

COMP47910 Secure Software Engineering 2025

Assignment 3

Dimitrios Kyriakidis (24293868)

Supervisor: Liliana Pasquale



UCD School of Computer Science

University College Dublin

August 2025

Table of Contents

COMP47910 Secure Software Engineering 2025	1
Assignment 3	1
Dimitrios Kyriakidis (24293868)	1
Table of Contents	2
Executive Summary	5
Vulnerability Discovery and Methodology	6
Deliberate Design Choices	7
Report structure for each vulnerability	8
A01:2021 Broken Access Control	9
CWE-284: Improper Access Control – Cart Item Ownership Enforcement	9
CWE-285, CWE-269, & CWE-840: Improper Authorization, Privilege Management, and Business Logic Flaws	9
CWE-639: Insecure Direct Object References (IDOR)	10
CWE-384: Session Fixation	11
CWE- 352: Cross-Site Request Forgery (CSRF) for AJAX Calls	12
CWE- 862: Missing Authorization	14
CWE-1275: Sensitive Cookie with Improper SameSite Attribute & CWE-613: Insufficient Session Expiration	15
A02:2021 Cryptographic Failures	16
CWE-311: Missing Encryption of Sensitive Data	16
CWE-256 & CWE-257: Plaintext Storage of Password	17
CWE-319: Cleartext Transmission of Sensitive Information	18
CWE-315 & CWE-312: Cleartext Storage of Sensitive Information	19
CWE-598: Information Disclosure (Session ID in URL ReWrite)	20
CWE-798 & CWE-259: Use of Hard-Coded Credentials	21
A03:2021 Injection	23

CWE-89: Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)	23
CWE-20: Improper Input Validation	24
CWE-79: Improper Neutralization of Input (XSS Hardening in Layout).....	25
CWE-1336: Improper Neutralization of JavaScript.....	26
A04:2021 Insecure Design	27
CWE-307: Improper Restriction of Excessive Authentication Attempts	27
CWE-654 & CWE-308: Reliance on a Single Factor of Authentication	28
CWE-209: Generation of Error Message Containing Sensitive Information (User Enumeration).....	30
CWE-204: Response Discrepancy Information Exposure (Login)	31
A05:2021 Security Misconfiguration	33
CWE-693: Protection Mechanism Failure (Content Policy Security Not Set).....	33
CWE-1021: Improper Restriction of Rendered UI Layers (Clickjacking)	33
CWE-250 & CWE-301: Execution with Unnecessary Privileges (Database Least-Privilege)	34
CWE-550: Application Error Disclosure	35
A06:2021 Vulnerable and Outdated Components	37
CWE-1104 & CWE-937: Use of Unmaintained or Vulnerable Components	37
CWE-770 & CWE-190: Resource Exhaustion and Integer Overflow in Apache Tomcat.....	38
A07:2021 Identification and Authentication Failures	40
CWE-345: Insufficient Verification of Data Authenticity	41
CWE-620: Unverified Password Strength	42
A08:2021 Software and Data Integrity Failures	44
CWE-494: Download of Code without integrity checks	44
A09:2021 Security Logging and Monitoring Failures.....	45
CWE-778: Insufficient Logging.....	45
CWE-117: Improper Output Neutralization for Logs.....	46
CWE-532: Information Exposure Through Log Files	46
CWE-223: Omission of Security-Relevant Information	47
A10:2021 Server-Side Request Forgery (SSRF).....	49

CWE-918: Server-Side Request Forgery (SSRF)	49
Appendix: Multi-Factor Authentication (MFA) Implementation.....	51
CWE-654 & CWE-308: Reliance on a Single Factor of Authentication	51

Executive Summary¹

This report documents the work done to fix the security vulnerabilities of the “**SecuityApi**” bookstore application. I focused on defence in depth, while adding multiple layers and fully using the spring boot security framework.

Key improvements include:

- **Proactive Configuration:** The core of the security system is in config package, and mostly in **SecurityConfig.java**. This central configuration enforces HTTPS, adds critical headers like **Content Security Policy (CSP)**, and makes secure sessions.
- **Hardened Authentication:** The simple login process is completely replaced. The new system includes account logout, captcha, two-factor authentication.
- **Strong Authorization:** Use of *@PreAuthorize* annotation and the Spring Boot Security framework.
- **Secure Data Handling:** I created custom utilities to protect data. User passwords are hashed with *BCrypt* and salted. Sensitive customer information is encrypted using a **CryptoStringConverter**. A **LogSnitizer** also ensures data is sanitized before being shown on the server logs.
- **CAPTCHA, MFA and other updates:** Many updates have been gradually have been added to support a robust system that will truly defend against real vulnerabilities.

¹ The gradual updates, transformed the bookstore app into a security first class application, providing compliance to industrial practices OWASP top 10 2021, CWE warnings), while remaining usable as before. Project is stores in GitHub repository <https://github.com/kyriakidisdimitrios/securityApi>

Vulnerability Discovery and Methodology²

Finding security weaknesses was a step-by-step approach. Various methods were used to find out what was “wrong”³.

The first tool that was used was **Snyk**, which scanned the libraries of the dependencies, so **pom.xml** and other code were changed accordingly. Next, **OWASP ZAP** application was used to run automated-authenticated scans, to find general weaknesses.

With those results on hand, I focused on the **main list** of security issues, provided by our professor Liliana Pasquale for the assignment. My classmates Kishore M. and Francis Nana also provided me with their assessments on my initial vulnerability prone project.

Also, Deep dive diagnostics with **TRACE** Logging was used for debugging spring boot security chain’s decisions, regarding problems such as authentication flaws, session management, authorization failures. Finally, built-in analysis tools of IntelliJ were used to identify and fix most of the warning and typos.

² Some of the vulnerabilities, found by Kishore M. and Francis Nane, were also part of the assignment brief; I have noted them as requirements of the assignment. Similarly, some vulnerabilities that fit unto multiple categories, have been organized under category that is considered more precise.

³ This report addresses **ALL** vulnerabilities that were specified. To verify this, the reader is encouraged to search this document by the CWE number.

Deliberate Design Choices

It is important to note that while following the tutorials from this module, I made some deliberate design choices to use appropriate security control without adding complexity that would be unnecessary.

- **No use of DTOs (Data Transfer Objects):** Adding DTOs would not have increased significantly the security (for this specific application) but rather would have increased the amount of code to maintain.
- **No classes like UserServiceImpl:** Service classes mostly handle of the intelligence and complexity of the application. Creating additional classes that would inherit them, would have been redundant and would provide no security benefit (for this specific application).
- **Email based MFA over QR Code Authenticators:** For **Multi-Factor Authentication** feature, I deliberately chose to implement an **Email-based One-Time Password (OTP)** system instead of using a library like *aerogear-otp-java* for QR-code based authenticator. QR code TOTP is great, but I felt like building email OTP system from scratch is a more real-world corporate solution and an engineering challenge.

Report structure for each vulnerability

To make the report easy to follow, this is a simple breakdown of the subcategories that are used to answer what is asked from each vulnerability:

Requirement	Vulnerability Subsection
Type of vulnerability and the point in your project where it was present.	Introduction and section Where the problem was.
Appropriate mitigation to fix the vulnerability.	How I fixed it – Where strategy is described.
The appropriate mitigation to fix the vulnerability.	How I fixed it – Where code and configuration changes are shown.
Why it is effective.	Why this fix works.

A01:2021 Broken Access Control

CWE-284: Improper Access Control – Cart Item Ownership Enforcement

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

We modified the Cart functionality, to make sure that all cart operations like update or remove, are scoped in the authenticated customer, to prevent **IDOR** as explained at the **CWE-639** section.

WHERE THE PROBLEM WAS

CartController.java, CartItemService.java, CartItemRepository.java

HOW IT IS FIXED

- Replaced *updateQuantity* and *removeCartItemById* methods, with owner verified methods *updateQuantityOwned(cartItemId, quantity, customer)* and *removeCartItemOwned(cartItemId, customer)*.
- Added safe parameter “parsing” using *getOrDefault* with default values -1 for IDs, 0 for quantities, to prevent invalid inputs.
- Added checks for validity *cartItemId >= 0, quantity >= 1*, to reject malformed requests.
- All cart-related endpoints now do require active logged in session, which is explained at **CWE-639** section.

WHY THIS FIX WORKS

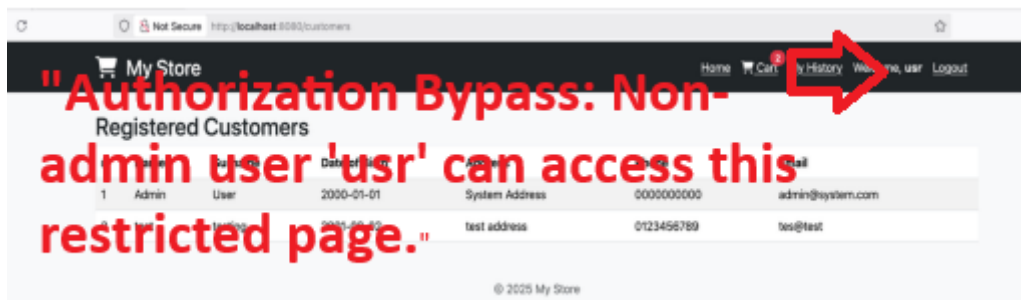
Prevents users from manipulating the carts that don’t belong to them, mitigating IDOR vulnerabilities.

CWE-285, CWE-269, & CWE-840: Improper Authorization, Privilege Management, and Business Logic Flaws

Endpoint “/customers”, was designed to show list of all registered users and their personal information and didn’t check for user permissions properly. The single flaw created many related weaknesses, e.g. a failure of authorization (**CWE-285**), led to improper privilege management, and was itself a significant business logic error (**CWE-840**).

WHERE THE PROBLEM WAS

customercontroller.java was missing authorization checks. Any authenticated user could navigate to /customers URL and be granted access to all private data for every other user in the system.



```
// Customer list page
@GetMapping("/customers") @kyriakidisdimtrios
public String listCustomers(Model model) {
    List<Customer> customers = customerService.getAllCustomers();
    model.addAttribute("customers", customers);
    return "customers";
    // TODO: Restrict this page to ADMIN only in the future with Spring Security
}
```

HOW IT IS FIXED

Authorization has been implemented in the Spring boot framework. Only users that are granted the role ADMIN can have access to customer list. Permission checks are performed automatically by framework.

SecurityConfig.java:

```
// filterChain
.authorizeHttpRequests(auth -> auth // ...other rules
//WE ADDED THIS NEW RULE
.requestMatchers("/admin/**").hasRole("ADMIN") .
//WE ADDED THIS NEW RULE
.requestMatchers("/customers/**").hasRole("ADMIN")
.anyRequest().authenticated())
```

As a defence in depth measure, the `@PreAuthorize("hasRole('ADMIN')")` annotation has also been added above the `listCustomers` in the **CustomerController** as well.

WHY THIS FIX WORKS

User without proper privileges is blocked before the code fetches sensitive customer information can be executed. Also, it is no longer possible for a developer to forget a manual check, because the permission check is centralized and fail-safe. The above fixed code resolves the improper authorization, the resulting privilege management, and the business logic flaw.

CWE-639: Insecure Direct Object References (IDOR)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

There was a serious flaw in the functionality of cart. When user was changing the quantity of an item in the cart, the application was trusting the *cartItemId* that was been sent from the user's browser. Thus, a logged-in attacker could send the *cartItemId* and change or delete their cart.

WHERE THE PROBLEM WAS

CartController.java, CartItemService.java, CartItemRepository.java

HOW IT IS FIXED

Added ownership properties on methods to enforce current user's ID in the query.

CartItemRepository.java:

```
Optional<CartItem> findByIdAndCustomer_Id(Long id, Long customerId)
void deleteByIdAndCustomer_Id(Long id, Long customerId)
```

CartItemService.java:

```
updateQuantityOwned(Long cartItemId, int quantity, Customer customer)
removeCartItemOwned(Long cartItemId, Customer customer)
```

CartController.java:

```
/cart/update-ajax now calls cartItemService.updateQuantityOwned(cartItemId,
quantity, customer);
/cart/remove-ajax now calls cartItemService.removeCartItemOwned(cartItemId,
customer);
```

On missing session, both endpoints return *{ success:false, message:"Not logged in" }*.

WHY THIS FIX WORKS

Multi-layer fix makes **IDOR** impossible. The check is now on the database query itself. System cannot modify an item that does not belong to a logged-in user.

CWE-384: Session Fixation

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

The application was at the risk of *session fixation* attack. If a session identifier created pre – login “survives” the authentication, the attacker who knows the ID can hijack the authenticated session (non-secure cookies, missing ID rotation, session id not being disabled).

WHERE THE PROBLEM WAS

A collection of missing security configurations across all the session management system. Core locations were **SecurityConfig.java** and **application.properties**.

HOW IT IS FIXED

By implemented a multi-layer defence to shut down all common session fixation vectors. IDs should always be unpredictable, are not exposed 9e.g. to URLs), and are protected.

Enable Session Rotation (SecurityConfig) – Automatically migrate the session on a successful login, that destroys the old session and makes a new unpredictable ID:

```
.sessionManagement(sess -> sess  
.sessionFixation(SessionManagementConfigurer.SessionFixationConfigurer::migrate  
Session))
```

Disabled URL Session Tracking (application.properties) – Disabled the insecure feature of putting session IDs in the URL, by setting up this property:

```
server.servlet.session.tracking-modes=COOKIE
```

Hardened the Session Cookie(application.properties) - Added several properties to make session cookie more secure.

```
server.servlet.session.cookie.http-only=true // Stops JavaScript from reading  
the cookie server.servlet.session.cookie.secure=true // Ensures the cookie is  
only sent over HTTPS server.servlet.session.cookie.same-site=Strict // Stops  
the browser from sending the cookie to other websites
```

WHY THIS FIX WORKS

This is a strong defence against session fixation and hijacking. The ID rotation breaks the links between pre-login and the authenticated one. The other settings are defences in depth.

CWE- 352: Cross-Site Request Forgery (CSRF) for AJAX Calls

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

Application was using **AJAX** for dynamic actions, like updating an item's quantity that let my colleague be able to exploit it (e.g. updating the quantity in the shopping cart). Disabling **CSRF**, made me initially possible to better implement the AJAX functionality, but attacker can still trick the victim's browser into sending a malicious request.

WHERE THE PROBLEM WAS

SecurityConfig.java

```
88 @EnableWebSecurity //Enables Spring Security for your web app
89 public class SecurityConfig {
90     @Bean & kyriakidisdimetrios
91     public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
92         http
93             .csrf(CsrfConfigurer<HttpSecurity> csrf -> csrf.disable()) // needed for POST to work without tokens
94             .authorizeHttpRequests( AuthorizationManagerRequestMat... auth -> auth
95                 .requestMatchers( @ "/login", @ "/register", @ "/customLogout", @ "/css/**", @ "/js/**", @ "/webjars/**"),
96                 .permitAll() AuthorizationManagerRequestMat...
97                 .anyRequest().permitAll() //Your current config is not enforcing login or roles. If you want to protect at
98             );
99
100     //
101     // .permitAll()
102     // .requestMatchers("/admin/**").authenticated() // 🛡 Admin pages require login
103     // .anyRequest().authenticated() // 🛡 All other routes require login
```

HOW IT IS FIXED

We create a bridge that Spring Security needs, between server side and client side, via JavaScript.

- **Keep CSRF enabled(SecurityConfig.java)**, line `.csrf(csrf -> csrf.disable())` is erased.
- **Exposing the Token in the Meta Tags(layout.html)**

```
<head> ...other things<meta name="_csrf" th:content="${_csrf.token}"/>
<meta name="_csrf_header" th:content="${_csrf.headerName}"/> </head>
```

- **Reading/Sending Token via JavaScript(cart.js)**

```
const CSRF_TOKEN_META = document.querySelector('meta[name="_csrf"]');
const CSRF_HEADER_META = document.querySelector('meta[name="_csrf_header"]');
function buildJsonHeaders() {
    const headers = { 'Content-Type': 'application/json' };
    const token = CSRF_TOKEN_META?.getAttribute('content');
    const header = CSRF_HEADER_META?.getAttribute('content');
    if (token && header) { headers[header] = token; } return headers; }
// Update, remove items
fetch('/cart/update-ajax', { method: 'PUT', headers: buildJsonHeaders(), //
    Gives the CSRF token
    body: JSON.stringify({ cartItemId, quantity }) })
```

HOW IT IS FIXED

This way, Spring Security can verify that every request to change data is authentic and comes from our own website.

CWE- 862: Missing Authorization

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

In the original version. No role-based access was applied to `/admins/books/add`. Any authenticated user who knew this endpoint could add books to the catalogue.

WHERE THE PROBLEM WAS

CustomerController.java, SecurityConfig.java

```
@PostMapping("/admin/books/add")
public String addBook(@ModelAttribute Book book, Model model) {
    // Check for existing book with same title, authors, and year
    if (bookService.bookExists(book.getTitle(), book.getAuthors(), book.getYear())) {
        model.addAttribute("error", "A book with the same title, authors, and year already exists.");
        model.addAttribute("book", book);
        return "admin_book_form";
    }
}
```

HOW IT IS FIXED

Enforced strict authorization for all admin endpoints with Spring Security. We created a rule that only admins are allowed to access any URLs that start with `/admin/`

FilterChain (SecurityConfig.java),

```
// In SecurityConfig.java's filterChain method
.authorizeHttpRequests(auth -> auth //...other permitAll rules
.requestMatchers("/admin/**").hasRole("ADMIN").anyRequest().authenticated())
```

Method Level defence(CustomerController.java)

```
@PreAuthorize("hasRole('ADMIN')")
@PostMapping("/admin/books/add")
public String addBook(@ModelAttribute Book book, Model model)
```

WHY THIS FIX WORKS

Prevents non admin users to perform administrative actions, even if they know the endpoints. Spring Security handles the request **before it reaches the controller**.

CWE-1275: Sensitive Cookie with Improper SameSite Attribute & CWE-613: Insufficient Session Expiration

IDENTIFIED BY KISHORE M., DURING SECURITY REVIEW

Application session management had many security flaws. The most specific one, was the *JSESSIONID* cookie missing the *SameSite* attribute. Without this, application was vulnerable to **CSRF** attacks. Furthermore, the application had not sufficient session expiration, because it had long timeout by default and not strong rules for handling sessions that were expired or not valid.

WHERE THE PROBLEM WAS

application.properties

HOW IT IS FIXED

Enforced a specific strict set off rules, for session management, not just only for **CSRF**, e.g. *SameSite* to be Strict.

application.properties:

```
# === SESSION SETTINGS ===
# Enforces a 15-minute timeout (addresses CWE-613)
server.servlet.session.timeout=15m
# Disables insecure URL-based session tracking (addresses CWE-384)
server.servlet.session.tracking-modes=COOKIE
# Stops JavaScript from accessing the cookie, to preventing XSS
server.servlet.session.cookie.http-only=true
# Ensures the cookie is only over HTTPS connection
server.servlet.session.cookie.secure=true
# Fixes CWE-1275 by preventing the cookie being sent on cross-site requests
server.servlet.session.cookie.same-site=Strict
```

WHY THIS FIX WORKS

It applies directly to session cookies. *SameSite=Strict* is the fix for the reported vulnerability. Additional flags like *Secure* and *HttpOnly* also protect session from common attacks. *Secure* prevents cookies from unencrypted networks, while *HttpOnly* prevents it to be stolen by using Javascript in a **XSS** attack from client side. Setting a short *session.timeout* (15 minutes) addresses insufficient session expiration and minimizes the chance attacker to use a session ID which is “stolen”.

A02:2021 Cryptographic Failures

CWE-311: Missing Encryption of Sensitive Data

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application had a risk of exposing sensitive data. The primary risk was that sensitive values from user requests, such as payment card and passwords, could accidentally be written to the server log in plain text. There was a lack of centralized strategy that needed to be fixed and be queued as a must do thing on the security chain.

WHERE THE PROBLEM WAS

application.properties

HOW IT IS FIXED

Enforce strong encrypt on data transit(application.properties):

```
spring.datasource.url=jdbc:mysql://localhost:3306/securityapi?sslMode=REQUIRED
```

Ensure sensitive data like card numbers are never stored:

Checkout logic in **CartController**, uses card number one time only and never saves or logs them,

Implement log scrubbing, to find and mask any sensitive data that is written (SensitiveDataSanitizer.java):

SensitiveDataSanitizer.java: Contains list of sensitive keywords like password, card, cvv, and the method to mask their values:

```
// A denylist of keywords to be masked - Part of SensitiveDataSanitizer.java
private static final Set<String> SENSITIVE_KEYS = Set.of(
    "password", "pass", "pwd",
    "paymentInfo", "card", "cardNumber", "cc", "cvc", "cvv");
```

SensitiveRequestLoggingFilter.java: If debug logging is enabled, it intercepts the request parameters and passes them to **SensitiveDataSanitizer** to mask the sensitive values.

```
try { Map<String, String[]> safeParams =
SensitiveDataSanitizer.maskParams(request.getParameterMap()); //Mask before log
    log.debug("REQ {} {} params={}", request.getMethod(),
request.getRequestURI(), safeParams);
} catch (Exception ignore) { /* ... */ }
```

WHY THIS FIX WORKS

Transport encryption, never storing sensitive data and automated log scrubbing provide extra safety over potential exploitations. This meets the requirements of the **CWE-311** without needing to alter the **MVC** logic or database structures.

CWE-256 & CWE-257: Plaintext Storage of Password

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application was storing passwords in plaintext because controller, or service was not hashing them, leading to potential exposure of sensitive data.

WHERE THE PROBLEM WAS

CustomerService, CustomerController

HOW IT IS FIXED

Not use any plaintexts for passwords, but rather hashed the text and use it in a centralized way. We used **BCrypt** hashing to accomplish this, and it is used in the *CustomerService*, so this makes sure that the “save customer” action will securely hash the sensitive data, no matter where the action was triggered from.

Centralized Hashing(CustomerService): *saveCustomer* is changed to use **BCryptPasswordEncoder** before saving the password. Also, it uses this code: *!customer.getPassword().startsWith("\$2a\$")* to prevent passwords that are already hashed before they are processed by **BCryptPasswordEncoder**.

Remove Hashing from Controller: During the updates, many ways of hashing were tested. Password hashing code that was used in **CustomerController** was removed. This makes controller simpler, and makes more robust Service, providing better business logic.

Secure Password Verification(CustomerService.java) *authenticateCustomer* method makes use of *passwordEncoder.matches* to compare user’s password with the stored hash. In this way we don’t reveal any credentials:

```
public boolean authenticateCustomer(String username, String rawPassword) {  
    Customer customer =  
customerRepository.findByUsername(username).orElse(null);  
    if (customer == null) { /*...handle user not found*/ }  
    return passwordEncoder.matches(rawPassword, customer.getPassword());}
```

Preventing Leaks that could happen by accident: In the class **Customer.java**, *@JsonIgnore* and *@ToString.Exclude* annotations were added on the password field. This way, API responses and server logs are prevented of revealing passwords.

```
@NotBlank(message = "Password is required")  
@JsonIgnore  
@ToString.Exclude  
private String password;
```

WHY THIS FIX WORKS

Hashing is now unavoidable, and logic is centralized in the service layer. Using **BCrypt** and salting the hashes, makes the database nearly impossible to reverse-engineer and exploit.

CWE-319: Cleartext Transmission of Sensitive Information

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

There was a catholic configuration issue, were transmitted sensitive data over unencrypted channels. Communication was over **HTTP**, thus sensitive data could be stolen by attacker who were using the same network.

WHERE THE PROBLEM WAS

SecurityConfig.java

HOW IT IS FIXED

Enforced HTTPS for All Web Traffic(SecurityConfig.java): adding the `.requiresChannel().anyRequest().requiresSecure()`, telling Spring Security to reject all non-TLS connections.

```
http
    //Reject any request not over a secure TLS (HTTPS) connection.
    .requiresChannel(ch -> ch.anyRequest().requiresSecure())
```

Added Automatic HTTP-to-HTTPS Redirection(HttpToHttpsRedirectConfig.java): Configures the Tomcat server to listen to HTTP port 8080 and redirects them to HTTPS port 9443.

```
@Bean
public WebServerFactoryCustomizer<TomcatServletWebServerFactory>
servletContainer() {
    return server -> {if (httpPort > 0) {
        Connector connector = new
Connector(TomcatServletWebServerFactory.DEFAULT_PROTOCOL);
        connector.setPort(httpPort);
        connector.setRedirectPort(httpsPort); /* Redirect to the secure
port */
        server.addAdditionalTomcatConnectors(connector);}};}
```

Implemented HSTS -Strict Transport Security(SecurityConfig.java): To prevent attacker to makes tricks and successfully connect to HTTP, I added the HSTS header, which says that for the next year, it must connect only via HTTPS.

```
// Security headers hardening
.headers( HeadersConfigurer<HttpSecurity> headers -> headers
    .httpStrictTransportSecurity( HstsConfig hsts -> hsts
        .maxAgeInSeconds(31536000)
        .includeSubDomains(false)
        .preload(false)) // HSTS → reduces downgrade/mitM
    .contentSecurityPolicy( ContentSecurityPolicyConfig csp -> csp.policyDirect
        "default-src 'self'; img-src 'self' data:; script-src 'self'
// CSP limits script/inline/script-src → mitigates XSS (CWE-79/1336)
// frame-ancestors 'none' → Clickjacking (CWE-1021)
.referrerPolicy( ReferrerPolicyConfig rp -> rp.policy(ReferrerPolicyHeade
// Referrer-Policy reduces info leakage (CWE-200)
.frameOptions(HeadersConfigurer.FrameOptionsConfig::sameOrigin)
// Clickjacking defense (legacy) (CWE-1021)
.contentTypeOptions( ContentTypeOptionsConfig cto -> {}) // X-Cont
)
```

Enabled Secure Cookies(application.properties): All session cookies are flagged as secure, thus only transmitted over HTTPS.

```
server.servlet.session.cookie.secure=true
```

Encrypted the Database Connection(application.properties): Enabled TLS encryption for MySQL and application connection, by using `sslMode=REQUIRED`

```
spring.datasource.url=jdbc:mysql://localhost:3306/securityapi?sslMode=REQUIRED
```

WHY THIS FIX WORKS

Provides end-to-end protections for all sensitive data. Combines HTTPS enforcement, automatic redirection and HSTS.

CWE-315 & CWE-312: Cleartext Storage of Sensitive Information

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Similarly to **CWE-319**, the application was storing customer information (like address, email, phone number, etc) as plain text. If attacker could gain access to database, their information could potentially be leaked.

WHERE THE PROBLEM WAS

HOW IT IS FIXED

Created a new utility class(CryptoStringConverter.java): Implements JPA's *AttributeConverter* interface, and has the logic to encrypt, decrypt string data.

```
@Converter // Part of CryptoStringConverter.java
public class CryptoStringConverter implements AttributeConverter<String,
String> {
    @Override
    public String convertToDatabaseColumn(String attribute) {
        try { /* encryption methods */ } catch (Exception e) { /* ... */ }
        return encryptedString;
    }
    @Override
    public String convertToEntityAttribute(String dbData) {
        // This method contains the logic to DECRYPT the database text when
        reading.
        try { /* decryption methods */ } catch (Exception e) { /* ... */ }
        return decryptedString;}}}
```

Encryption Key is external from now on(CryptoStringConverter.java): Converter is getting the key from environment named APP_DATA_KEY. This way code and secrets are separated.

```
private static SecretKeySpec loadKey() {
    try {
        String b64 = System.getenv("APP_DATA_KEY");
        if (b64 == null || b64.isBlank()) return null;
        byte[] key = Base64.getDecoder().decode(b64); //...
```

Applied Converter to the Entity(Customer.java): *@Convert(converter = CryptoStringConverter.class)* annotation was added on top of *address*, *phoneNumber* and *email*.

```
@Convert(converter = CryptoStringConverter.class)
private String address;
@Convert(converter = CryptoStringConverter.class)
private String phoneNumber;
@Convert(converter = CryptoStringConverter.class)
private String email;
```

WHY THIS FIX WORKS

Encryption now is automatic and invisible to the application. The protection is catholicly enforced every time that Customer object is saved. This encryption is **JPA** handled, so repository methods like *findByEmail* are still working.

CWE-598: Information Disclosure (Session ID in URL ReWrite)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application was vulnerable to leak session attributes, when URL rewriting was done. Tomcat web server for example is configured by default to automatically add user's session ID on URL when cookies were disabled.

WHERE THE PROBLEM WAS

SecurityConfig.java

HOW IT IS FIXED

Insecure fallback has completely been disabled.

Created new class to hold this security setting(DisableUrlSessionIdConfig.java);

Tracking mode was set to cookies only(DisableUrlSessionIdConfig.java):

```
@Configuration
public class DisableUrlSessionIdConfig {
    @Bean
    public ServletContextInitializer servletContextInitializer() {
        return servletContext->{
//Disables URL rewriting, forces cookies only tracking
servletContext.setSessionTrackingModes(EnumSet.of(SessionTrackingMode.COOKIE));
};}}
```

WHY THIS FIX WORKS

Removes the vulnerability at its source. Secure and recommended trade-off. When users have disabled cookies, user cannot login. Session ID is kept in secure *HttpOnly* cookie, that is not revealed in the URL.

CWE-798 & CWE-259: Use of Hard-Coded Credentials

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

Username and password on the database were stored in plain text. Similar problem was with SQL script, which had hard-coded, plain text password for the default admin account.

WHERE THE PROBLEM WAS

application.properties

HOW IT IS FIXED

Switched to least privilege user(application.properties): As described on the **CWE-250** section, we stopped using the high-privilege user *root*, but instead used *jimboy3100*, which has far fewer permissions.

```
# Least privilege user
spring.datasource.username=jimboy3100
spring.datasource.password=Jimboy31
```

Replaces the Plaintext Admin Password: Removed the plain text password 'admin' from SQL script. Generated a new SQL command containing the same password but hashed and salted.

```
INSERT INTO customers (... , password, ...) VALUES (... ,
'$2a$10$8so23zwzK...FaVYMZA.', ...);
```

WHY THIS FIX WORKS

The credentials belong to the least privilege user, which drastically reduces the amount of damage an attacker can do if source code is leaked. Also replacing default password with strong hash ones, lets the code be successfully hardened.

A03:2021 Injection

CWE-89: Improper Neutralization of Special Elements used in an SQL Command (SQL Injection)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

SQL injection is a critical vulnerability that can happen when the application mixes SQL queries with user texts. The application itself does not have such a flow because Spring Boot JPA provides some protection. If the developer decides to write native SQL queries, it will create severe vulnerabilities.

WHERE THE PROBLEM WAS

JPA repositories.

HOW IT IS FIXED

Security control is implemented in the Spring Data JPA in all the application. This is not a specific fix, but a safer pattern to solve this type of vulnerability.

Methods like *findByUsername*, are defined in the interface.

Use Repositories Exclusively(CustomerRepository.java): All databases access is driven through repository interfaces, like CustomerRepository.java.

No Native Queries: Full review performed on the repositories that is no use of *EntityManager.createNativeQuery*.

No String Concatenation: Verified that nowhere in the application is a database query being built by manually joining string.

```
public interface CustomerRepository extends JpaRepository<Customer, Long> {  
    // Spring Data JPA automatically turns this into a safe, parameterized  
    query:  
    // "SELECT * FROM customers WHERE username = ?"  
    // safe parameter.  
    Optional<Customer> findByUsername(String username);}
```

WHY THIS FIX WORKS

Spring Data JPA that does not use native queries combined with user input. The underlying database driver makes use of escaping for the special characters.

CWE-20: Improper Input Validation

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

The attacker could send malformed data to server, to cause server errors, or abuse the application features. Specific problems found on **Customer.java** that didn't have strict formatting rules for fields like username and password, and on **CartController.java** which didn't validate number inputs for cart quantities, allowing negative numbers.

WHERE THE PROBLEM WAS

Customer.java, CartController.java

HOW IT IS FIXED

Enforce strong, multi-layer validation that will work catholicly on the application. The plan was to reject invalid data **at the earliest possible time**.

Bean Validation at the Entity Level: Added validation annotations for attributes on **Customer.java** and **CartItem.java**, like:

- `@NotBlank` `@Email` and `@Past`, to ensure the field that that formatted in the proper way.
- `@Pattern` annotation for stronger regexp to enforce strict usernames., names and passwords.
- `@Min(1)` as cart item quantity for non-negative numbers.

```
// Part of Customer.java
@Column(nullable = false, unique = true)
@NotBlank(message = "Username is required")
@Pattern(regexp = "^[A-Za-z0-9._-]{3,32}$", message = "...")
private String username;
@NotNull(message = "Date of birth is required")
@Past(message = "Date of birth must be in the past")
private LocalDate dateOfBirth;
@NotBlank(message = "Email is required")
@email(message = "Email should be valid")
private String email;

// Part of CartItem.java
@Column(nullable = false)
@Min(1) @Max(9999)
int quantity;
```

Controller-Level Validation(CustomerController.java): Applied `@Valid` annotation on the `registerCustomer` method. In essence, this says to Spring Boot to run all the Bean Validation checks on the Customer, before my own code runs. Also checks for business rules, like ensuring the date of birth 1900-2010 were added.

```
// Domain validation
if (customer.getDateOfBirth().isBefore(LocalDate.of(1900, 1, 1)) ||
    customer.getDateOfBirth().isAfter(LocalDate.of(2010, 12, 31))) { // ...
```


Domain Logic Validation: Before user is saved, the controller performs checks to see if the username, phone number, or email exists in database and provide feedback to user.

WHY THIS FIX WORKS

Multi-layer approach is effective, because it makes sure that no invalid information can be stored in the system. By enforcing the validation at the entity, controller and domain logic layers, the application becomes robust, against improper inputs, and invalid data is rejected early.

CWE-79: Improper Neutralization of Input (XSS Hardening in Layout)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

The shared page template *layout.html* had some weaknesses. The problem was that the missing `<meta>` tags for **Cross Site Request Forgery (CSRF)** token to the JavaScript. Without these tags, AJAX calls would not have been **CSRF** protection is the correct way. Additionally, while the logout link was present, it's a security best practice to make sure that only the correct user is logged in.

WHERE THE PROBLEM WAS

layout.html

HOW IT IS FIXED

Make sure that anti-CSRF tokens were always available to every page and enforce actions only on displayed logged in users.

Added CSRF Meta Tags(layout.html): Added two `<meta>` tags on the head of the template, that would automatically get populated by Thymeleaf and Spring Security with unique CSRF token and the correct header for user's session.

```
<head>
  <meta name="_csrf" th:content="${_csrf.token}"/>
  <meta name="_csrf_header" th:content="${_csrf.headerName}"/>
</head>
```

Preserved Conditional Rendering for Logout(layout.html): Verified that logout link is correctly implemented in Thymeleaf `th:if`. Logout is only sent only if valid session.

```
<a th:if="${session.loggedInUser}" th:href="@{/customLogout}">Logout</a>
```

WHY THIS FIX WORKS

This fix hardens the security on every page. It adds CSRF meta tags for JavaScript and AJAX requests. Conditional rendering for the logout link is a good security practice that prevents non authenticated users to access specific endpoints.

CWE-1336: Improper Neutralization of JavaScript

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

This vulnerability is like the **CWE-693** one but focus more on JavaScript injection. Even though, ThymeLeaf has good default protection, application is missing browser level **Content Security Policy (CSP)**. If browser makes a mistake, e.g. use unsafe *th:utext* attribute or by including user input in an inline script, the attacker could inject and run JavaScript scripts in the browser of the victim.

WHERE THE PROBLEM WAS

securityconfig.java

HOW IT IS FIXED

Content Security Policy (CSP) header is added to the application configuration(SecurityConfig.java): This policy tells the browser which pages are trusted to run scripts. On the *header()* block, the code *.contentSecurityPolicy()* is implementing this policy:

```
http
    .headers(headers -> headers
        .contentSecurityPolicy(csp -> csp.policyDirectives(
            "default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-
            inline'; frame-ancestors 'none'"))))
```

WHY THIS FIX WORKS

This is 2nd layer of defence. Server-side escaping is the main level of protection, although CSP makes sure that scripts that are considered unsafe, are not injected on the pages, and the browser will refuse to run.

A04:2021 Insecure Design

CWE-307: Improper Restriction of Excessive Authentication Attempts

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

There wasn't any defense against brute force password attacks. When an attacker knows the username (or not), of the victim, he can use brute force and dictionaries to find the password of the victim.

WHERE THE PROBLEM WAS

application.properties

HOW IT IS FIXED

Track the number of attempts that failed for each username. After a specific number of failures, account is locked temporarily for a specific period.

Created a new class to use in-memory tacking of the number of wrong attempts per username and store the timestamp when the lockout expires(LoginAttemptService.java)

Made Lockout Rules(application.properties):

```
security.auth.max-failed-attempts=10
security.auth.lockout-minutes=10
```

Embed Security Filters(LockoutFilter.java): Made custom filter and placed it in the filter chain **before** the authentication process. If account is locked, it redirects to a custom page immediately.

```
@Override // Part of LockoutFilter.java
protected void doFilterInternal(...) throws ServletException, IOException {
    if ("POST".equalsIgnoreCase(request.getMethod()) &&
        "/login".equals(request.getServletPath())) {
        String username = request.getParameter("username");
        if (attemptService.isLocked(username)) {
            long mins = attemptService.minutesLeft(username);
            response.sendRedirect("/login?locked&mins=" + mins);
            return;}}chain.doFilter(request, response);}
```

Login Handlers(LoginFailureHandler.java) used to notify **LoginAttemptService** about the failed attempts. When an attempt is successful, it counts back to zero.

```
// Part of LoginAttemptService.java
public void onFailure(String username) {
    if (username == null) return;
    Entry e = store.computeIfAbsent(username.toLowerCase(), k -> new Entry());
    e.count++;if (e.count >= maxFailedAttempts) {
        e.lockUntil = Instant.now(clock).plus(lockoutDuration);}}
```

```
public boolean isLocked(String username) {
    if (username == null) return false;
    Entry e = store.get(username.toLowerCase());
    if (e == null || e.lockUntil == null) return false;
    return Instant.now(clock).isBefore(e.lockUntil);}
}
```

Use Feedback added: System redirects to `/login?locked&mins={duration}`, when user unsuccessfully tries to login. Then **login.html** was modified to read the URL parameters and show a message.

```
response.sendRedirect("/login?locked&mins=" + mins);
```

WHY THIS FIX WORKS

Automated username attacks have effectively been shut down. There is only a small number of guesses that can be made at a time, making the brute force impractical to use. User friendly messages show the user why they are locked out, reducing the confusion and requests for support.

CWE-654 & CWE-308: Reliance on a Single Factor of Authentication

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

The application was designed to rely on a single authentication system, so if an attacker managed to find out the username and password of an account, there would be security breach and exploitation of that account.

WHERE THE PROBLEM WAS

SecurityConfig.java

HOW IT IS FIXED

A separate section is created in this report about the full Multi-Factor Authentication implementation; the immediate strategy here was to add a second check to both the login and the register pages. **CAPTCHA** was implemented as a 2nd verification step, while it is a strong technique to reduce not only brute force attacks, but also bot spam like mass account creations.

A new class was created that generates random texts challenge and validates the answer against the values which was stored in the user's session(CaptchaService.java).

```
// Part of CaptchaService.java
public String generateCaptcha(HttpSession session) { /*...*/
    session.setAttribute("captcha", captchaStr);
    return captchaStr;}
public boolean validateCaptcha(String userInput, HttpSession session) {
    String storedCaptcha = (String) session.getAttribute("captcha");
    return storedCaptcha != null && storedCaptcha.equalsIgnoreCase(userInput);}
```

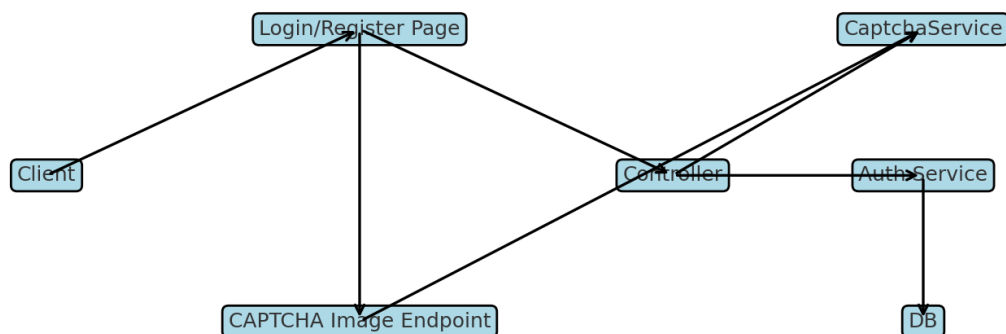
A new class created intercept POST requests(**CaptchaValidationFilter.java**), and check if the CAPTCHA is correct first, rejecting the request otherwise.

```
@Override //Part of CaptchaValidationFilter.java
protected void doFilterInternal(...) throws ServletException, IOException {
    if ("POST".equalsIgnoreCase(request.getMethod()) &&
        "/login".equals(request.getServletPath())) {
        String captcha = request.getParameter("captcha");
        if (!captchaService.validateCaptcha(captcha,
            request.getSession(false))) {
            response.sendRedirect("/login?error=Invalid%20CAPTCHA");
            return;}}chain.doFilter(request, response);}
```

This filter was placed into the Security Chain(**SecurityConfig.java**), where it runs before the *UsernamePasswordAuthenticationFilter* and after the *LockoutFilter*:

```
http.addFilterBefore(lockoutFilter,
    UsernamePasswordAuthenticationFilter.class);
http.addFilterAfter(captchaFilter, LockoutFilter.class);
```

CAPTCHA Integration in Authentication Flow



WHY THIS FIX WORKS

It is a barrier against automated scanners. An Attacker is no longer able to test many passwords but rather must solve this unique CAPTCHA⁴ for each attempt.

CWE-209: Generation of Error Message Containing Sensitive Information (User Enumeration)

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

Security assessment from my colleague Francis Nane, identifies that the registration form uses specific error messages. When a new user tries to register a taken username, email or phone, it responds to the issue itself, revealing for example that the email is taken. From a strict security standpoint, this could be an exploitation, but from a business perspective, it is a risk that provides user friendly causality, reasoning why the registration failed. For these reasons, it was decided to keep this vulnerability. Although, by searching out for **CWE-209**, generic error replies technique was implemented, but to intercept the tack traces if an unexpected occurs. Safe logging provides a more sanitized logging output too.

WHERE THE PROBLEM WAS

Customercontroller.java

Create an Account

Username

Username already exists

First Name

Surname

Date of Birth

Address

HOW IT IS FIXED

Generic Error Replies: I ensured **GlobalExceptionHandler** is in place to catch error, to show generic messages without revealing stack traces.

```
@ExceptionHandler(Exception.class) //Part of GlobalExceptionHandler.java
public String handleGeneralException(Exception ex, RedirectAttributes
redirectAttributes) {
    logger.error("Unexpected error: {}", s(ex.getMessage()), ex);
    redirectAttributes.addFlashAttribute("errorMessage",
```

⁴ It is not a replacement for the full MFA but is a 2nd layer of verification that needs human intervention.

```
"An unexpected error occurred. Please contact support if the issue persists.");  
return "redirect:/error";}
```

Safe Logging(LogSanitizer.java): All server-side logs for what caused the failure as been filtered by LogSanitizer, to prevent injections of logs or exposure of them.

```
public static String s(Object o) { //Part of LogSanitizer.java  
    if (o == null) return "null"; String str = String.valueOf(o)  
        .replace('\r', '_').replace('\n', '_').replace('\t', ' ');  
    return str.length() > MAX ? str.substring(0, MAX) + "..." : str;}
```

(Not changed) Messages to user, for specific registration errors (Customercontroller.java):

```
// Unchanged. Same for email and phone number  
if (customerService.findByUsername(customer.getUsername()) != null) {  
    result.rejectValue("username", "error.customer", "Username already exists");  
    return "register";}
```

WHY THIS FIX WORKS

Generic error replies and safe logging implementations is fully explained on the introduction. As for Francis Nana suggestion, the attacker cannot effectively exploit the registration because:

CAPTCHA is required to every attempt.

Account Lockout is in place, because there is strong implementation of account lockout (**CWE-307**) that will prevent brute force.

CWE-204: Response Discrepancy Information Exposure (Login)

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

Although this application uses default behaviour, this flaw would be present if there was a custom login logic, like giving response “username not found”, or “invalid password”. The practise here was to ensure that this would never happen, thus created a login failure handler that takes into control the login error process⁵.

WHERE THE PROBLEM WAS

LoginFailureHandler.java

⁵ **CaptchaValidationFilter.java** which also uses this technique, was designed to the return the specific captcha error, to improve user experience: `response.sendRedirect("/login?error=Invalid%20CAPTCHA");`

HOW IT IS FIXED

Expanding the custom class for login failures to handle login errors too (LoginFailureHandler.java):

```
        if (attemptService.isLocked(username)) {
            long mins = attemptService.minutesLeft(username);
            log.warn("Account temporarily locked for user='{}' ({} min left)",
s(username), mins);
            response.sendRedirect("/login?locked" + (mins > 0 ? ("&mins=" +
mins) : ""));}
        else {log.warn("Authentication failed for user='{}'", s(username));
            response.sendRedirect("/login?error");}
```

WHY THIS FIX WORKS

This fix is effective, because in case of a failed login attempt that does not relate to CAPTCHA, the response will be identical, whether username or password is wrong. This is a significant barrier, when brute force is checking both username and password, especially from dictionaries.

A05:2021 Security Misconfiguration

CWE-693: Protection Mechanism Failure (Content Policy Security Not Set)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application was missing **Content Security Policy (CSP)**. Without it, browser can load any scripts, images etc, regardless of whether they should be there or not. This could create a great risk for **Cross Site Scripting (XSS)**. This topic has been discussed on the **CWE-1336** too.

WHERE THE PROBLEM WAS

SecurityConfig.java

HOW IT IS FIXED

Implement default-deny content policy, and a whitelist that informs browser which sources are trusted. This acts as a 2nd layer of defence.

Content Security Policy (CSP) header is added to the application configuration(SecurityConfig.java):

```
http
    .headers(headers -> headers
        .contentSecurityPolicy(csp -> csp.policyDirectives(
            "default-src 'self'; script-src 'self'; style-src 'self' 'unsafe-
            inline'; frame-ancestors 'none'")))
```

WHY THIS FIX WORKS

This is a more generic reasoning for why this “fix works” for **CWE-1336**, which needed the “*script-src 'self'*” that allowed only JavaScript files that are loaded from their own domain. The full **CSP** acts like a strong, browser enforced firewall giving multiple layers of protection. Even if the developer accidentally makes **XSS** flaw, the code is blocked from running. Other directives like “*frame-ancestors 'none'*” prevent other attacks, e.g. clickjacking, reducing the attack range.

CWE-1021: Improper Restriction of Rendered UI Layers (Clickjacking)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

There was significant vulnerability on the application for Clickjacking. The application didn’t send specific header to prevent it, so the attacker could use an `<iframe>` to make malicious actions or use browser plugins like **Tampermonkey** to make fake buttons or links. The user could think that is clicking to “win the lottery” but instead be triggering a different action.

WHERE THE PROBLEM WAS

SecurityConfig.java

HOW IT IS FIXED

Layered defence tells browsers to not use `<iframe>` from another domain. Thus, we use CSP as primary defence for newer browser and `X-Frame-Options` as a fallback for the old browsers.

Content Security Policy - Modern Defence(SecurityConfig.java):

```
.contentSecurityPolicy(csp -> csp.policyDirectives(
    "frame-ancestors 'none'")) // Blocks all framing
```

X-Frame-Options - Legacy Defence(SecurityConfig.java):

```
.headers(headers -> headers
    .frameOptions(fo -> fo.sameOrigin())) // Allows framing from same origin
only
```

WHY THIS FIX WORKS

Provides layered, browser enforced defence against clickjacking. By using CSP directive, or X-Frame-Options header, browser cannot load the site in a malicious iframe, and clickjacking is neutralized.

CWE-250 & CWE-301: Execution with Unnecessary Privileges (Database Least-Privilege)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

The application was connected to MySQL using root “user”. This is a privileged account that can delete entire tables or change the permissions of users. This violates the “Principle of Least Privilege”, which says that the application should work with minimum permissions to do its job. As colleague Francis Nana noted, the application was using the root user, while a user with lower privileges would suffice.

WHERE THE PROBLEM WAS

application.properties

HOW IT IS FIXED

I created a new dedicated database user with limited permissions:

- User: jimboy3100 (host %)
- Privileges: SELECT, INSERT, UPDATE, DELETE **only** on securityapi.*
- Applied FLUSH PRIVILEGES;

```
DROP USER IF EXISTS 'jimboy3100'@'%';  
CREATE USER 'jimboy3100'@'%' IDENTIFIED BY 'Jimboy31';  
GRANT SELECT, INSERT, UPDATE, DELETE ON securityapi.* TO 'jimboy3100'@'%';  
FLUSH PRIVILEGES;
```

Hardened the initial script for the readme.md:

- Correct **FK-safe delete order** (child → parents), reset AUTO_INCREMENT.
- **Fixed admin insert syntax** (bcrypt hash for admin, proper boolean TRUE).

WHY THIS FIX WORKS

Limits the application to not using DDL commands and not use unnecessary privileges. If the application had been exploited by attackers, like SQL injection, the attacker would have limited permissions. They wouldn't be able to change database structures, access other databases, etc.

CWE-550: Application Error Disclosure

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application was using default Spring Boot error handling, revealing "Whitelabel Error Pages" with stack traces. When system crashed, the user could see a detailed error message, which could contain the internal Java classes.

WHERE THE PROBLEM WAS

application.properties

HOW IT IS FIXED

I catholically intercepted any exception that is not handled and could create an error page, and instead chose to show the user a simple, generic error message with no internal details, while logging the stack traces on the server console.

Disabled Default Error Pages(application.properties):

```
server.error.whitelabel.enabled=false
```

**Created a Global Exception Handler that is annotated with
@ControllerAdvice(GlobalExceptionHandler.java):**

```
@ExceptionHandler(Exception.class)
public String handleGeneralException(Exception ex, RedirectAttributes
redirectAttributes) {
    logger.error("Unexpected error: {}", s(ex.getMessage()), ex);
    redirectAttributes.addFlashAttribute("errorMessage",
        "An unexpected error occurred. Please contact support if the issue
persists.");
    return "redirect:/error";}
```

WHY THIS FIX WORKS

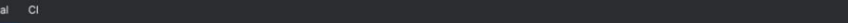
The attacker can no longer cause the application to leak a stack trace⁶. The behaviour is applied across the whole application. The system provides a professional user experience to the client and sends technical messages to the developers.

⁶ The trade-off of this fix is that for applications that are only used internally, and that are not well maintained, the user can't just directly phone call to the developer, telling him that an email is send to him with a printscreen of whitepage's stack traces, and explain the triggering actions and their environment properties that caused the error.

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application was vulnerable because it was built using outdated third-party libraries. Many of these dependencies were older versions that contained publicly known issues (CVE), exposing the application to a wide range of exploits due to unmaintained code. A specific example of this, was identified by Francis Nana, was the use of an outdated **thymeleaf-expression-processor** library. This library had the **CWE-937** flaw, where an attacker could perform a **Cross-Site Scripting (XSS)** exploitation.

WHERE THE PROBLEM WAS



The screenshot shows the JFrog Local tab in a dark-themed IDE. The top bar indicates 'Last scanned at 4 Aug 2025, 12:37:27'. The dependency tree is expanded, showing the following structure:

- pom.xml (/Users/fnana/Development/secure_software_engineering/securityApI/pom.xml)
 - org.apache.tomcat.embed:tomcat-embed-core:10.1.41 (indirect)
 - CVE-2025-48988
 - CVE-2025-49125** (highlighted)
 - org.springframework:spring-web:6.2.7 (indirect)
 - CVE-2025-41234

On the right, a detailed view for CVE-2025-49125 is shown, including a severity score of 7.5 (v3) and tabs for 'What Can I Do', 'CVE Information', and 'Impact Graph'.

[illegible]

HOW IT IS FIXED

- **Updated Spring Boot BOM:** Spring Boot **Bill of Materials** was updated to 3.3.13, which fixes dozens of issues including those in the **Spring Framework**, **Hibernate**, **Jackson**.
- **Made Explicit Security Overrides:** Fixed explicit security components to their latest patches, e.g. **spring-security-crypto** to 6.3.9.

- **Pinned All Other Dependencies:** Reviewed and pinned the versions of all important libraries, to their newest versions, e.g. **mysql-connector-j** to **8.4.0** and **commons-lang3** to **3.18.0**.
- **Removed Unnecessary Starters:** Removed dependencies that were not necessary.

Updated the dependencies(pom.xml):

```
<properties><java.version>21</java.version>
  <spring-boot.version>3.3.13</spring-boot.version>
  <mysql.connector.version>8.4.0</mysql.connector.version>
  <apache-commons-lang3.version>3.18.0</apache-commons-lang3.version>
</properties><dependencies><dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-crypto</artifactId>
  <version>6.3.9</version>
</dependency></dependencies>
```

WHY THIS FIX WORKS

Updating to newer official patches significantly reduces the risk from wide a range of known attacks and makes the project more stable.

CWE-770 & CWE-190: Resource Exhaustion and Integer Overflow in Apache Tomcat IDENTIFIED BY KISHORE M., DURING SECURITY REVIEW

Snyk identified that my application was using an outdated version of **Apache Tomcat**, which was vulnerable to **Denial of Service (DoS)** attacks. The *org.apache.tomcat.embed:tomcat-embed-core* dependency has 3 attack vectors:

- **HTTP/2 Connections (CWE-770):** Server didn't limit the number of requests sent on HTTP/2 connections.
- **Multipart Requests (CWE-770):** Server didn't have limits on number of parts for multipart requests that are used for uploads.
- **Integer Overflow (CWE-190):** Server was vulnerable to overflows of integers, which could allow attacker to bypass security checks like file size.

WHERE THE PROBLEM WAS

pom.xml

HOW IT IS FIXED

Updated dependency(pom.xml):

```
<dependency>
```

```
<groupId>org.apache.tomcat.embed</groupId>  
<artifactId>tomcat-embed-core</artifactId>  
<version>10.1.43</version>  
</dependency>
```

Library *tomcat-embed-core*, updated to a modern patched one.

WHY THIS FIX WORKS

Version **10.1.43** of *tomcat-embed-core*, has patches for both the HTTP/2 (**CVE-2025-53506**) and multipart request vulnerabilities (**CVE-2025-48988**). It also, it includes robust input validation to prevent integer overflows (**CVE-2025-52520**).

CWE-521 & CWE-1391: Use of Weak Credentials

The application lacked a mechanism that would require users to only register their accounts if their password was strong enough to not be guessed. Also, as Kishore M. noted, the application had weaknesses because database users were using high privilege access to the database.

WHERE THE PROBLEM WAS

register.html, customerService.java

HOW IT IS FIXED

Enforced Password Strength in Code(PasswordPolicy.java): This class defines a strict set of rules for passwords (3-12 characters length, including one uppercase lowercase, a number and a symbol). This policy is enforced in **CustomerService** each time a new user has been tries to be created. If the password is weak, registration is rejected.

```
public static boolean isStrong(String pw, String username, String email) {
    if (pw == null) return false; // Part of PasswordPolicy.java
    String p = pw.trim();
    if (p.length() < 3 || p.length() > 12) return false;
    if (!p.matches("(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[!A-Za-z0-9]).*"))
return false;
    /* ... */ }
}
```

Implemented a Least-Privilege Database User: As described on **CWE-250**, the usage of root account has been stopped, and a new user *jimboy3100*, grants access with minimal SELECT, INSERT, UPDATE and DELETE permissions.

```
-- Create the user with your desired password. application.properties should
have the same password.
CREATE USER 'jimboy3100'@ '%' IDENTIFIED BY 'Jimboy31';
-- Grant only the minimal privileges required by the application
GRANT SELECT, INSERT, UPDATE, DELETE ON securityapi.* TO 'jimboy3100'@ '%';
-- Apply changes
FLUSH PRIVILEGES;
```

WHY THIS FIX WORKS

When the password was made with 8 characters length and strict policy, there were no successful brute-force attacks conducted by OWASP **ZAP** on my tests. Relying only on a user's choice for password for their password, makes some accounts vulnerable. Now, an attacker who succeeds with a SQL injection, wouldn't have full authority on the database anymore.

CWE-345: Insufficient Verification of Data Authenticity

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

WHERE THE PROBLEM WAS

SecurityConfig.java, login.html, register.html, checkout.html and all admin forms

HOW IT IS FIXED

Enabled CSRF Protection(login.html, register.html, checkout.html, and all the admin forms): Included hidden `_csrf.token` field, which is required to request any data changes.

```
<form th:action="@{/admin/books/add}" method="post" th:object="${newBook}">
  <input type="hidden" th:name="${_csrf.parameterName}"
th:value="${_csrf.token}"/>
```

Implemented Session-Bound Operations(CartController.java, CartItemService.java, CartItemRepository.java): As described on the **CWE-639** section on **IDOR**, there was a refactoring on all functions that are sensitive, like cart updates to be "session-bound". All these operations are tied with the user's session; hence system verifies the user which is trying to change them.

```
@Transactional //Part of CartItemService.java
public void updateQuantityOwned(Long cartItemId, int quantity, Customer
customer) {
    // Customer's ID to ensure ownership
    CartItem cartItem = cartItemRepository
        .findByIdAndCustomer_Id(cartItemId, customer.getId())
        .orElseThrow(() -> new CartItemException("...not found for this
user."));
    /* ... */ }
```

Added Server-Side Validation of Identifiers: Application makes server-side check to validate the client identifiers, e.g. for adding a book to a cart, the server verified that the `bookId` exists in the store.

```
@Transactional
public void addToCart(Customer customer, Long bookId, int quantity) throws
BookNotFoundException, CartItemException {
    // checks if the bookId sent by the client is a real book in the database.
    Book book = bookService.getBookById(bookId);}
```

WHY THIS FIX WORKS

This makes the application more secure because it provides a multi-layered fix, that it does not blindly trust all the requests. The **CSRF** prevents forged requests from being changed. Combining the session to sensitive operations guaranteed that they are legit and been asked by a real user.

CWE-620: Unverified Password Strength

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

Application didn't have password strength enforcement, on both server and client side. Most critically, on the server side **CustomerService.java** and **CustomerController.java** would accept any password, e.g. "password" or "12345", as long as it's not empty, making the user account vulnerable to attacks.

```
57 @NotBlank(message = "Password is required")
58 private String password;
```

WHERE THE PROBLEM WAS

CustomerService.java

HOW IT IS FIXED

Make an end-to-end password policy on client and server.

Server-Side Policy (PasswordPolicy.java): It has a denylist of common passwords and makes sure that password contains 3-12 characters which must contain uppercase letter, lowercase letter, a number and a special symbol.

```
public static boolean isStrong(String pw, String username, String email) {
    if (pw == null) return false; // Part of PasswordPolicy.java
    String p = pw.trim(); if (p.length() < 3 || p.length() > 12) return false;
    if (!p.matches("(?=.*[a-z])(?=.*[A-Z])(?=.*\\d)(?=.*[A-Za-z0-9~!@#$%^&*]).*"))
return false;
    if (COMMON.contains(p.toLowerCase())) return false; // Denylist check
    // ...
    return true;}

```

Server-Side Enforcement(CustomerService.java): saveCustomer method calls PasswordPolicy.isStrong, before hashing or saving. If the password is weak, it throws the specific error message.

```
public void saveCustomer(Customer customer) {
    if (!PasswordPolicy.isStrong(customer.getPassword(), ...)) {
        logger.warn("Weak password rejected for username={}",
s(customer.getUsername()));
    }
}

```

```
throw new IllegalArgumentException(PasswordPolicy.requirements());}}
```

Client-Side Feedback (register.html & register.js): Password strength meter added to registration page, and a register.js file for the logic “Weak, Fair, Strong”.

Enter address

Phone Number

Email

Enter email

Password

....

Strength: Fair 3–12 chars, upper/lower/digit/symbol

Enter the text from the image

880ZP Refresh

WHY THIS FIX WORKS

Good user experience is combined with security that is hard to be breached. The most important part is that **CustomerService.java**, will not accept the account creation if the password is weak.

A08:2021 Software and Data Integrity Failures

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

CWE-494: Download of Code without integrity checks

Application had the risk of downloading and executing malicious code, because:

1. **Maven Dependencies (pom.xml):** Project was using third-party libraries without any version control enforcement.
2. **Front-End Libraries (.html templates):** Using public **Content Delivery Network (CDN)**, for using **jQuery** or **Bootstrap** libraries, creates a risk that if they are hacked, it could share the malware on our application.

WHERE THE PROBLEM WAS

pom.xml

HOW IT IS FIXED

To eliminate all runtime trust in third-party sources and enforce integrity checks on the dependencies, we needed to pin front-end libraries and rely on Maven's checksum validation, and for the critical front-end libraries we would stop using them and instead host them locally.

Pinned Versions in Maven (pom.xml): As I described at section **CWE-1104**, I removed general versions (e.g. **latest**) and relied on specific secure versions. This also enabled Maven's behaviour to validate the libraries via checksum.

Relied on Maven's Checksum Validation: When downloading files, it compares its size to the original one.

Pinned Versions in Maven (pom.xml):

```
<dependency><groupId>org.webjars</groupId>
  <artifactId>bootstrap</artifactId>
  <version>5.3.3</version></dependency>
<dependency><groupId>org.webjars</groupId>
  <artifactId>jquery</artifactId>
  <version>3.7.1</version></dependency>
```

WHY THIS FIX WORKS

By pinning all the dependencies, we made sure that the application is always been built with libraries that are not tampered.

A09:2021 Security Logging and Monitoring Failures

CWE-778: Insufficient Logging

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application was bling on security events. This lack of audit was a major vulnerability, e.g. **CustomerService** was missing authentication events and **GlobalExceptionHandler** was missing system failures logging. It was quite impossible to find out an ongoing attack, like a brute force.

WHERE THE PROBLEM WAS

All Service files that were missing logging for crucial incidents.

HOW IT IS FIXED

CustomerService was modified to log failed attempts, and show a *WARN* message to the log, including the sanitized username who attempted the login. As described in section **CWE-550**, where **GlobalExceptionHandler** was implemented, this handler acts as logger for any unhandled exception been shown with *ERROR* message.

Logging in the Authentication Logic(**CustomerService.java**):

```
public boolean authenticateCustomer(String username, String rawPassword) {
    Customer customer =
customerRepository.findByUsername(username).orElse(null);
    if (customer == null) {
        logger.warn("Login failed: user not found {}", s(username));
        return false;}}

```

Centralized Exception Logging(**GlobalExceptionHandler.java**):

```
@ExceptionHandler(Exception.class)
public String handleGeneralException(Exception ex, RedirectAttributes
redirectAttributes) {
    logger.error("Unexpected error: {}", s(ex.getMessage()), ex);}

```

WHY THIS FIX WORKS

This fix creates reliable audit of critical events and gives the ability to understand what is happening to the application.

CWE-117: Improper Output Neutralization for Logs

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Application had a Log injection vulnerability, because it was writing input to the logs that wasn't sanitized. This problem was catholic, and affected classes like **CustomerService**, **LockoutFilter**, **LoginFailureHandler**, and **GlobalExceptionHandler**. The attacker could exploit it by using usernames contains characters like \n that change lines, and others. This would cause misleading entry warning.

WHERE THE PROBLEM WAS

All Service files that were missing logging for crucial incidents.

HOW IT IS FIXED

Create a centralized logic for log sanitation and use it everywhere, ensuring that log data will be clean and cannot be used for manipulation.

Created a LogSanitizer Utility(LogSanitizer.java):

```
public final class LogSanitizer {
    private static final int MAX = 200;
    private LogSanitizer() {} // Prevent instantiation
    public static String s(Object o) {
        if (o == null) return "null";
        String str = String.valueOf(o)
            .replace('\r', '_')
            .replace('\n', '_')
            .replace('\t', '_');
        return str.length() > MAX ? str.substring(0, MAX) + "..." : str;}}}
```

Applied Sanitization Everywhere(e.g. LoginFailureHandler and CustomerService):

```
log.warn("Authentication failed for user='{ }'", s(username))
```

WHY THIS FIX WORKS

Provides defense on log injections in all the application. Sanitizer makes it impossible for the attacker to manipulate the logs; thus, logs can be reliable as source of investigation.

CWE-532: Information Exposure Through Log Files

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

Although, there was no custom direct storing of logs into log files, done by me⁷. But if there was such a functionality, then attackers that could gain access to the server logs, would be able to gather information about user accounts or sensitive information of the system.

WHERE THE PROBLEM WAS

All Service files that were missing logging for crucial incidents.

HOW IT IS FIXED

Adoption of privacy by design approach to logging. Never log sensitive personal data unless necessary. This was performed into 2 stages; Remove any custom logging that would reveal sensitive data and apply sanitation to all user-controlled data before written to logs.

Avoided Sensitive Data Logging(SensitiveRequestLoggingFilter.java): Masks sensitive information like “paymentInfo” before logged. Also performed an audit on the entire application manually, to ensure that no user passwords are revealed.

Introduced and Applied the LogSanitizer(LogSanitizer.java): As we described on the **CWE-117**, it is used when user-controlled data is been used. It sanitizes the username in **LoginFailureHandler**, sanitizing the *loggedInUser* session in **CustomerController’s** logout method, and the exception messages in **GlobalExceptionHandler**.

```
logger.info("Customer '{}' Logout", s(u));
```

WHY THIS FIX WORKS

Useful tool for monitoring, but respects user privacy and reduces the risk of information leak.

CWE-223: Omission of Security-Relevant Information

IDENTIFIED BY FRANCIS NANA, DURING SECURITY REVIEW

The application’s error handling was not helpful and led to a poor and insecure user experience. The problem was on error handling in classes like **CustomerController.java** and **LoginFailureHandler.java**. Simple messages with messages like “Registration failed”) would lead to a confused and bad user experience⁸.

⁷ From what I have learned, servers like Payara keep logs in large files that are used to investigate various incidents.

⁸ Although the assessment from Francis Nana identified the on **CWE-209** for generic messages to user on registration; that heuristic is opposed to this one. Functionalities like **CAPTCHA** and account lockout, make user account brute forces impractical.

WHERE THE PROBLEM WAS

CustomerController.java, LoginFailureHandler.java

HOW IT IS FIXED

Create an error message system that provides clear feedback to the legitimate users. The messaging will differentiate as follows:

Registration messaging: Specific feedback, user friendly messages.

Login messaging: **LoginFailureHandler** was updated to handle generic message for any credential-based failures, as described on **CWE-207**.

Lockout messaging: **LoginFailureHandler** and **login.html** were updated to provide helpful messages for locked accounts, including how many minutes user should wait.

Improved Registration Error Messaging(CustomerController.java):

```
if (customerService.findByUsername(customer.getUsername()) != null) {  
    result.rejectValue("username", "error.customer", "Username already  
exists");  
    return "register";}9
```

Generic Login Error Messaging(LoginFailureHandler.java):

```
@Override  
public void onAuthenticationFailure(...) {  
    if (attemptService.isLocked(username)) {} else {  
        response.sendRedirect("/login?error");} // Generic
```

Specific Lockout Messaging(LoginFailureHandler.java):

```
if (attemptService.isLocked(username)){  
    long mins = attemptService.minutesLeft(username);  
    response.sendRedirect("/login?locked&mins=" + mins); return;}
```

WHY THIS FIX WORKS

This provides the correct balance between security, usability and user friendliness. Attackers cannot gain any useful intelligence from the system.

⁹ Same functionality was implemented for phone number, email address and even password strength.

A10:2021 Server-Side Request Forgery (SSRF)

CWE-918: Server-Side Request Forgery (SSRF)

KEY REQUIREMENT IN THE OFFICIAL ASSIGNMENT BRIEF

This vulnerability is so unique, that I had to deliberately create the functionality that would be exploited, and demonstrate how to defend against it, because it is a key vulnerability requirement for this assignment. The new feature added at `/cart/import-by-url` endpoint in the **CartController.java**. In its initial state, webpage would accept any URL as parameter and redirect, which would cause **Server-Side Request Forgery (SSRF)**. The attacker could abuse the internal IP addresses, like <http://127.0.0.1:8080>, and causing internal exploitation, or just use the server as a proxy for illegal(?) activity.

WHERE THE PROBLEM WAS

Any controller that could handle redirections or communicates with microservices. I demonstrated this specific risk in the **CartController.java**, where `/cart/import-by-url` was created for this assignment.

HOW IT IS FIXED

Created a class containing logic whether the URL is safe(**UrlValidatorUtil.java**), which checks if protocol is http or https, resolves the URL's hostname to an IP, and verifies if the IP is private (link-local) or public.

```
try { // Part of UrlValidatorUtil.java
    InetAddress[] inetAddress = InetAddress.getAllByName(host);
    for (InetAddress addr : inetAddress) {if (!isPublicRoutable(addr))
{return "Target resolves to internal/private address";}} // ...
```

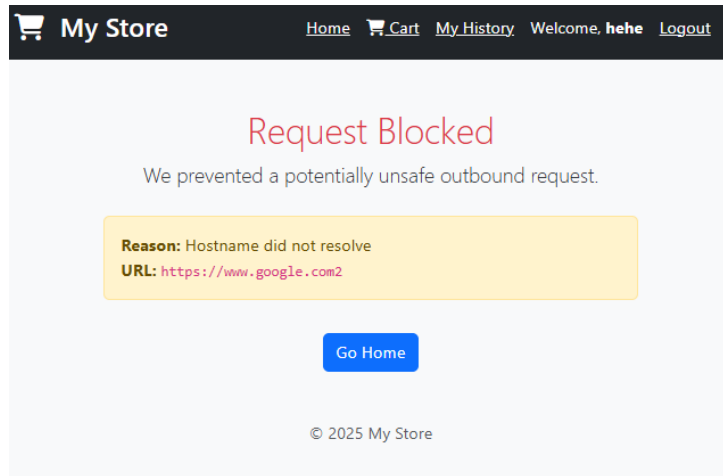
Modified the logic for the `/cart/import-by-url` endpoint(**CartController.java**), where before making a request, it calls `UrlValidatorUtil.explainIfBlocked` method, which returns the reason of the blocked unsafe redirection and shows **ssrf_blocked.html** error page:

```
@GetMapping("/import-by-url")
public String importFromUrl(@RequestParam("sourceUrl") String sourceUrl, Model
model) {
    // First, perform the server-side validation check
    String reason = UrlValidatorUtil.explainIfBlocked(sourceUrl);
    // If the URL is found to be unsafe, then block
    if (reason != null) {
        model.addAttribute("blockedUrl", sourceUrl);
        model.addAttribute("reason", reason);return "ssrf_blocked";}
    // Only if the URL is safe, then proceed
    return "redirect:" + sourceUrl;}
```

Created a new Template(**ssrf_blocked.html**) providing clear feedback to user for the rejection:

```
<p class="lead">We prevented a potentially unsafe outbound request.</p>
<div class="alert alert-warning w-75 mx-auto mt-4 text-start">
```

```
<strong>Reason:</strong>
<span th:text="${reason} ? : 'URL failed security validation.'">URL
failed security validation.</span>
<br/><strong>URL:</strong><code th:text="${blockedUrl} ? : '-'>-
</code></div>
```



WHY THIS FIX WORKS

It performs strict server-side validations **before** making the network requests. *UrlValidatorUtil* is checking the IP address against blocklists of private IP addresses. The attacker cannot scan the internal network and **SSRF** threat is neutralized.

Appendix: Multi-Factor Authentication (MFA) Implementation

As noted on the brief of the assignment, **Multi-Factor Authentication (MFA)** is a security control that is preferred. To accomplish this, I integrated an **Email Based One Time Password (OTP)** system into the application. This provides a 2nd layer of security, because we are determined that a password alone is not enough.

CWE-654 & CWE-308: Reliance on a Single Factor of Authentication

Application initially relied only on a single factor of authentication, which was the combination of username and password. If the attacker somehow managed to steal the password (even with phishing), there would be any 2nd mechanism to prevent the exploitation.

WHERE THE PROBLEM WAS

SecurityConfig.java, LoginSuccessHandler.java

HOW IT IS FIXED

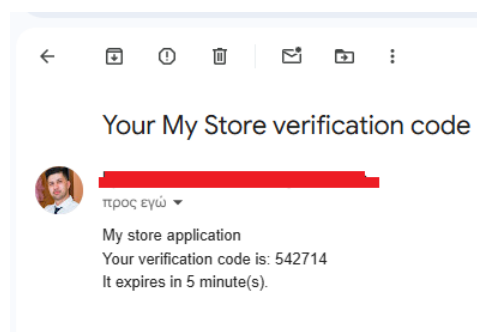
Database & Entity Changes(Customer.java). I have updated the entity to track **MFA** status for each user, the temporary code and the expiry time:

```
@Column(nullable = false)
private boolean mfaEnabled = false;
private String mfaSecretCode;
private LocalDateTime mfaCodeExpiry
```

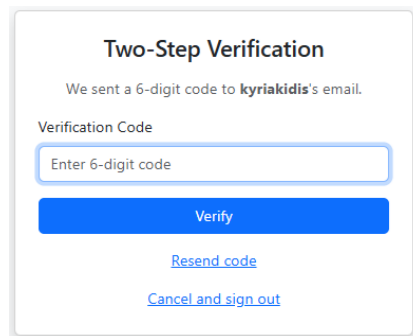
MFA Logic(MfaService.java). This service is used for generating a secure 6-digit code, sending it to the user's email and verifying it when it is applied:

```
public void initiateMfa(Customer customer) {
    String code = String.format("%06d", RNG.nextInt(1_000_000));
    customer.setMfaSecretCode(code);

    customer.setMfaCodeExpiry(LocalDateTime.now(clock).plusSeconds(ttlSeconds));
    customerRepository.save(customer);
    /*...*/ }
```



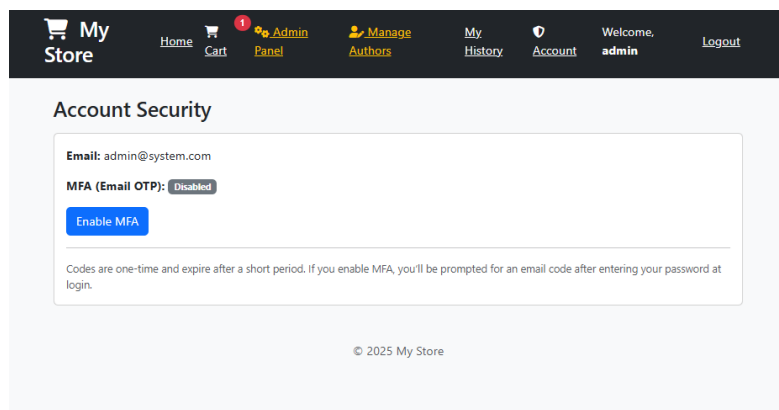
Login Flow Interruption(LoginSuccessHandler.java), that method *onAuthenticationSuccess* is modified such as when a password is correct, handler checks if the **MFA** is enabled. If it is enabled, it stops the process and redirects to **MFA** verification:



The image shows a 'Two-Step Verification' form. At the top, it says 'Two-Step Verification' and 'We sent a 6-digit code to kyriakidis's email.' Below this is a 'Verification Code' section with a text input field containing the placeholder 'Enter 6-digit code'. Under the input field is a blue 'Verify' button. At the bottom of the form are two links: 'Resend code' and 'Cancel and sign out'.

```
@Override
public void onAuthenticationSuccess(...) {
    /*...*/ if (c != null && c.isMfaEnabled()) {
        mfaService.initiateMfa(c);
        session.setAttribute("MFA_USERNAME", username);
        SecurityContextHolder.clearContext();
        response.sendRedirect("/mfa");
        return; // Stop the login process here} /*...*/ }
```

User Interface - Controllers and HTML(MfaController.java, AccountController.java classes, and two new HTML templates mfa_verify.html and account_security.html), that provide a user interface for entering the **OTP** code and enabling or disabling **MFA** on account settings. Customer *admin/admin* has **MFA** disabled for the sake of assessing the exercise.



The image shows the 'Account Security' page of a web application. The top navigation bar includes links for 'Home', 'Cart', 'Admin Panel', 'Manage Authors', 'My History', 'Account', and 'Logout'. The 'Account' link is highlighted, and the user is logged in as 'admin'. The main content area is titled 'Account Security' and shows the email 'admin@system.com'. Below this, it says 'MFA (Email OTP): Disabled' with a blue 'Enable MFA' button. A note at the bottom states: 'Codes are one-time and expire after a short period. If you enable MFA, you'll be prompted for an email code after entering your password at login.' The footer shows '© 2025 My Store'.

Address

Enter address

Phone Number

Email

Enter email

☐ Enable Two-Factor Authentication (Email OTP) **Recommended**

When enabled, after entering your password you'll receive a 6-digit code via email. Enter that code to complete sign-in.

Password

Enter password

Strength: Weak 3–12 chars, upper/lower/digit/symbol

Enter the text from the image

CDJ8P Refresh

WHY THIS FIX WORKS

An attacker who has stolen a user's password cannot use the account anymore. While in corporate environment, **MFA** is mostly not mandatory; accounts with **MFA** enabled are way safer. The application defenses are effective against a wide range of attacks and fully address the **CWE-654** and **CWE-308** requirements.