

Evaluación Técnica - Consultor Senior de Seguridad de Aplicaciones



Consultor:

Erazo Mendoza Jeremy Sebastián
Offensive Security and Exploit Development Engineer

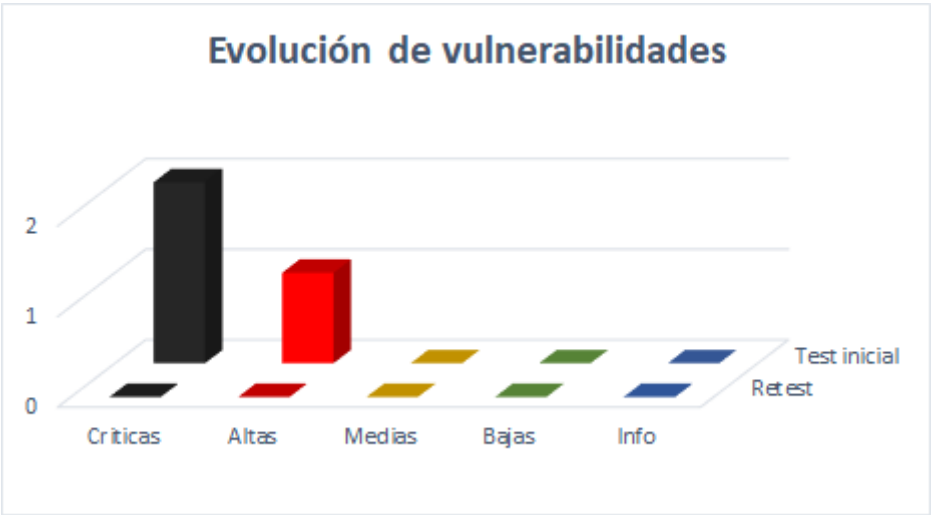
Quito, Junio del 2025

Contenido

Análisis de Vulnerabilidades WebGoat	3
Inyección SQL en endpoint /SqlInjection/attack10.....	3
Acceso indebido a perfiles mediante IDOR (Insecure Direct Object Reference).....	6
Ejecución de comandos arbitrarios mediante deserialización insegura (Insecure Deserialization)	10

Análisis de Vulnerabilidades WebGoat

Durante la revisión de seguridad se detectaron tres vulnerabilidades críticas que representan riesgos significativos para la plataforma. Estas incluyen un acceso no autorizado a perfiles de usuario (IDOR), una inyección SQL en un endpoint clave, y una vulnerabilidad de deserialización insegura con potencial de ejecución remota de código. Cada uno de estos hallazgos compromete la seguridad y estabilidad del sistema, por lo que requieren atención prioritaria y acciones correctivas inmediatas.



Hallazgos	Criticidad	Resultado
Acceso indebido a perfiles mediante IDOR	Alta	Activo
Inyección SQL en endpoint /SqlInjection/attack10	Crítica	Activo
Deserialización insegura con ejecución remota de código (RCE)	Crítica	Activo

Inyección SQL en endpoint /SqlInjection/attack10

Estado General		Activo
Fecha de descubrimiento	18/07/2025	
Datos		
Criticidad	9.8 (Critical)	
Vector CVSS	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H	
Categoría	Inyección SQL (SQL Injection) / Control de acceso e integridad	

Descripción

Se identificó que el sistema procesa sin validación ni parametrización el parámetro `action_string` recibido en el endpoint `/SqlInjection/attack10`. Esto permite que un atacante inyecte código SQL arbitrario en las consultas realizadas al backend.

El análisis de pruebas realizadas muestra que mediante la inclusión de payloads maliciosos (como `%' DROP TABLE access_log;--`) es posible manipular la lógica SQL, permitiendo la ejecución de comandos no autorizados, como la eliminación de tablas en la base de datos.

Esta vulnerabilidad representa un fallo grave en la construcción y manejo de sentencias SQL, resultando en un riesgo severo para la confidencialidad, integridad y disponibilidad de la información almacenada.

Impacto

La explotación exitosa de esta vulnerabilidad puede causar:

- **Confidencialidad:** Exposición o filtración de datos sensibles almacenados en la base de datos.
- **Integridad:** Modificación o eliminación arbitraria de datos, como la eliminación de tablas o registros.
- **Disponibilidad:** Posible interrupción o caída del servicio debido a daños en la base de datos.
- **Privacidad:** Riesgo de incumplimiento normativo por exposición o manipulación de información personal.
- **Auditoría:** Dificultad para detectar y rastrear actividades maliciosas por falta de controles y registros adecuados.

Descubrimiento

Durante la auditoría de seguridad en el módulo `/SqlInjection/attack10` de la aplicación WebGoat, se realizó un análisis exhaustivo del manejo de entradas en las consultas SQL. Se identificó que el parámetro `action_string` es incorporado directamente en sentencias SQL sin aplicar ningún mecanismo de mitigación contra inyección, como consultas preparadas o sanitización rigurosa.

El endpoint procesa el parámetro concatenándolo directamente en la consulta, lo que se evidenció al enviar un payload malicioso diseñado para alterar la lógica del SQL. En particular, el payload `%' DROP TABLE access_log;--` desencadenó la ejecución del comando `DROP TABLE`, confirmando que la construcción dinámica del query es vulnerable a inyección SQL clásica tipo in-band.

```
38  @Test
39  public void tableMissingIsSuccess() throws Exception {
40      mockMvc
41          .perform(
42              MockMvcRequestBuilders.post("/SqlInjection/attack10")
43                  .param("action_string", "'X'; DROP TABLE access_log;--")
44                  .andExpect(status().isOk())
45                  .andExpect(jsonPath("$.lessonCompleted", is(true)))
46                  .andExpect(jsonPath("$.feedback", is(messages.getMessage("sql-injection.10.success"))));
47  }
48  }
49  }
```

La vulnerabilidad se debe a:

- Ausencia total de parametrización en las consultas SQL, lo que permite que entradas maliciosas modifiquen el comportamiento esperado del query.
- Falta de validación o saneamiento contextual del parámetro `action_string` que podría neutralizar caracteres de control o comandos SQL incrustados.
- Permisos excesivos en la conexión a base de datos, ya que la ejecución del `DROP TABLE` indica que el usuario del sistema tiene privilegios administrativos que pueden ser explotados.
- Falta de segregación lógica y física de las responsabilidades de la base de datos, que facilita impactos críticos al no limitar el daño potencial de comandos inyectados.

Se validó la ausencia de mecanismos defensivos como consultas preparadas (prepared statements), procedimientos almacenados con parámetros, ni controles de acceso basados en roles en la base de datos.

Este hallazgo representa un riesgo crítico, pues un atacante remoto puede ejecutar comandos arbitrarios en la base de datos, comprometiendo la confidencialidad, integridad y disponibilidad del sistema entero. Además, la capacidad de ejecución de sentencias DDL (Data Definition Language) como `DROP TABLE` demuestra una exposición agravada por la configuración permisiva del entorno.

Mitigación / Recomendación

La vulnerabilidad de SQL Injection detectada radica en la concatenación insegura de inputs de usuario en consultas SQL dinámicas. Para mitigar esta vulnerabilidad en entornos productivos con requisitos de seguridad robustos, se recomienda una estrategia integral que incluya:

1. Implementación estricta de consultas parametrizadas (Prepared Statements)

- Utilizar APIs de acceso a datos que soporten binding de parámetros, garantizando la separación inequívoca entre lógica SQL y datos.
- Evitar completamente la concatenación o interpolación de cadenas para construir sentencias SQL.

2. Validación y normalización de inputs en capas superiores

- Aplicar validación rigurosa en la capa de presentación o servicio (p. ej., esquemas JSON, validadores basados en expresiones regulares, límites de tamaño).
- Normalizar y sanitizar datos para mitigar vectores secundarios, como ataques por inyección de caracteres unicode o encodings especiales.

3. Aplicación del principio de mínimo privilegio a nivel base de datos

- Configurar usuarios de conexión con permisos estrictamente restringidos — solo `SELECT`, `INSERT`, `UPDATE`, `DELETE` según sea necesario.
- Prohibir expresamente comandos DDL (`CREATE`, `DROP`, `ALTER`) y ejecución de procedimientos que puedan comprometer la integridad estructural.

4. Uso de frameworks ORM con capacidades de prevención de inyección

- Implementar ORMs maduros (p. ej., Hibernate, JPA) que abstraen y encapsulan la generación de consultas, minimizando errores humanos en la construcción manual.

Ejemplo de código corregido avanzado en Java con JDBC y PreparedStatement:

```
1 import java.sql.Connection;
2 import java.sql.PreparedStatement;
3 import java.sql.ResultSet;
4 import java.sql.SQLException;
5
6 public class SecureSqlInjectionHandler {
7
8     private final DataSource dataSource;
9
10    public SecureSqlInjectionHandler(DataSource dataSource) {
11        this.dataSource = dataSource;
12    }
13
14    public void executeSafeQuery(String actionString) throws SQLException {
15        if (actionString == null || actionString.isBlank() || actionString.length() > 128) {
16            throw new IllegalArgumentException("El parámetro actionString no es válido"); // "parámetro": Unknown word.
17        }
18
19        String sanitizedInput = sanitizeInput(actionString);
20
21        final String sql = "SELECT * FROM some_table WHERE some_column = ?";
22
23        try (Connection conn = dataSource.getConnection();
24             PreparedStatement pstmt = conn.prepareStatement(sql)) { // "pstmt": Unknown word.
25
26            // Vinculación segura del parámetro // "Vinculación": Unknown word.
27            pstmt.setString(1, sanitizedInput); // "pstmt": Unknown word.
28
29            try (ResultSet rs = pstmt.executeQuery()) { // "pstmt": Unknown word.
30                while (rs.next()) {
31                    // Procesar resultados con manejo de recursos optimizado // "Procesar": Unknown word.
32                }
33            }
34        }
35    }
36
37    private String sanitizeInput(String input) {
38        // Normalización avanzada: eliminar caracteres invisibles o sospechosos // "Normalización": Unknown word.
39        String normalized = java.text.Normalizer.normalize(input, java.text.Normalizer.Form.NFKC);
40        // Implementar whitelist para caracteres permitidos (ejemplo simplificado) // "Implementar": Unknown word.
41        if (normalized.matches("[\\p{Alnum}\\p{L}\\p{S}\\p{Z}]*")) {
42            throw new IllegalArgumentException("Entrada contiene caracteres no permitidos"); // "Entrada": Unknown word.
43        }
44        return normalized;
45    }
46 }
47
```

Acceso indebido a perfiles mediante IDOR (Insecure Direct Object Reference)

Estado General		Activo
Fecha de descubrimiento	18/07/2025	
Datos		
Criticidad	8.1 (High)	
Vector CVSS	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:L/A:N	
Categoría	Control de acceso horizontal	

Descripción

Se identificó que el sistema permite acceder a perfiles de usuario sin validación adecuada de identidad, mediante el uso de referencias directas a objetos. Este comportamiento es una implementación vulnerable al patrón conocido como IDOR (Insecure Direct Object Reference).

La aplicación expone un endpoint que devuelve los datos del perfil del usuario autenticado, pero también permite consultar directamente por ID, sin verificar si dicho recurso pertenece a quien realiza la solicitud.

En un escenario típico, un atacante autenticado puede manipular el ID de un recurso en la URL o cuerpo del request para acceder a datos de otro usuario legítimo.

Impacto

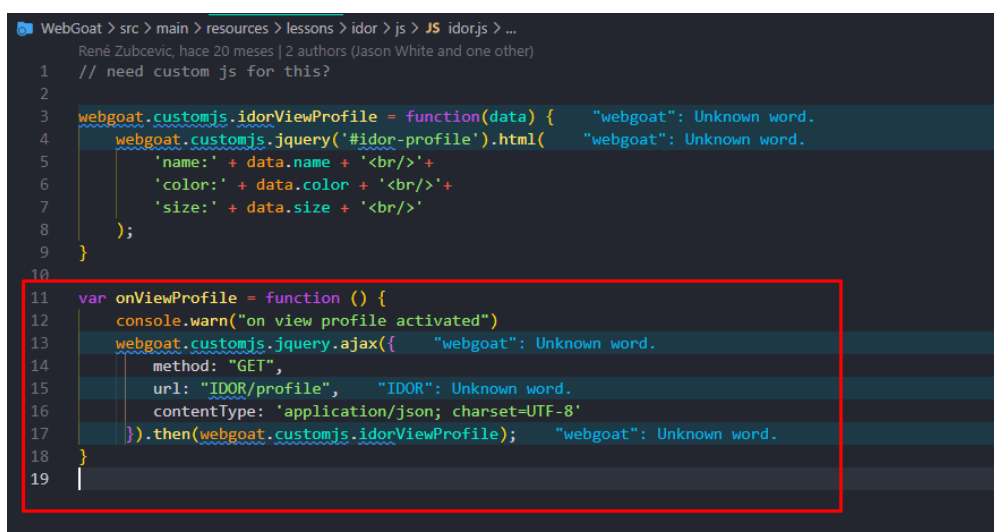
La explotación de este hallazgo compromete varios pilares fundamentales de la seguridad:

- **Confidencialidad:** Permite obtener información personal de otros usuarios autenticados, como nombre, color favorito y talla, sin autorización.
- **Integridad:** Facilita la modificación de perfiles ajenos al alterar solicitudes PUT hacia endpoints vulnerables como `/profile/{id}`.
- **Privacidad:** La exposición de datos personales de múltiples usuarios representa un incumplimiento de normativas como GDPR o equivalentes.
- **Auditoría deficiente:** No se genera ningún registro de actividad que permita evidenciar accesos indebidos.

Descubrimiento

Durante el análisis del módulo IDOR en la plataforma WebGoat, se identificó una funcionalidad vulnerable que permite el acceso no autorizado a información sensible de otros usuarios, lo cual es representativo de un fallo de control de acceso directo sobre objetos internos.

Específicamente, en el archivo `idor.js` se define la función `onViewProfile`, la cual realiza una solicitud HTTP tipo GET al endpoint `IDOR/profile` sin incluir ningún mecanismo de validación de identidad o autorización en el lado del cliente:



```
WebGoat > src > main > resources > lessons > idor > js > JS idorjs > ...
René Zubcevic, hace 20 meses | 2 authors (Jason White and one other)
1 // need custom js for this?
2
3 webgoat.customjs.idorViewProfile = function(data) { "webgoat": Unknown word.
4   webgoat.customjs.jquery("#idor-profile").html( "webgoat": Unknown word.
5     'name:' + data.name + '<br/>' +
6     'color:' + data.color + '<br/>' +
7     'size:' + data.size + '<br/>'
8   );
9 }
10
11 var onViewProfile = function () {
12   console.warn("on view profile activated")
13   webgoat.customjs.jquery.ajax({ "webgoat": Unknown word.
14     method: "GET",
15     url: "IDOR/profile", "IDOR": Unknown word.
16     contentType: 'application/json; charset=UTF-8'
17   }).then(webgoat.customjs.idorViewProfile); "webgoat": Unknown word.
18 }
19
```

Esta implementación sugiere que el backend expone directamente recursos sensibles basados únicamente en identificadores, sin aplicar controles de autorización adecuados. Dado que no se realiza ninguna validación sobre el contexto del usuario autenticado, un atacante podría manipular el endpoint (`/IDOR/profile?id=2` por ejemplo, si se le permite enviar un parámetro) para acceder a perfiles de otros usuarios, cayendo en una condición clásica de IDOR.

Este comportamiento refleja una omisión crítica de los controles de acceso basados en objetos, una de las vulnerabilidades más comunes y severas según OWASP, categorizada dentro del riesgo Broken Access Control (OWASP Top 10 - A01:2021).

Mitigación / Recomendación

La vulnerabilidad identificada en el módulo IDOR constituye un defecto de control de acceso en el backend, específicamente una falta de validación de autorización basada

en contexto de usuario autenticado. Para mitigar esta clase de fallos y prevenir condiciones similares en ambientes productivos o de entrenamiento avanzado, se proponen las siguientes acciones estructuradas por capas:

- Validación basada en identidad del sujeto: Toda solicitud que involucre el acceso a recursos identificables (como perfiles de usuario, documentos, archivos, etc.) debe estar protegida por una lógica que verifique si el usuario autenticado tiene autorización explícita para acceder al objeto solicitado.
 - Por ejemplo: si un usuario autenticado solicita `/IDOR/profile?id=2`, el backend debe verificar que el ID del perfil solicitado corresponde con el del usuario autenticado (`req.user.id === req.query.id`).
- Evitar confiar en datos manipulables desde el cliente (como IDs en la URL o el body del request) para determinar el acceso. Esta lógica debe estar exclusivamente en el servidor.

Eliminar la exposición directa de identificadores internos (como IDs secuenciales) en rutas públicas. En su lugar, utilizar identificadores opacos o tokens temporales validados contra permisos (por ejemplo, UUIDs o referencias aleatorias con tiempo de expiración).

- Implementar middlewares de autorización en cada endpoint sensible. Si se usa un framework como Express.js, se puede centralizar la verificación mediante middlewares que evalúan el `req.user` contra los datos del recurso solicitado antes de devolver una respuesta.

```
// Ejemplo básico en Node.js      "Ejemplo": Unknown word.
router.get('/profile/:id', authorizeUserAccess, async (req, res) => {
  const user = await db.findUserById(req.params.id);
  if (!user) return res.status(404).json({ error: 'User not found' });
  return res.json(user);
});
```

Para finalizar vamos a construir cómo debería ser el código seguro para el manejo de JWT en una aplicación como WebGoat, basada en Spring Boot (Java), tomando en cuenta buenas prácticas cryptográficas, de validación de tokens, control de acceso y protección ante vulnerabilidades comunes:

Dependencias necesarias:

```
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-api</artifactId>
  <version>0.11.5</version>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-impl</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
<dependency>
  <groupId>io.jsonwebtoken</groupId>
  <artifactId>jjwt-jackson</artifactId>
  <version>0.11.5</version>
  <scope>runtime</scope>
</dependency>
```

Generación Segura del JWT:


```

1 import io.jsonwebtoken.*;
2 import io.jsonwebtoken.security.Keys;
3 import java.security.Key;
4 import java.util.Date;
5
6 public class JwtUtil {
7
8     private static final Key key = Keys.secretKeyFor(SignatureAlgorithm.HS512);
9     private static final long EXPIRATION_TIME = 1000 * 60 * 15; // 15 minutos "minutos": Unknown word.
10
11     public static String generateToken(String username, String role) {
12         return Jwts.builder()
13             .setSubject(username)
14             .claim("role", role)
15             .setIssuedAt(new Date())
16             .setExpiration(new Date(System.currentTimeMillis() + EXPIRATION_TIME))
17             .signWith(key, SignatureAlgorithm.HS512)
18             .compact();
19     }
20
21     public static Jws<Claims> validateToken(String token) throws JwtException {
22         return Jwts.parserBuilder()
23             .setSigningKey(key)
24             .build()
25             .parseClaimsJws(token);
26     }
27 }

```

Validación y Autenticación del JWT:

```

1 @Component
2 public class JwtAuthenticationFilter extends OncePerRequestFilter {
3
4     @Override
5     protected void doFilterInternal(HttpServletRequest request,
6                                     HttpServletResponse response,
7                                     FilterChain filterChain)
8         throws ServletException, IOException {
9
10         String token = resolveToken(request);
11
12         if (token != null) {
13             try {
14                 Jws<Claims> claims = JwtUtil.validateToken(token);
15
16                 String username = claims.getBody().getSubject();
17                 String role = claims.getBody().get("role", String.class);
18
19                 if (username != null && role != null) {
20                     List<GrantedAuthority> authorities = List.of(new SimpleGrantedAuthority("ROLE_" + role));
21                     UsernamePasswordAuthenticationToken auth =
22                         new UsernamePasswordAuthenticationToken(username, null, authorities);
23                     SecurityContextHolder.getContext().setAuthentication(auth);
24                 }
25             } catch (JwtException e) {
26                 response.sendError(HttpServletResponse.SC_UNAUTHORIZED, "Invalid or expired JWT");
27                 return;
28             }
29         }
30
31         filterChain.doFilter(request, response);
32     }
33
34     private String resolveToken(HttpServletRequest request) {
35         String bearer = request.getHeader("Authorization");
36         return (bearer != null && bearer.startsWith("Bearer ")) ? bearer.substring(7) : null;
37     }
38 }
39
40

```

Ejecución de comandos arbitrarios mediante deserialización insegura (Insecure Deserialization)

Estado General		Activo
Fecha de descubrimiento	18/07/2025	
Datos		
Criticidad	9.8 (Crítica)	
Vector CVSS	CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:C/C:H/I:H/A:H	
Categoría	Inseguridad en la gestión de objetos serializados (Deserialización insegura)	

Descripción

Se ha identificado una vulnerabilidad crítica de deserialización insegura en el endpoint /InsecureDeserialization/task de la aplicación, donde el sistema acepta objetos serializados en base64 como parámetro (token) y los deserializa sin aplicar mecanismos de validación ni restricciones de tipo.

Este comportamiento permite que un atacante crafting un payload malicioso con clases arbitrarias (como VulnerableTaskHolder) y comandos embebidos (por ejemplo, sleep, ping, rm, etc.), logrando ejecución remota de comandos (RCE) en el entorno backend vulnerable.

La clase VulnerableTaskHolder permite la ejecución condicional de comandos dependiendo del sistema operativo, exponiendo una ventana de ejecución arbitraria en tiempo de deserialización.

Impacto

La explotación de esta vulnerabilidad compromete la plataforma en múltiples vectores:

- Confidencialidad: Un atacante puede ejecutar comandos que exfiltren información sensible desde el servidor (por ejemplo, leer archivos locales, obtener credenciales, etc.).
- Integridad: Puede modificar archivos, manipular servicios del sistema operativo o alterar la lógica de negocio de la aplicación.
- Disponibilidad: Comandos como sleep o fork bombs pueden usarse para degradar o negar el servicio (DoS).
- Ejecución remota de código (RCE): Se facilita la ejecución de instrucciones arbitrarias en el contexto del servidor backend vulnerable.
- Evasión de seguridad: Al tratarse de deserialización, muchos motores de análisis estático o firewalls de aplicaciones (WAFs) no detectan este vector sin decodificación previa.

Descubrimiento

Durante el análisis estático del código fuente correspondiente al módulo de deserialización (org.owasp.webgoat.lessons.deserialization.DeserializeTest), se

identificó una implementación vulnerable al patrón Insecure Deserialization, que habilita ejecución arbitraria de comandos del sistema operativo a partir de objetos serializados manipulables por el usuario.

Se puede ver que el método `success()` dentro de la clase de pruebas automatizadas ejecuta la siguiente lógica condicional:

```
21 @Test
22 void success() throws Exception {
23     if (OS.indexOf("-win") > -1) {
24         mockMvc
25             .perform(
26                 MockMvcRequestBuilders.post("/InsecureDeserialization/task")
27                     .param(
28                         "token",
29                         SerializationHelper.toString(
30                             new VulnerableTaskHolder("wait", "ping localhost -n 5")))
31                 .andExpect(status().isOk())
32                 .andExpect(jsonPath("$.lessonCompleted", is(true)));
33     } else {
34         mockMvc
35             .perform(
36                 MockMvcRequestBuilders.post("/InsecureDeserialization/task")
37                     .param(
38                         "token",
39                         SerializationHelper.toString(new VulnerableTaskHolder("wait", "sleep 5")))
40                 .andExpect(status().isOk())
41                 .andExpect(jsonPath("$.lessonCompleted", is(true)));
42     }
43 }
44 }
```

Esta porción del código evidencia que el servidor acepta objetos serializados proporcionados por el cliente, codificados en Base64 a través de `SerializationHelper.toString(...)`, además el objeto `VulnerableTaskHolder` contiene instrucciones de ejecución arbitraria de comandos (`sleep 5`, `ping localhost -n 5`), las cuales son inyectadas en tiempo de ejecución desde el input del usuario sin ningún tipo de control, también el endpoint `/InsecureDeserialization/task` acepta este token serializado y lo deserializa en el backend sin validación de tipo, clase, ni contenido. Esto abre la puerta a ataques tipo:

- RCE (Remote Code Execution) si el atacante introduce un objeto malicioso.
- Gadget chains si hay bibliotecas cargadas vulnerables a explotación en tiempo de deserialización.
- Bypass de lógica mediante alteración de atributos del objeto deserializado.

Mitigación / Recomendación

Para mitigar eficazmente la vulnerabilidad de deserialización insegura, se recomienda una defensa por capas que combine validaciones estructurales, restricciones de tipos, aislamiento del contexto de ejecución y, en última instancia, evitar por completo la deserialización de datos provenientes del cliente:

1. Evitar deserialización de objetos arbitrarios

- Eliminar por completo el uso de deserialización directa desde el lado del cliente.
- Reemplazar `ObjectInputStream` por formatos seguros como JSON con validación de esquema (Jackson, Gson, etc.).

2. Implementar `ObjectInputFilter` (Java 9+)

Si es inevitable usar deserialización, aplicar filtros estrictos:

```
ObjectInputFilter filter = ObjectInputFilter.Config.createFilter("org.safe.**,*java.base/*.*");
ObjectInputStream in = new ObjectInputStream(socket.getInputStream());
in.setObjectInputFilter(filter);
```

3. Validar clase permitida manualmente

Si se utiliza una clase como `VulnerableTaskHolder`, verificar explícitamente que el objeto deserializado sea una instancia válida:

4. Firmar y verificar tokens

Utilizar firmas digitales (HMAC o JWT firmados) para validar la autenticidad del contenido enviado por el cliente.

5. Código corregido

Ejemplo de refactorización para eliminar la vulnerabilidad (usando JSON)

```
@PostMapping("/InsecureDeserialization/task")
public ResponseEntity<> secureEndpoint(@RequestBody TaskDTO task) {
    if (!isAllowedCommand(task.getCommand())) {
        return ResponseEntity.status(HttpStatus.BAD_REQUEST).body("Comando no permitido");
    }
    executeCommand(task.getCommand());
    return ResponseEntity.ok().body(Map.of("lessonCompleted", true));
}

// Validación explícita "Validación": Unknown word.
private boolean isAllowedCommand(String command) {
    return List.of("sleep", "ping").contains(command);
}
```