddressValidAlert(!valid);
    } else {
      console.error('Geocoding failed:', data.status);
    }
  } catch (error) {
    console.error('Error:', error);
  }
};
```

```
const handleCity = (e) => {
  const cityRegex = /^[\s\S]*$/u;

  if (e.target.value.length < 1 || !cityRegex.test(e.target.value)) {
    setCityAlert(true);
  } else {
    setForm({
      ...form,
      city: e.target.value,
    });
    setCityAlert(false);
  }
};

const handleCountry = (event, value) => {
  const countryRegex = /^[\s\S]*$/u;
  if (value.length < 1 || !countryRegex.test(value)) {
    setCountryAlert(true);
  } else {
    setForm({
      ...form,
      country: value,
    });
    setCountryAlert(false);
  }
};

const handleZipCode = (e) => {
  const postalCodeRegex = /^\d+-?\d+$/;

  if (e.target.value.length < 1 || !postalCodeRegex.test(e.target.value)) {
    setZip_codeAlert(true);
  } else {
    setForm({
      ...form,
      zip_code: e.target.value,
    });
    setZip_codeAlert(false);
  }
};
```

Validate and sanitize untrusted data and HTTP metadata

→ 5.2.6

Initially, there was no verification of HTTP file metadata, allowing the insertion of an image link instead of a real image file. To address this issue and safeguard our application against SSRF attacks, additional checks have been added to the pages where image insertion is allowed (RegisterUserPage.jsx and StorePage.jsx). Now, in addition to the existing checks on the inserted file, an extra verification has been implemented to ensure that the file type begins with "image/". This ensures that only actual files are sent to the backend, preventing the possibility of submitting links posing as images.

```
const handleImageChange = (event) => {
  const file = event.target.files[0];

  setImage(event.target.files[0]);
  const imageRegex = /\.(jpe?g|tiff?|png|webp)$/i;
  if (
    file < 5000000 ||
    imageRegex.test(image.name) ||
    file.type.startsWith('image/')
  ) {
    setImage(file);
    setShowAlertImage(false);
  } else {
    setShowAlertImage(true);
  }
};
```

Implement ways for users to export and delete their in-app data

→ 8.3.2

With the growing fixation of the average user to reclaim their data from websites or services they no longer trust or use, we decided to create a way for said users to delete their account or export the data that was shared with our application, which was not previously available (either option can be accessed in the user page after login in).

Deletion of user data

This was handled in a way that deletes all the critical user data and marks the account as deleted, without removing the user itself from the database. This approach enhances scalability, as it is less performance demanding and ensures that referential integrity is maintained within the relational database.

```
const handleDeleteAccount = async () => {
  try {
    const formData = new FormData();
    formData.append('id', id);
    formData.append('token', token);

    const response = await axios.put(
      '/user/deleteUserData',
      formData
    );
    if (response.status === 200) {
      handleLogout();
    }
  } catch (error) {
    console.error('Error during API call', error);
  }
};
```

```java
//Mark user for deletion by ID
@Transactional
@PutMapping(path = "/deleteUserData")
public @ResponseBody String deleteUserData(@RequestParam Integer id, @RequestParam String token) {
  App_User usr;

  // Check if a User with this ID exists
  try {
    usr = app_userRepository.findapp_userByID(id);
  } catch (Exception e) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
  }

  if (usr == null) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "The specified User does not exist!");
  }

  if (!usr.getActive_Token().equals(token)) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "Token does not match the given user ID!");
  }

  // Mark user data as deleted
  try {
    usr.setSalt(salt:"");
    usr.setName(name:"");
    usr.setEmail(email:"");
    usr.setPassword(password:"");
    usr.setActive_Token(active_Token:"");
    usr.setDeleted(deleted:true);
    app_userRepository.save(usr);

  } catch (Exception e) {
    e.printStackTrace();
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
  }

  return "SUCCESS: User data successfully marked as deleted!";
}
```

Exportation of user data

In the system, making a request to export user data will generate a pdf in the Spring backend with the help of the dependency “itextpdf” containing an overall view of user data, such as name, email and purchases done.

Once this pdf document has been created, it will be sent to the frontend where it will automatically download and open.

```java
// Export all user data to a file
@GetMapping(path = "/exportUserData", produces = MediaType.APPLICATION_PDF_VALUE)
public @ResponseBody ResponseEntity<byte[]> exportUserData (@RequestParam Integer id, @RequestParam String token) {
  App_User usr;

  // Check if a User with this ID exists
  try {
    usr = app_userRepository.findapp_userByID(id);
  } catch (Exception e) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Internal processing error!");
  }

  if (usr == null) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "The specified User does not exist!");
  }

  if (!usr.getActive_Token().equals(token)) {
    throw new ResponseStatusException(HttpStatus.UNPROCESSABLE_ENTITY, "Token does not match the given user ID!");
  }

  // Generate the output user object for the frontend
  Document doc = new Document();
  ByteArrayOutputStream out = new ByteArrayOutputStream();

  try {
    PdfWriter.getInstance(doc, out);
    doc.open();

    //Add all user data to the PDF
    Paragraph title = new Paragraph("------------ My DETI Store Data ----------");
    title.setAlignment(Element.ALIGN_CENTER);
    doc.add(title);
    doc.add(new Paragraph("\nGenerated on: [" + new Date().toString() + "]\n"));
    doc.add(new Paragraph("\n\n------------ Account Data ----------"));
    doc.add(new Paragraph("\nUsername: " + usr.getName()));
    doc.add(new Paragraph("\nEmail: " + usr.getEmail()));
    doc.add(new Paragraph("\n\n------------ Shopping Cart ----------"));
    doc.add(new Paragraph("\n" + usr.printCart()));
    doc.add(new Paragraph("\n\n------------ Request History ----------"));
    doc.add(new Paragraph("\n" + usr.printRequestHistory()));
    doc.close();
  } catch (Exception e) {
    throw new ResponseStatusException(HttpStatus.INTERNAL_SERVER_ERROR, "Error while generating PDF");
  }

  byte[] pdfBytes = out.toByteArray();
```

```java
  // Set the response headers
  HttpHeaders headers = new HttpHeaders();
  headers.setContentType(MediaType.APPLICATION_PDF);
  headers.setContentDispositionFormData("inline", "my_deti_store_data.pdf");

  return ResponseEntity
          .ok()
          .headers(headers)
          .body(pdfBytes);
  }
```

```javascript
const handleExportData = async () => {
  try {
    const response = await axios.get(
      '/user/exportUserData',
      {
        params: {
          id: user.id,
          token: item.token,
        },
        responseType: 'arraybuffer',
      }
    );

    if (response.status === 200) {
      // Create a Blob from the PDF data
      const blob = new Blob([response.data], { type: 'application/pdf' });

      // Create a download link and click it
      const url = window.URL.createObjectURL(blob);
      const a = document.createElement('a');
      a.href = url;
      a.download = 'my_deti_store_data.pdf';
      document.body.appendChild(a);
      a.click();
      document.body.removeChild(a);
      window.URL.revokeObjectURL(url);
    }
  } catch (error) {
    console.error('Error during API call', error);
  }
};
```

Implement more in-depth request header, origin, CORS and anti-CSRF checks

→ 13.2.3

Cross-Site Request Forgery, or CSRF for short, is a type of attack that occurs when a victim is unknowingly manipulated into sending malicious requests to an application where the system trusts the victim, allowing the attacker to make requests using the victim's privileges and information, causing the application to log the victim as responsible for the requests.

One way of mitigating this is by implementing simple yet effective Origin request Header checks, which were done in the backend with the use of the Spring Security library.

The default spring security implementation adds this functionality by default, but for extra security we also implemented this feature inside the CORS configuration.

```
@Configuration
@EnableWebSecurity
public class WebSecurityConfig {

  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
    http
    .headers((headers) ->
                headers
```

```
configuration.setAllowedHeaders(Arrays.asList("X-Requested-With", "Origin", "Content-Type", "Accept", "Authorization"));
```

→ 14.4.1, 3, 4, 5, 6 and 7

One of the biggest issues we identified with our application was the low trust involved in the internal requests.

Since the backend never checked the authenticity of requests, these could be easily forged by an untrusted entity without even alerting the backend, granting full functionality to any actor that spent some time testing the various inputs and requests that our application provides.

To mitigate this massive issue, we implemented two features, and interceptor that catches the requests and briefly checks the Method of the request and if the Content-Type header is defined (issue 14.4.1, which although was marked as "Valid" in the ASVS assessment could still use with more checks).

After this we once again used the Spring Security library to implement more request and header checks, such as checking the Content Security Policy header (issue 14.4.3), verifying the

X-Content-Type-Options: nosniff header (issue 14.4.4), verifying the Strict-Transport-Security header (issue 14.4.5), validating the Referrer-Policy header as “no-refferer“ (issue 14.4.6) and finally by checking the Content-Security-Policy frame-ancestors and response headers.

Some of these are implemented into the default security measures of the Spring Security library.

```java
@Override
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(interceptor).addPathPatterns(...patterns: "/**");
}

@Override
public void addCorsMappings(CorsRegistry registry) {
    registry.addMapping(pathPattern: "/**")
            .allowedMethods(...methods: "GET", "POST", "PUT")
            .allowedHeaders(...headers: "Content-Type");
}
```

```java
@Bean
public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
  http
  .headers((headers) ->
            headers
                .contentTypeOptions(withDefaults()) //  X-Content-Type-Options: nosniff
                .xssProtection(withDefaults())      //  X-XSS-Protection header
                .cacheControl(withDefaults())       //  Cache-Control: no-cache
                .httpStrictTransportSecurity(withDefaults())  //  Strict Transport Security headers
                .frameOptions(withDefaults())
                .contentSecurityPolicy(cps -> cps.policyDirectives(policyDirectives: "script-src 'self' ..."))
                .referrerPolicy(referrerPolicy -> referrerPolicy.policy(ReferrerPolicyHeaderWriter.ReferrerPolicy.NO_REFERRER))
        )
  .cors(withDefaults())
  .csrf((csrf) -> {
    csrf.ignoringRequestMatchers(new AntPathRequestMatcher(pattern: "/user/*"));
    csrf.ignoringRequestMatchers(new AntPathRequestMatcher(pattern: "/product/*"));
  })
  .authorizeHttpRequests((requests) -> {
    /*requests.anyRequest().authenticated()*/
    requests.requestMatchers(new AntPathRequestMatcher(pattern: "/user/**")).permitAll();
    requests.requestMatchers(new AntPathRequestMatcher(pattern: "/product/**")).permitAll();
    requests.anyRequest().permitAll();
  });
  return http.build();
}
```

Additional Features

After the ASVS analysis and corresponding implementation fixes for the most prominent found problems, we were tasked with choosing and implementing two extra features from a total of three different options:

- Implement a password strength evaluation according to the ASVS requirements V2.1 with breach verification using an external service;
- Implement multi-factor authentication required for users to gain access to the Web application using either:
    - OAuth 2.0 + OIDC login;
    - TOTP authentication login;
    - FIDO/FIDO2 authentication login.
- Encrypt the database storage and cypher critical data.

Since we incorporated most of the first feature (password strength evaluation and breach detection) in the fixes for the issues found on the ASVS evaluation, and since out database (H2) can be encrypted with a chosen cypher by only specifying so on the connection string, we decided to implement all three of the additional features.

→ Password strength evaluation and breach verification

Password strength assessment and breach verification are crucial elements in an application, as highlighted in the "Better Password Security and Requirements" section. This is due to the significant vulnerability of applications to weak passwords or poor choices made by users.

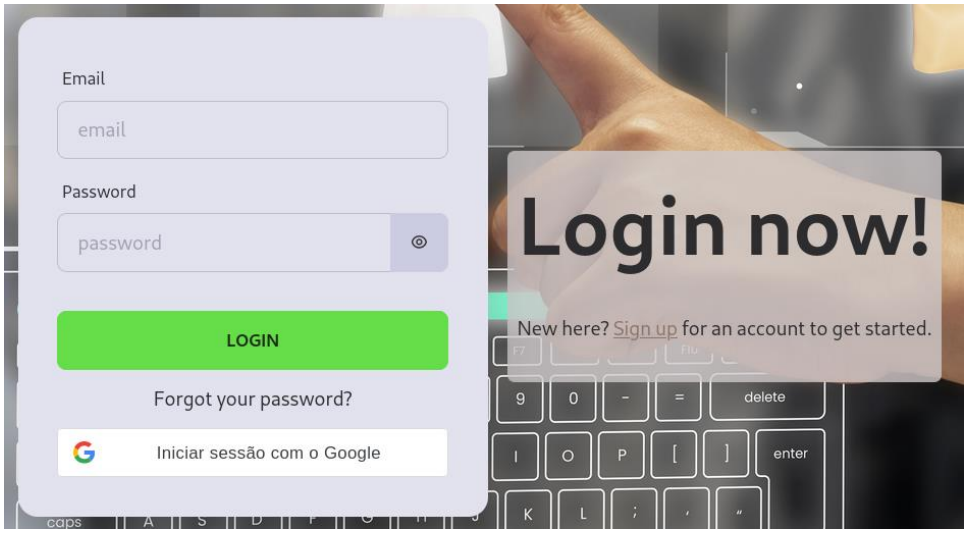
Given that parameters 2.1.3, 2.1.4, 2.1.5, 2.1.6, 2.1.10, 2.1.11, and 2.1.12 were already implemented and validated in our initial application, we proceeded to implement the remaining parameters (2.1.1, 2.1.2, 2.1.7, 2.1.8, 2.1.9) to meet the minimum password security requirements outlined in version 2.1. All these implementations are thoroughly explained in the "Better Password Security and Requirements" section.

→ Multi-factor authentication

Multi-factor authentication provides another layer of security to any system, requiring users to authenticate their identity through multiple independent factors.

We chose to implement two-factor authentication using an OAuth 2.0 + OIDC login strategy. We used Google Identity Services to implement two-factor authentication and integrate it seamlessly into our app.

With Google Identity services users can log into our web app by their Google Account, where Google initiates the authentication process by redirecting the user to Google Identity Services (GIS) where they can enter their credentials.



After the user is authenticated on Google Identity Services, GIS issues an authorization grant to our web app where it returns a JWT token which contains the user's relevant info and authorization status.

With GIS we can ensure that the user is fully authenticated, and it can securely access our app, since it passed through Google Identity authentication mechanisms, which may include SMS, email or one tap confirmations to authenticate the user throughout the GIS process.

→ Database storage encryption

Database encryption is one of the most important security aspects of any service implementation, it makes sure that critical data is protected in the event of information leaks or access by unauthorized users via backdoors or faulty applications/services that have been exploited.

In our case, since we are using the H2 Database system, the entire contents of the database are stored inside a singular file that can be encrypted automatically by the database management system itself.

This means that all that was needed to be added to the application itself was a new connection string to the database that includes a new “CIPHER” parameter and the database new requires two passwords to access, one for the user of the database and another for decrypting and accessing the database itself, which is referred to as the “File Key”.

Here the file password is the first word before the space, whereas the user password is the one after:

```
#  Database Connection                          Choosen Cipher
spring.datasource.url= jdbc:h2:mem:sio_db_proj2;CIPHER=AES;DB_CLOSE_DELAY=-1;NON_KEYWORDS=KEY,VALUE
#  Database User
spring.datasource.username= Spring_User
#  File Password + User Password
spring.datasource.password= S^yVgQGpzM%f8LJ) pUx8Y2*Ey5&#Wc?v
```

The H2 database supports tree cipher modes, AES-128 (also known as Rijndael), XTEA-32 and FOG. We opted for the AES-128 since it is more advanced than XTEA-32 and we did not choose FOG as it is only a pseudo-cipher used for hiding information from regular text editors.

Conclusion

To briefly conclude this project, we conducted a ASVS level 1 security assessment for our previous application utilizing a set ASVS checklist and then implemented features and security checks for the eight most critical areas of our application, fixing a total of 19 security issues.

We then implemented three extra features for our application to better prevent hostile attacks against our system and improve system reliability, security, and client trust.

With the ASVS checklist we found many overlooked or obscure issues that although mostly quick to fix, could easily lead to attackers exploiting the application and its users.

These issues may not seem complex at first, but they quickly amount to many possible unexpected entry points or exploits that could be used to disrupt or steal data from our system.

Although the previous iteration's secure application was much more secure than the standard one, in the iteration we discovered that much more work needs to be done to provide a fully functional system that is also protected from these problems without compromising on user experience.