



UNIVERSITÀ DI PISA

Master Degree in Computer Engineering

System and Network Hacking Security Report

Secure Book Selling Website

Team Members:

Francesco Martoccia

Luca Tartaglia

Salvatore Lombardi

Academic Year 2022/2023

Contents

1	Broken Authentication	2
1.1	Weak Password Check	2
1.1.1	Mitigation	2
1.2	Multiple Login Attempts	3
1.2.1	Mitigation	3
1.3	Session Hijacking and Session Fixation	3
1.3.1	Mitigation	3
1.4	Credential Storage and Management	4
1.4.1	Password Hashing	4
1.4.2	One-Time Password (OTP) Hashing	4
1.5	Password Change and Account Recovery	5
1.5.1	Password Change	5
1.5.2	Account Recovery	5
2	Insecure Communication	7
2.1	Mitigation	7
3	SQL Injection	9
3.1	Mitigation	9
4	Cross Site Scripting XSS	10
4.1	XSS Attacks	10
4.2	XSS Mitigations	10
5	Cross Site Request Forgery XSRF	12
5.1	XSRF Attacks	12
5.2	XSRF Mitigation	12
6	Multi-Step Procedure	13
6.1	Multi-Step Procedure Vulnerability	13
6.2	Multi-Step Checkout Workflow	13
6.3	Multi-Step Checkout Security Control Flow	16
6.3.1	Security Functions Implemented	17
6.4	Additional Security: Credit Card Info Encryption and Decryption	18

1 - Broken Authentication

1.1 Weak Password Check

Login bruteforcing, also known as credential stuffing, is a prevalent attack where an attacker utilizes a database of common or stolen passwords and systematically tries them against a web application using automated tools such as Hydra. This attack leverages publicly available password lists and can also include permutations or variations of dictionary words. To counteract such attacks, the most secure solution is **multi-factor authentication (MFA)**, which verifies user identity based on multiple factors, such as a password and a physical token. However, due to the high cost and complexity of implementing MFA, it is not always feasible for every application.

1.1.1 Mitigation

The first and most effective **countermeasure** against login bruteforcing is enforcing **strong password policies**. Passwords should meet certain criteria to ensure they are difficult to guess or crack. These criteria typically include:

- A **minimum length** requirement
- Inclusion of **uppercase** and **lowercase** letters
- Use of **numeric characters**
- Incorporation of **special characters**
- **Avoidance** of **common words**, names, and dictionary entries

To assist users in creating strong passwords, our application employs **password strength estimation**, which evaluates the quality of a password and provides feedback on its weaknesses. The password strength estimator used is **zxcvbn**.

In the application, passwords are checked by the ***checkPasswordStrength*** method. It first verifies the password against a regular expression criteria by using the ***regexControl*** function, that ensures the password meets the defined structure, including a minimum length of 9 characters, the inclusion of uppercase and lowercase letters, numbers, and special characters. Then, it calls the ***zxcvbnControl*** function, in order to assess the password's strength by considering its complexity and the likelihood of it being guessed (based on common patterns and previously exposed passwords).

1.2 Multiple Login Attempts

Account locking is a security mechanism designed to protect user accounts from unauthorised access. When multiple failed login attempts are detected, the system temporarily locks the account to prevent brute force attacks.

1.2.1 Mitigation

Here's a step-by-step breakdown of how account locking is implemented in our application:

- The system tracks the number of **failed login attempts** for each user within a specific **time window**.
- If the number of failed attempts exceeds the allowed limit, the system initiates an account lock. The user cannot perform a login attempt for an initial period of 30 seconds if the account has not been blocked before. If the account has been previously blocked, and the user fails the log in attempt that is given him after 30 seconds, the system adds 30 minutes to the block time.
- By gradually increasing the block time, the system ensures that users are given multiple chances to remember their password while still protecting against brute force attacks.
- Upon successful login, the system resets the failed attempts counter and block time to zero. This prevents locking out users who eventually remember their correct password.

1.3 Session Hijacking and Session Fixation

If an attacker steals somehow the token of an ongoing session of a legitimate user, then he/she can inject requests on the same session, thus impersonating the user. This is known as **Session Hijacking**.

A **Session Fixation** attack is a type of security exploit where an attacker forces a user's session identifier (session ID) to a known value. This allows the attacker to hijack the user's session and potentially gain unauthorised access to the user's account or sensitive information.

1.3.1 Mitigation

- The use of **HTTPS** ensures that all sensitive communications, including session cookies, are transmitted over HTTPS. This encrypts data in transit, making it difficult for attackers to intercept and steal session tokens.
- Implementing **session regeneration** immediately after successful user authentication or logout ensures that a new session ID (SID) is generated, effectively nullifying any previously known or fixed session IDs.
- When a user logs out, all data related to their session is deleted.

- A timestamp associated with the last interaction of a specific user is maintained. When the elapsed time between the user's last interaction and the current instant is greater than the value relative to the life-cycle of a session, the session related to that user is invalidated and all data associated with it is deleted. This prevents security risks associated with prolonged session lifetimes.

1.4 Credential Storage and Management

Storing **credentials** in an insecure manner can lead to significant security vulnerabilities. When credentials are not properly protected, attackers can gain **unauthorized access** to sensitive user data, resulting in data breaches and compromised accounts. Common issues include storing passwords in **plaintext**, using **weak hashing** algorithms, or failing to implement **salting**. These practices can make it easier for attackers to exploit stolen data and perform credential stuffing or brute force attacks. To mitigate these risks, our application implements robust credential storage mechanisms.

1.4.1 Password Hashing

Our application uses the **BCrypt algorithm** for **hashing passwords**, leveraging the PHP ***password_hash*** function with the `PASSWORD_DEFAULT` option. BCrypt is a strong hashing algorithm designed to be computationally expensive, which helps protect against brute force attacks by slowing down the hashing process.

- **Password Hashing:** The ***password_hash*** function generates a hash for a given password, using the `PASSWORD_DEFAULT` constant. This constant is designed to change over time as new and stronger algorithms are added to PHP. Consequently, the length of the resulting hash may vary, so it is stored in a database column that can expand beyond 60 characters. We use a column with a length of 255 characters to accommodate any future changes.
- **Salting:** BCrypt automatically includes a **salt** within the hashed password. Salting adds random data to the password before hashing, ensuring that even identical passwords will result in different hashes. This prevents attackers from using precomputed tables (**rainbow tables**) to crack the hashes.

1.4.2 One-Time Password (OTP) Hashing

For **one-time passwords** (OTPs), where speed is a critical requirement, we use the **SHA-256** algorithm with a **salt**. The PHP ***hash*** function is employed for this purpose. **SHA-256** is a cryptographic hash function that provides a good balance between **security** and **performance**.

- **Hashing OTPs:** The ***hash*** function generates a **SHA-256** hash for the OTP combined with a salt. This approach enhances security while maintaining the necessary performance.
- **Salting:** Salting is applied to OTPs to ensure that the hash values are resistant to precomputed tables attacks.

1.5 Password Change and Account Recovery

1.5.1 Password Change

Password change functionality is critical for maintaining account security. Common issues include not validating the current password, not enforcing password strength requirements, and not verifying the new password correctly.

To mitigate these risks, our application enforces several key security measures for password changes:

- **Current Password Verification:** The current password must be entered correctly before a new password can be set. This helps protect against unauthorized password changes resulting from other security flaws.
- **Password Strength Controls:** The same password strength requirements applied during user registration are enforced during password changes. This ensures that new passwords are sufficiently strong and resilient against attacks.
- **Password Re-entry:** Users are required to enter the new password twice to confirm it, reducing the likelihood of errors.
- **No Forced Periodic Changes:** Our application does not enforce periodic password changes, as studies have shown that such policies often lead to weaker passwords and predictable patterns, contrary to their intended purpose.

The password change functionality is implemented in *passwordChange.php*. Users are prompted to enter their current password and the new password (twice). This approach ensures that unauthorized access to accounts is minimized and that users maintain strong passwords.

1.5.2 Account Recovery

Account recovery processes are another critical area where broken authentication can pose significant risks. If an attacker can exploit weaknesses in the recovery process, they can gain control over user accounts. Common issues include inadequate verification of the user's identity and insecure handling of recovery tokens. To ensure secure account recovery, our application implements the following measures:

- **OTP (One-Time Password) for Recovery:** When a user requests password recovery, an OTP is generated and sent via email. This OTP must be entered correctly within a specified time frame (2 minutes) to proceed with the password reset.
- **OTP Generation and Validation:** The random OTP is generated and combined with a salt before being hashed with SHA-256. The hashed OTP is saved on the DB, in order to perform the validation later. The OTP has a limited validity period, enhancing security.
- **New Password Setting:** Once the OTP is validated, users can set a new password. The same strength requirements apply as during initial registration and password changes.

The account recovery process involves two main components: ***otpRequest.php*** and ***passwordRecovery.php***. Users first request an OTP by entering their email address. The system verifies the request, generates an OTP, and sends it to the user's email. The user must then enter the OTP and their new password to complete the recovery process. This ensures that only the legitimate account owner can reset the password.

2 - Insecure Communication

Insecure communication channels can pose significant security risks to web applications. When data is transmitted over insecure channels, it is susceptible to different vulnerabilities, leading to severe consequences, including the theft of sensitive information, such as usernames, passwords, and financial data. Additionally, attackers can manipulate data in transit, causing unauthorized actions or compromising the integrity of the communication.

Common issues with insecure communication include:

- **Eavesdropping:** Attackers can intercept and read data transmitted over unencrypted channels, gaining access to sensitive information.
- **Man-in-the-Middle (MitM) Attacks:** Attackers can insert themselves between the communicating parties, intercepting and potentially altering the data being exchanged.
- **Data Integrity Compromise:** Without proper encryption, data can be modified during transmission, leading to data corruption or unauthorized actions.
- **Credential Theft:** Sensitive information such as login credentials can be easily stolen when transmitted over insecure channels.

2.1 Mitigation

To mitigate these risks, our application uses HTTPS (HyperText Transfer Protocol Secure) to ensure that all data transmitted between the client and server is encrypted. HTTPS employs SSL/TLS protocols to provide a secure communication channel, protecting the data from interception and tampering.

The implementation of HTTPS for our application involves several key steps, including the creation and installation of a Certificate Authority (CA) certificate and the generation of a server certificate.

Below is an overview of the process used to set up HTTPS for our domain, by using the *gen_certificates.sh* script:

Certificate Authority (CA) Setup

- **Generate the CA's Private Key:** The first step is to generate a private key for the Certificate Authority (CA) using the `openssl genrsa` command. This private key is essential for signing certificates.
- **Create the CA's Self-Signed Certificate:** A self-signed certificate is created for the CA using the `openssl req -x509` command. This certificate is used to sign

other certificates and is trusted within our application. It is valid for 1825 days (5 years).

- **Install the CA Certificate:** The CA certificate is installed into the system's trusted certificate store by copying it to `/usr/local/share/ca-certificates/` and running `sudo update-ca-certificates`. This ensures that it is recognized as a valid certificate authority by the system.
- **Verify the CA Certificate:** The CA certificate details are verified using `openssl x509 -in ... -noout -text` to ensure correctness. Additionally, the script checks if the CA certificate is correctly installed by searching the subject name in the system's CA certificates list.

Server Certificate Setup

- **Generate the Server's Private Key:** A private key for the server is generated using the `openssl genrsa` command. This key is used to establish secure communications.
- **Create a Certificate Signing Request (CSR):** A CSR is generated for the server using the `openssl req -new` command. This request includes information about the server and is used to create the server's certificate.
- **Generate an X509 V3 Extension File:** An extension file is created to specify the properties of the server certificate, including key usage and subject alternative names (SANs). This is done using a simple `cat` command to write the required fields into a file.
- **Sign the Server Certificate:** The server certificate is signed with the CA certificate and key using the `openssl x509 -req` command. This signed certificate is valid for 825 days and is used by the server to establish secure communications with clients.

3 - SQL Injection

A SQL injection attack consists of insertion or “**injection**” of a SQL query via the input data from the client to the application. A successful SQL injection exploit can read sensitive data from the database, modify database data (Insert/Update/Delete), execute administration operations on the database (such as shutdown the DBMS), recover the content of a given file present on the DBMS file system and in some cases issue commands to the operating system. SQL injection attacks are a type of injection attack, in which SQL commands are injected into data-plane input in order to affect the execution of predefined SQL commands.

3.1 Mitigation

Prepared statements are a powerful technique for writing secure and efficient database queries. By separating the SQL code from the data, they protect against SQL injection attacks and can also improve performance by reusing the compiled query structure for different parameters.

The key characteristics of prepared statements are two:

- **Pre-compilation:** the SQL query structure is defined once and compiled by the database server. This step happens before the actual data is supplied.
- **Parameter Binding:** the data values (parameters) are bound to the pre-compiled query structure separately. This ensures that the data is treated strictly as data and cannot be mistaken for SQL code.

4 - Cross Site Scripting XSS

4.1 XSS Attacks

XSS attacks involve **injecting client-side malicious code** (usually JavaScript or HTML) from an attacker into a web page provided by a trusted server. This code is then executed by one or more unsuspecting clients.

XSS attacks occur when an attacker exploits a web application to send malicious code, usually as browser-side scripts, to a client. The client's browser, unable to identify the script as malicious, executes it.

The malicious script can **access cookies, session tokens, and other sensitive information stored in the browser**, potentially allowing the attacker to take over the victim's account. It can also redirect the victim to attacker-controlled web content or perform other malicious operations on the victim's computer, such as installing programs. Additionally, some scripts can modify the content of the HTML page.

Various types of XSS attacks exist (Reflected/First order, Stored/Second order and DOM Based), each differing in their execution methods but ultimately resulting in similar consequences.

4.2 XSS Mitigations

The countermeasure adopted in the project to ensure security from the point of view of XSS attacks was to apply the **Three-fold Strategy**, which consists of 3 main phases: Output Validation, Input Validation and Avoiding dangerous insertion points.

These three safety requirements have been guaranteed as follows:

1. Output Validation:

To sanitize the data before embedding it within HTML, the ***htmlspecialchars()*** function was employed. This function **converts all special characters into HTML entities**, effectively encoding any potentially dangerous characters.

In specific cases, the **"ENT_QUOTES"** flag has also been included as an argument, ensuring the conversion of both double and single quotes into their corresponding HTML entities.

2. Input Validation:

This phase involves the **check and sanitization of all external inputs** (in our

case all POSTs requests submitted by users).

It was decided to use the ***filter_input()*** function, which is designed to filter and sanitize the contents of the variables it is applied to. This function ensures that the variable's content is free from malicious code, thereby preventing injection attacks. In particular, several flags have been used as needed:

- **"FILTER_SANITIZE_FULL_SPECIAL_CHARS"** to convert special characters into HTML entities.
- **"FILTER_SANITIZE_NUMBER_INT"** to remove all characters except numeric values and "+" and "-" signs.
- **"FILTER_SANITIZE_NUMBER_FLOAT"** to sanitize decimal numbers by removing all characters except numeric values, the "+" and "-" symbols, the "." (used for decimal numbers) and the character "e" (used for scientific notation).

3. Avoiding Dangerous Insertion Points:

For this step, it has been ensured that **insecure (external) data will not enter parts of the pages that are difficult to sanitize**. The following rules has been followed to avoid the insertion of external data:

- Avoid inserting untrusted data inside a **< script > tags**
- Avoid inserting untrusted data inside an **HTML attributes** supposed to **contain code/scripts** (such as the event handler **"onload"**)
- Avoid inserting untrusted data inside an **HTML attributes** that may **contain URLs** (such as **< img src = "..." > tags**)

5 - Cross Site Request Forgery XSRF

5.1 XSRF Attacks

This type of attack **manipulates the victim into sending harmful requests, leveraging their identity and privileges to execute malicious actions on their behalf.** This attack **exploits the user's authenticated session**, making it impossible for the site to differentiate between a forged request and a legitimate one sent by the victim.

Specifically, this attack targets features that **cause a state change on the server**, such as changing an email address or password. However, it does not aim to recover data from the victim, as the attacker does not receive the responses and thus gains no benefit from them.

Additionally, an attacker can use this type of attack to **obtain the victim's private data** through a technique known as **"XSRF login"**. This involves forcing an unauthenticated user to log into an account controlled by the attacker. If the victim does not realize this, they may inadvertently add sensitive information (such as personal details or credit card information) to the account. The attacker can then easily view this information along with the victim's activity history.

5.2 XSRF Mitigation

To address this vulnerability, a **specific check and function has been implemented** for its management.

The code checks if the **token** received as input when accessing a page of the web application (via a POST request) **is valid and matches the one saved in the session.** If the token is undefined (i.e., takes **null** or **false values**) or if it does not match the session token, it is considered a potential cross-site request forgery (XSRF) attack.

If token checking fails, the ***redirectIfXSRFAttack()*** function is called. This method logs an attempted XSRF attack message and terminates the request with a **"405 Method Not Allowed"** response code.

6 - Multi-Step Procedure

The vulnerability to be addressed and to ensure security concerns the **Multi-Step procedure during the checkout phase** for one or more books.

6.1 Multi-Step Procedure Vulnerability

During this phase, a user could execute HTTP requests in any sequence, deviating from the workflow defined by the developers in the application.

A malicious user might bypass one of the mandatory steps, such as entering credit card information, and directly finalize the purchase by providing only the shipping details.

To resolve this vulnerability, a common technique for verifying multi-step procedures has been implemented. Each step checks whether the previous operation has been completed before granting access to the page content. If the prior step has not been completed, access to the page is denied.

6.2 Multi-Step Checkout Workflow

It was decided to implement a workflow consisting of three steps for users who wish to purchase one or more books. The application enforces these steps to be completed in the specified order. Skipping any step will redirect the user back to the shopping cart page.

- **Step 1 - Payment Information Insertion:**

This step takes place after the user clicks on the **"checkout"** button on the **"shopping cart"** page. The user can access the next page only after correctly entering the payment information.


- **Step 2 - Shipping Information Insertion:**


This page is accessible only if the user has first correctly inserted the credit card information. Here, the user is required to enter the shipping information.

- **Step 3 - Purchase Summary Confirmation:**

In this final phase, which is accessed only if the first two phases have been completed correctly, the user reviews the purchase summary. By clicking on the **"Finalize Purchase"** button, the user completes the purchase buying the desired books.



SHOPPING CART

 Home Profile Purchases


Cart  1 Log Out

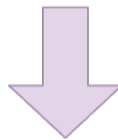
 Welcome Luca

Shopping Cart


Books & Details	Price	Quantity	Total
 <div>One Piece Vol. 97 Author: Eiichiro Oda Publisher: Star Comics</div>	\$4.94	1	\$4.94 


Total price **\$4.94**

Back to shopping  Checkout



STEP 1. PAYMENT INFO

 Home Profile Purchases

Cart  1 Log Out

 Welcome Luca

1. Payment

Credit Card

Cardholder's Name

Name Surname

Card Number


1234 5678 1234 5678

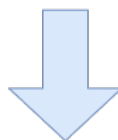
Expire

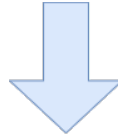
MM/YY

CVV


CVV

 Continue to Shipping Info





STEP 2. SHIPPING INFO

 Home Profile Purchases Cart 0 Log Out Welcome Luca

2. Shipping Information

Full Name


Address

City Province
Please insert an Address


Postal Code Country



STEP 3. PURCHASE SUMMARY

 Home Profile Purchases Cart 0 Log Out Welcome Luca

Checkout

Books & Details	Total Price
 <p>One Piece Vol. 97 Author: Eiichiro Oda Publisher: Star Comics Quantity: 1</p>	\$4.94

3. Purchase Summary

Total Amount
\$4.94

Shipping Info ☒
Luca
Via Garibaldi
Italy
PI
Pisa
56125

Payment Method ☒
Credit Card Number: ****1111

Figure 6.1: Multi-Step Checkout Workflow

6.3 Multi-Step Checkout Security Control Flow

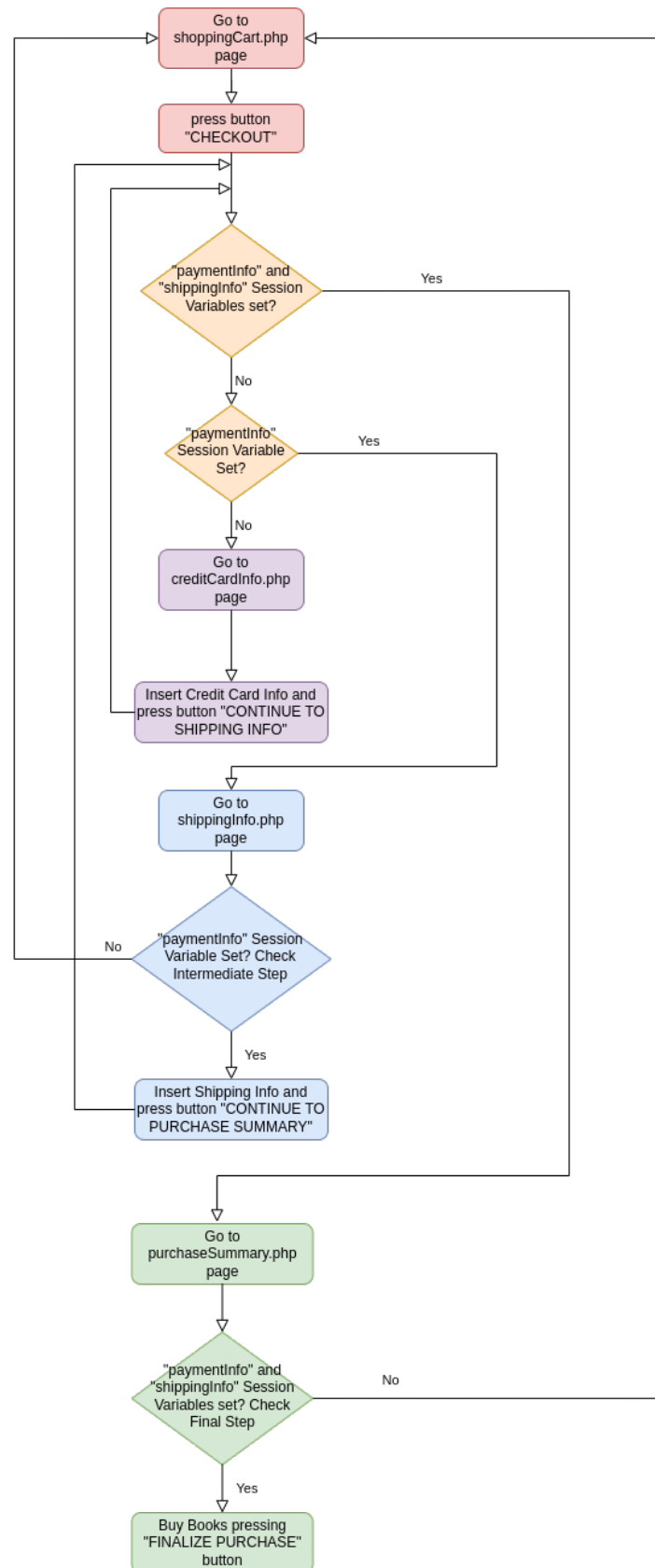


Figure 6.2: Multi-Step Checkout Flowchart

6.3.1 Security Functions Implemented

Specific security functions have been implemented to prevent users from bypassing individual checkout steps.

These functions are explained in detail following the operational flow of the checkout phases.

1. Shopping Cart Page:

- ***getNextStepToCheckout()***:

This function is called when a user goes from the shopping cart to the multi-step checkout procedure.

This function checks session variables and returns the web page to which the browser should be redirected (using the *htmlspecialchars()* function to convert the path to an HTML entity). It handles the following scenarios:

- (a) If both session variables for **credit card information and shipping information are set**, the function redirects the user to the purchase summary page.
- (b) If **only the session variable for credit card information is set**, the user is redirected to the shipping information page.
- (c) If **only the shipping address information is set** (if the user decides to modify it from the purchase summary page and then goes back to the home page), the function redirects the user to the credit card information page.

2. Credit Card Info Page:

- ***routeMultiStepCheckout()***:

This function is used across all three steps of the checkout process. It performs similar checks as the previous function but redirects using the *header()* function to specify the location, ensuring the browser is redirected to the correct page accordingly.

3. Shipping Info Page:

- ***checkIntermediateStepCheckout()***:

This function is used to correctly redirect users to the shipping information page. Like the previous function, it checks session variables, specifically ensuring the *paymentInfo* array is correctly set. If it is set, the user proceeds to the second step of the checkout process. Otherwise, the user is redirected to the shopping cart page, reporting an unauthorized access attempt.

4. Purchase Summary Page:

- ***checkFinalStepCheckout()***:

This function performs the final check in the multi-step checkout process. Unlike the previous function, it verifies whether both the variables *"paymentInfo"* and *"shippingInfo"* have been correctly set. If both are set, the user can be redirected to the final page (purchase summary) to complete the purchase. Otherwise, the user is redirected to the shopping cart page, reporting an unauthorized access attempt.

6.4 Additional Security: Credit Card Info Encryption and Decryption

Even though the credit card data is not stored in the database and is transmitted securely via HTTPS, security measures have been enhanced by encrypting the data before saving it into the session variable.

To achieve this, the server uses a 128-bit secret key, randomly generated with the help of the OpenSSL library, to encrypt and decrypt data. Currently, these functions are used exclusively to encrypt credit card data, but they are designed to handle the encryption of any type of data.

The defined functions are as follows:

- ***encryptData()***
The function takes the data to be encrypted as an argument and uses the AES-128-GCM algorithm. It returns the encrypted data.
- ***decryptData()***
The function takes the ciphertext as an argument and decrypts it using the same scheme employed for encryption.
- ***getCreditCardInfo()***
This function is called by the client. It specifically handles retrieving the *"paymentInfo"* array saved in the session variable, decrypting it, and returning the plaintext credit card information.