# SSDLC pipeline in Jenkins

## Introduzione

Il progetto consiste nella realizzazione di una pipeline in *Jenkins* nella quale attraverso vari strumenti di analisi statica del codice (**SAST**) e analisi di composizione del software (**SCA**) si analizza un repository Java per identificarne le vulnerabilità e archiviare il codice nel caso in cui i risultati delle scansioni di sicurezza abbiano dato esiti positivi. I principali strumenti utilizzati per le scansioni di sicurezza sono: *SonarQube, SpotBugs, OWASP Dependency-Checker.*

## Configurazione della Pipeline CI/CD

### Descrizione dei passaggi per configurare Jenkins

Per configurare *Jenkins* è stato utilizzato ***Docker*.** Ho definito un'immagine custom per creare un container che contenesse Jenkins. L'immagine custom è composta da 11 layer che configurano l'ambiente nel seguente modo: si recupera l'immagine ufficiale di Jenkins LTS da Docker Hub, all'interno dell'immagine si copia un file denominato *plugins.txt* che contiene tutti i plugin necessari al funzionamento della nostra pipeline e alla preconfigurazione di *Jenkins* (il plugin ***Jenkins Configuration as Code*** si rivelerà essenziale). Dopodiché, attraverso la *Plugins-CLI* si installano le dipendenze contenute nel file *plugins.txt*; si crea una variabile d'ambiente da passare alla JVM (Java Virtual Machine) per indicare a *Jenkins* di saltare il setup wizard iniziale. Ciò ci permette di automatizzare il deployment di *Jenkins* ed evitare di doverlo configurare una volta che il container verrà inizializzato. Copiamo il file ***casc.yaml*** **(file che contiene la configurazione di Jenkins e dei tool che andremo ad utilizzare)** all'interno del nostro container per permettere a Jenkins di leggerlo e configurarsi di conseguenza. Passiamo all'utente root per installare ***Maven, Node.js, jq*** perché sono dipendenze necessarie per il funzionamento di *SonarQube* e altri tool (*jq* viene installato per effettuare il parsing delle risposte JSON delle API). Gli ultimi layer copiano uno script di inizializzazione all'interno del container, garantiscono i permessi di esecuzione per quest'ultimo e lo impostano come entrypoint (ovvero lo script viene eseguito appena il container si avvia). Lo script in questione agisce nel seguente modo: aspetta che il container contente *SonarQube* sia attivo per effettuare delle chiamate al suo servizio di web API per creare un token di autenticazione, un custom *Quality Gate* e un webhook che garantiranno il corretto funzionamento dello scanner all'interno della nostra pipeline (senza questi elementi NON è possibile utilizzare *SonarQube* e il suo Quality Gate all'interno della propria pipeline). Prima di chiudersi esporta il token di autenticazione per permettere al file *casc.yaml* di utilizzarlo e crea un file all'interno di un path specifico del container per indicare che il setup è stato completato in modo che in futuro, quando il container verrà inizializzato, i passaggi appena elencati non verranno ripetuti.

Una volta costruita l'immagine, andremo ad utilizzare il comando *docker compose up* all'interno della radice della cartella del progetto per avviare simultaneamente al container custom di Jenkins un container contenente ***SonarQube*** e un container contenente un ***database Postgres*** (utilizzato dal container *SonarQube*). Il tutto è definito all'interno del file

*docker-compose.yaml.* **Jenkins si configurerà automaticamente grazie al file casc.yaml copiato all'interno dell'immagine custom.** Il file in questione contiene l'authentication token e la configurazione dei tool come *SonarQube Scanner e OWASP Dependency-Checker* per l'utilizzo all'interno delle pipeline, oltre che la configurazione dell'URL del server di *SonarQube*. Una volta che Jenkins sarà pienamente operativo, sarà possibile accedere alla sua UI dal proprio browser (se il container è sulla propria macchina l'URL sarà *http://localhost:8080/*). Cliccare il tasto "*Create a new job"* e creare una **pipeline** per il nostro progetto.

## Strumenti integrati nella pipeline per l'analisi statica del codice

Gli strumenti utilizzati sono: ***SonarQube Scanner, SpotBugs.***

## Funzionamento della pipeline e implementazione del gate di sicurezza

Prima dell'esecuzione di qualsiasi stage della pipeline impostiamo due variabili d'ambiente nella nostra pipeline: *project-key,* variabile generata dinamicamente dal numero di build della pipeline che funge da identificativo univoco per il container *SonarQube*, e *scannerHome,* ovvero il percorso di installazione del Sonar scanner.
La pipeline attraversa i seguenti stage:

- **Check-out:** in questa fase Jenkins recuperare il codice sorgente dalla repository git clonandola attraverso un *git clone*.
- **Build:** attraverso *Maven* si effettua una compilazione del codice (il comando è *mvn clean install).*
- **First SAST Analysis:** analisi statica del codice attraverso *SonarQube.* Si richiama il *Sonar scanner* per effettuare un'analisi del codice sorgente del repository e del codice compilato.
- **First Quality Gate:** si confronta il report dell'analisi appena effettuata con il Quality Gate che abbiamo creato in precedenza. Il custom Quality Gate prevede che il codice abbia **0 bug. Il Quality Gate** e i suoi parametri sono stati creati grazie alle chiamate API verso il servizio di Web API di *SonarQube* nello script di inizializzazione del container di *Jenkins*. È previsto che il codice non superi il quality gate in questa fase della pipeline.
- **Second SAST Analysis:** a differenza della prima SAST Analysis, in questo stage andremo ad effettuare un'analisi statica del codice in un branch differente del repository (branch che conterrà codice con meno vulnerabilità rispetto a prima). Jenkins cambia branch con il comando *git checkout fixed-code* e ricompila il progetto. Dopodiché chiama il *Sonar scanner* e subito dopo *SpotBugs* per effettuare un'analisi più approfondita.
- **SCA Analysis:** stage in cui si analizza la composizione del software attraverso *OWASP Dependency-Check.*
- **Second Quality Gate:** in base al risultato delle analisi del *Sonar scanner,* si decide se interrompere la pipeline preventivamente o no. Nel caso in cui il codice rientri nei criteri stabiliti nel Quality Gate, si potrà procedere all'archiviazione in locale del codice; in caso negativo la pipeline viene completamente fermata e la build si considera fallita. È previsto che il nostro codice ora riesca a superare il quality gate.
- **Archive:** archiviazione degli artefatti del codice e dei report di *SpotBugs e OWASP Dependency*-Check in locale.

- **Notify:** si notificano i risultati delle scansioni di sicurezza ad un ipotetico team di sicurezza. In questa pipeline, per semplificare le cose, ho utilizzato dei semplici *echo* per stampare gli URL dei report dei vari tool; in una situazione reale si potrebbero notificare i dipendenti interessati attraverso una **e-mail, un messaggio Slack, o qualsiasi altra piattaforma di comunicazione a stampo Enterprise.** Per l'utilizzo delle e-mail è necessaria la configurazione di un server *SMTP*, mentre per *Slack* e altre piattaforme è necessario un account dedito a Jenkins e plugin aggiuntivi.
- **Post:** questa fase raccoglie le azioni da eseguire dopo il completamento delle fasi principali della pipeline. Indipendentemente dall'esito della build, Jenkins archivia il report di SpotBugs per consentirne l'ispezione. In caso di fallimento dell'esecuzione della pipeline, Jenkins notifica il team di sicurezza stampando a video dettagli come il nome del progetto, il numero della build e l'URL. Inoltre, invita il team a esaminare i log per ulteriori informazioni.

## Evidenza delle modifiche effettuate al codice

Ho modificato il codice risolvendo le 2 vulnerabili più gravi secondo SonarQube: la prima è relativa a dei possibili memory leak causati da una mancanza di chiusura di una connessione aperta ad un DB in un blocco di codice. Per rimediare al problema, ho aggiunto un try-with-resources statement per gestire automaticamente le risorse. Inoltre, ho aggiunto un messaggio di errore nel caso di eccezioni per un miglior error logging.

Codice originario:

```
40      @Override
41      public Book getBookById(String bookId) throws StoreException {
42          Book book = null;
43          Connection con = DBUtil.getConnection();
44          try {
45              PreparedStatement ps = con.prepareStatement(getBookByIdQuery);
46              ps.setString(1, bookId);
47              ResultSet rs = ps.executeQuery();
48
49              while (rs.next()) {
50                  String bCode = rs.getString(1);
51                  String bName = rs.getString(2);
52                  String bAuthor = rs.getString(3);
53                  int bPrice = rs.getInt(4);
54                  int bQty = rs.getInt(5);
55
56                  book = new Book(bCode, bName, bAuthor, bPrice, bQty);
57              }
58          } catch (SQLException e) {
59
60          }
61          return book;
62      }
```

Codice modificato:

```java
@Override
public Book getBookById(String bookId) throws StoreException {
    Book book = null;
    try (Connection con = DBUtil.getConnection();
            PreparedStatement ps = con.prepareStatement(getBookByIdQuery)) {
        ps.setString(1, bookId);
        try (ResultSet rs = ps.executeQuery()) {
            while (rs.next()) {
                String bCode = rs.getString(1);
                String bName = rs.getString(2);
                String bAuthor = rs.getString(3);
                int bPrice = rs.getInt(4);
                int bQty = rs.getInt(5);

                book = new Book(bCode, bName, bAuthor, bPrice, bQty);
            }
        }
    } catch (SQLException e) {
        throw new StoreException("Error retrieving book by ID: " + e.getMessage());
    }
    return book;
}

```

Questo problema si ripresenta più volte all'interno della stessa classe e anche in una differente. Ho modificato tutte le parti di codice che avevano questa falla.

La seconda vulnerabilità più importante che è stata risolta è un pezzo di codice vulnerabile ad una **SQL Injection.** In un metodo era presente la concatenazione diretta di stringhe in una query SQL, senza nessun tipo di parametrizzazione o di sanificazione dell'input. Ho modificato il codice parametrizzando la query e impostando i valori usando un metodo del *java.sql* package per sostituire i valori di placeholder.

Codice originario:

```java
@Override
public List<Book> getBooksByCommaSeperatedBookIds(String commaSeperatedBookIds) throws StoreException {
    List<Book> books = new ArrayList<Book>();
    Connection con = DBUtil.getConnection();
    try {
        String getBooksByCommaSeperatedBookIdsQuery = "SELECT * FROM " + BooksDBConstants.TABLE_BOOK
                + " WHERE " +
                BooksDBConstants.COLUMN_BARCODE + " IN ( " + commaSeperatedBookIds + " )";
        PreparedStatement ps = con.prepareStatement(getBooksByCommaSeperatedBookIdsQuery);
        ResultSet rs = ps.executeQuery();

        while (rs.next()) {
            String bCode = rs.getString(1);
            String bName = rs.getString(2);
            String bAuthor = rs.getString(3);
            int bPrice = rs.getInt(4);
            int bQty = rs.getInt(5);

            Book book = new Book(bCode, bName, bAuthor, bPrice, bQty);
            books.add(book);
        }
    } catch (SQLException e) {

    }
    return books;
}
```

Codice modificato:

```java
public class BookServiceImpl implements BookService {
    @Override
    public List<Book> getBooksByCommaSeperatedBookIds(String commaSeperatedBookIds) throws StoreException {
        List<Book> books = new ArrayList<>();
        String[] bookIdsArray = commaSeperatedBookIds.split(",");
        StringBuilder placeholders = new StringBuilder();

        // Create placeholders for the parameterized query
        for (int i = 0; i < bookIdsArray.length; i++) {
            placeholders.append("?");
            if (i < bookIdsArray.length - 1) {      You, 2 hours ago • Fix memory leak and SQL injection vulnerabiliti
                placeholders.append(",");
            }
        }

        String getBooksByCommaSeperatedBookIdsQuery = "SELECT * FROM " + BooksDBConstants.TABLE_BOOK
                + " WHERE " + BooksDBConstants.COLUMN_BARCODE + " IN (" + placeholders.toString() + ")";

        try (Connection con = DBUtil.getConnection();
                PreparedStatement ps = con.prepareStatement(getBooksByCommaSeperatedBookIdsQuery)) {

            // Set the values for the placeholders
            for (int i = 0; i < bookIdsArray.length; i++) {
                ps.setString(i + 1, bookIdsArray[i].trim());
            }

            try (ResultSet rs = ps.executeQuery()) {
                while (rs.next()) {
                    String bCode = rs.getString(1);
                    String bName = rs.getString(2);
                    String bAuthor = rs.getString(3);
                    int bPrice = rs.getInt(4);
                    int bQty = rs.getInt(5);

                    Book book = new Book(bCode, bName, bAuthor, bPrice, bQty);
                    books.add(book);
                }
            }
        } catch (SQLException e) {
            throw new StoreException("Error retrieving books by comma-separated IDs: " + e.getMessage());
        }

        return books;
    }
```

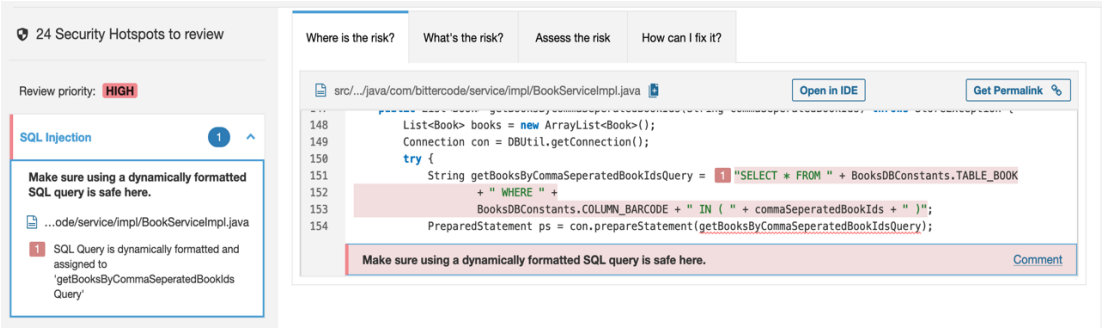
# Analisi delle vulnerabilità

I risultati dei vari report dei tool sono i seguenti:

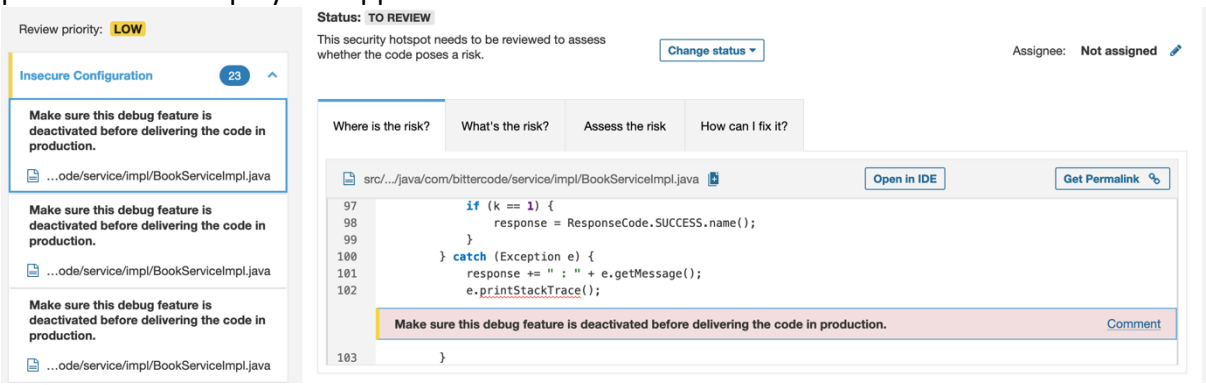
- **SonarQube:** 9 bug trovati con alta probabilità di bloccare l'applicazione, 0 vulnerabilità identificate automaticamente, 24 security hotspots (codice che deve essere verificato manualmente per determinare se insicuro o meno) – 1 hotspot con priorità alta e 23 con priorità bassa; 99 code smells (pratiche scorrette nello sviluppo software che potrebbero portare a problemi più profondi).
- **Spotbugs:** 35 avvisi – 26 relativi a pratiche scorrette nella scrittura del codice, 6 relativi a codice malevolo, 1 relativo alla sicurezza; i rimanenti avvisi sono di bassa importanza.
- **OWASP Dependency-Check:** 8 dipendenze vulnerabili, 36 vulnerabilità trovate – 4 vulnerabilità critiche, 19 vulnerabilità con priorità alta, 13 vulnerabilità con priorità media.

Analizziamo 10 vulnerabilità casuali identificate:

1. **Codice soggetto a SQL Injection:** un malintenzionato potrebbe cercare di concatenare alle query SQL valori anomali per esfiltrare o manipolare dati dal database senza accesso autorizzato. Nell'OWASP Top 10 (2021), le injection sono classificate al terzo posto: **A03 – Injection**. Per evitare problemi di SQL Injection si può agire nei seguenti modi: utilizzare query parametrizzate, validare e sanificare gli input esterni, usare un'API o libreria sicura per eseguire le proprie query, utilizzare controlli (LIMIT keyword in SQL) al fine di limitare l'esfiltrazione di dati…




2. **Stacktrace printing:** l'applicazione contiene codice che espone dettagliati stack trace e messaggi di errore all'utente finale. Espondendo informazioni sensibili e dettagli implementativi dell'applicazione, un utente malintenzionato può conoscere più facilmente potenziali vulnerabilità e punti deboli dell'applicazione. Nell'OWASP Top 10 (2021) le security misconfiguration prendono il quinto posto: **A05 – Security Misconfiguration.** Per risolvere questa vulnerabilità è semplicemente necessario rimuovere qualsiasi riga di codice utilizzata durante la produzione per il debugging prima di fare il deploy dell'applicazione.




3. **Codice soggetto a SQL Injection:** una query generata dinamicata, anche se parametrizzata, può essere soggetta ad attacchi SQL injection se l'input dell'utente non viene sanificato o se ci sono errori nella formazioni della query. Questa vulnerabilità rientra al terzo posto dell'OWASP Top 10: **A03 - Injection**. Per risolvere questa vulnerabilità sarebbe bene effettuare una sanitizzazione preventiva degli

input ed utilizzare una libreria per creare le query.

```
150                 placeholders.append(",");
151             }
152         }
153
154         String getBooksByCommaSeperatedBookIdsQuery = "SELECT * FROM " + BooksDBConstants.TABLE_BOOK
155                 + " WHERE " + BooksDBConstants.COLUMN_BARCODE + " IN (" + placeholders.toString() + ")";
156
157         try (Connection con = DBUtil.getConnection();
158              PreparedStatement ps = con.prepareStatement(getBooksByCommaSeperatedBookIdsQuery)) {
```

A prepared statement is generated from a nonconstant String in com.bittercode.service.impl.BookServiceImpl.getBooksByCommaSeperatedBookIds(String)

The code creates an SQL prepared statement from a nonconstant String. If unchecked, tainted data from a user is used in building this String, SQL injection could be used to make the prepared statement do something unexpected and undesirable.

```
159
160             // Set the values for the placeholders
161             for (int i = 0; i < bookIdsArray.length; i++) {
162                 ps.setString(i + 1, bookIdsArray[i].trim());
163             }
```

4. **Esposizione della Rappresentazione Interna:** il codice corrente della classe `Cart` conserva un riferimento a un oggetto mutabile `Book` senza effettuare una copia difensiva. Ciò significa che un malintenzionato potrebbe modificare l'oggetto `Book` dopo che è stato passato alla classe `Cart`, compromettendo potenzialmente la sicurezza e l'integrità dell'applicazione. Per risolvere questa vulnerabilità, è opportuno effettuare una copia difensiva dell'oggetto mutabile sia nel costruttore che nei metodi setter e getter, assicurando così che le modifiche all'oggetto esterno non influenzino l'oggetto interno memorizzato nella classe `Cart`.

```
5  public class Cart implements Serializable {
6
7      private Book book;
8      private int quantity;
9
10     public Cart(Book book, int quantity) {
11         this.book = book;
```

new com.bittercode.model.Cart(Book, int) may expose internal representation by storing an externally mutable object into Cart.book

This code stores a reference to an externally mutable object into the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Storing a copy of the object is better approach in many situations.

```
12         this.quantity = quantity;
13     }
14
15     public Book getBook() {
16         return book;
17     }
18
19     public void setBook(Book book) {
20         this.book = book;
21     }
```

5. **Gestione Impropria degli Header HTTP:** nella classe `HttpObjectDecoder.java` di Netty prima della versione 4.1.44, è possibile che un header `Content-Length` sia

accompagnato da un secondo header `Content-Length`, o da un header `Transfer-Encoding`. Questa vulnerabilità può portare a problemi di sicurezza come l'HTTP Request Smuggling, dove le richieste HTTP possono essere manipolate per ingannare server intermedi e compromettere la sicurezza dell'applicazione. Questa vulnerabilità può essere correlata a problematiche di sicurezza della categoria **A06 dell'OWASP Top 10 (2021): Vulnerable and Outdated Components**. Per risolvere questo problema, è fondamentale aggiornare Netty alla versione 4.1.44 o successiva, dove viene risolta la problematica in questione.

webapp-runner.jar (shaded: io.netty:netty:3.5.5.Final) NVD CVE-2019-20445 Critical CWE-444

| | |
|---|---|
| **File Path** | /var/jenkins_home/workspace/java-app/target/dependency/webapp-runner.jar/META-INF/maven/io.netty/netty/pom.xml |
| **SHA-1** | 4b0dfe39f0eff1375bad23519ce115f66200b6b1 |
| **SHA-256** | b8d5040d5a7e06bd24cda786c227a865a6f27ba3448505428425f348d5e9a98d |
| **Description** | HttpObjectDecoder.java in Netty before 4.1.44 allows a Content-Length header to be accompanied by a second Content-Length he ader, or by a Transfer-Encoding header. |
| **References** | THIRD_PARTY_ADVISORY<br>THIRD_PARTY_ADVISORY<br>THIRD_PARTY_ADVISORY<br>THIRD_PARTY_ADVISORY |

6. **Esposizione della Rappresentazione Interna:** nel codice della classe `User`, il metodo `getRoles` restituisce un riferimento diretto alla lista mutabile `roles`, esponendo l'oggetto interno a modifiche non autorizzate da parte di codice esterno. Questa vulnerabilità può essere correlata alla categoria **A06 dell'OWASP Top 10 (2021): Vulnerable and Outdated Components**. Per mitigare questo rischio, è necessario restituire una copia difensiva della lista `roles`, assicurando che le modifiche all'oggetto esterno non influenzino l'oggetto interno memorizzato nella classe `User`.

```
64    public List<UserRole> getRoles() {
65        return roles;
```

com.bittercode.model.User.getRoles() may expose internal representation by returning User.roles

Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object. If instances are accessed by untrusted code, and unchecked changes to the mutable object would compromise security or other important properties, you will need to do something different. Returning a new copy of the object is better approach in many situations.

```
66    }
```

7. **Stacktrace printing:** nel codice della classe `UpdateBookServlet` classe, in caso di eccezioni la classe stampa a video dettagli implementativi dell'applicazione. Ciò può portare utenti malintenzionati a identificare vulnerabilità e punti deboli nel codice. Questa vulnerabilità è correlata ad OWASP **A05: Security Misconfiguration.** Per risolvere il problema è sufficiente rimuove righe di codice utile per il debugging in fase di produzione.

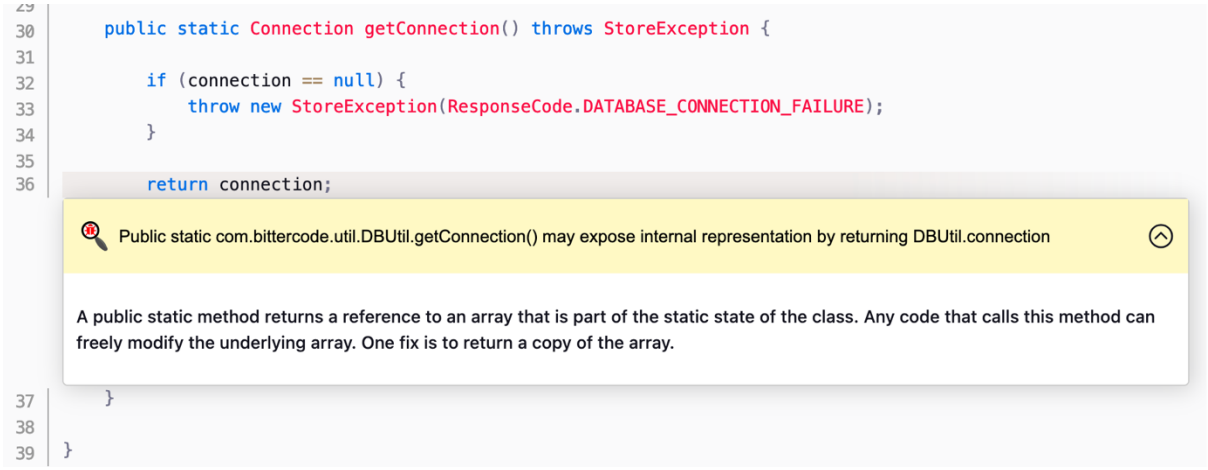
src/main/java/servlets/UpdateBookServlet.java Open in IDE Get Permalink

```
64                  Book book = bookService.getBookById(bookId);
65                  showUpdateBookForm(pw, book);
66              }
67
68          } catch (Exception e) {
69              e.printStackTrace();
```

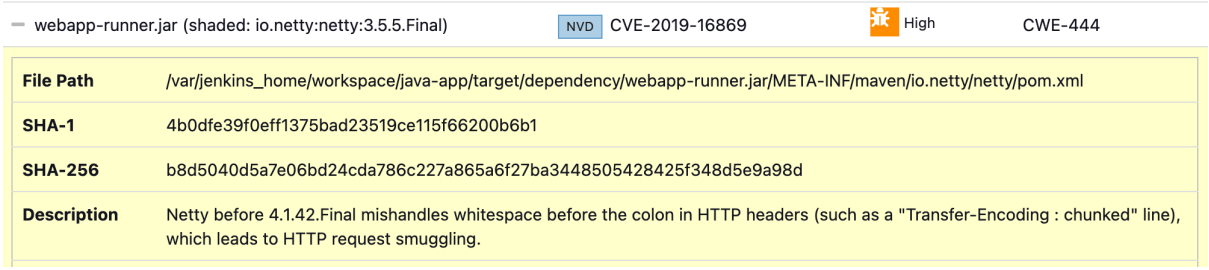
Make sure this debug feature is deactivated before delivering the code in production. Comment

```
70              pw.println("<table class=\"tab\"><tr><td>Failed to Load Book data!!</td></tr></table>");
```

8. **Esposizione della rappresentazione interna:** nella classe `DBUtil` abbiamo il metodo `getConnection()` che espone direttamente la connessione al database, permettendo a qualsiasi codice esterno di modificarla. Una soluzione valida a questa vulnerabilità potrebbe essere restituire una copia dell'oggetto piuttosto che l'oggetto direttamente.




9. **HTTP Request Smuggling (CVE-2019-16869):** Netty prima della versione 4.1.42.Final gestisce in modo errato gli spazi bianchi prima dei due punti negli header HTTP (come nella linea "Transfer-Encoding : chunked"), il che porta a HTTP request smuggling. Questa vulnerabilità è caratterizzata da un rischio elevato e può compromettere la sicurezza dell'applicazione permettendo a un attaccante di manipolare le richieste HTTP per ingannare i server intermedi. Per risolvere questo problema, è necessario aggiornare Netty alla versione 4.1.42.Final o successiva. Questa vulnerabilità può essere correlata alla categoria **A06: Vulnerable and Outdated Components.**




10. **Denial of Service Attack (CVE-2022-3509):** Il textformat nelle versioni core e lite di *protobuf-java* precedenti alla 3.21.7, 3.20.3, 3.19.6 e 3.16.3 può portare a un attacco di denial of service. Input contenenti istanze multiple di messaggi incorporati non ripetuti con campi ripetuti o sconosciuti causano la conversione degli oggetti tra forme mutabili e immutabili, risultando in potenzialmente lunghe pause di garbage collection. Per risolvere il problema è necessario aggiornare la libreria alle versioni menzionate sopra. Questa vulnerabilità può essere correlata alla categoria **A06:**

**Vulnerable and Outdated Components.**

protobuf-java-3.11.4.jar | OSSINDEX | CVE-2022-3509 | High | CWE-20

| | |
|---|---|
| **File Path** | /var/jenkins_home/workspace/java-app/target/onlinebookstore/WEB-INF/lib/protobuf-java-3.11.4.jar |
| **SHA-1** | 7ec0925cc3aef0335bbc7d57edfd42b0f86f8267 |
| **SHA-256** | 42e98f58f53d1a49fd734c2dd193880f2dfec3436a2993a00d06b8800a22a3f2 |
| **Description** | A parsing issue similar to CVE-2022-3171, but with textformat in protobuf-java core and lite versions prior to 3.21.7, 3.20.3, 3.19.6 and 3.16.3 can lead to a denial of service attack. Inputs containing multiple instances of non-repeated embedded messages with repeated or unknown fields causes objects to be converted back-n-forth between mutable and immutable forms, resulting in potentially long garbage collection pauses. We recommend updating to the versions mentioned above. |

## Link e informazioni utili

Componenti del gruppo:

- Ezechiele Spina (ezechiele.spina@studio.unibo.it)

Link repository del progetto:

- https://github.com/zeke-code/secure-ci-cd

Link della repository analizzata:

- https://github.com/zeke-code/onlinebookstore