Report Progetto SSDLC

Alessandro Arrighi

Luglio 2024

Laboratorio di Sicurezza dei Sistemi e Privacy
Corso di Tecnologie dei Sistemi Informatici
Alma Mater Studiorum - Università di Bologna

Contents

1 Introduzione

Il seguente documento specifica tutte le funzionalità e i tool integrati utilizzati all'interno del progetto di SSDLC. Questo è costituito da una Jenkins pipeline CI/CD per lo sviluppo sicuro del software. Come tool utilizzati per l'analisi statica del codice è stato utilizzato Sonarqube e per l'analisi delle dipendenze Dependency Track. Il progetto, si pone come obiettivo quello di automatizzare la fase di verifica della sicurezza del codice, prima che questo venga deployato.

2 Configurazione della Pipeline CI/CD

La pipeline è stata configurata con l'ausilio un server Jenkins hostato su un container. Per l'analisi del codice sono stati utilizzati Sonarqube e Dependcy Track, anch'essi su container. È stato quindi essenziale creare una rete apposita per garantire la comunicazione tra i servizi. La fase di implementazione di tutta l'infrastruttura è stata possibile grazie a Docker utilizzando Docker-Compose.

2.1 Configurazione di Jenkins

Jenkins è stato installato partendo dall'immagine Docker LTS fornita dalla pagina ufficiale di Jenkins. È stato però necessario creare un'immagine personalizza di Jenkins, con un Dockerfile, per poter installare un container dind, così che fosse possibile eseguire nella pipeline altri container. Una volta creata l'immagine, all'avvio dell'istanza è stato specificato il mount del docker.sock all'interno del container. Così facendo è possibile eseguire container all'interno di Jenkins.

Completati questi passaggi il server Jenkins è operativo, necessita solo di completare la configurazione utilizzando l'interfaccia grafica. Come primo passo è richiesto di inserire la password di amministrazione per sbloccare l'istanza, questa viene fornita da Jenkins stesso al suo avvio. Successivamente verranno installati automaticamente tutti i plugin base necessari ed infine si crea un'utenza di accesso al servizio. Completato il set-up guidato, Jenkins è operativo e funzionante.

Per le specifiche adottate in questo progetto, sono stati scaricati alcuni plugin aggiuntivi per il funzionamento di tutta la pipeline: il plugin di Docker, Sonarqube e Dependency Track. Con l'aggiunta di questi tools, deve essere configurato un token di accesso al server di Sonarqube ed al server di Dependency Track. Per poter accedere al repository, è necessario creare un token di accesso ed inserirlo nella sezione dedicata a GitHub all'interno della configurazione di sistema.

2.2 Strumenti integrativi della pipeline

L'analisi del codice è stata possibile grazie l'utilizzo di Sonarqube e Dependency Track.

Sonarqube è uno strumento di analisi della sicurezza del codice automatico. Permette quindi di analizzare un progetto rilevando tutte le vulnerabilità al suo interno, garantendo così di ottenere un codice sicuro. Sonarqube è stato hostato su un container Docker utilizzando l'immagine ufficiale fornita sul repository. Quando il server è funzionante è necessario solamente accedere alla web app, creare le credenziali di amministratore ed effettuare l'accesso. Infine, è necessario creare un nuovo progetto su cui effettuare le analisi; ogni progetto ha un token di accesso specifico che verrà utilizzato per configurare il tool di Sonarqube nel server Jenkins. Unitamente alla creazione del progetto è di fondamentale importanza creare un webhook, inserendo come url di destinazione l'indirizzo di Jenkins. In questo modo il server Sonarqube è in grado di notificare il risultato delle scansioni effettuate. Una volta ricevuto il codice ed eseguito le analisi, all'interno del progetto Sonarqube è possibile controllore tutte le vulnerabilità riscontrate.

Dependency Track è uno strumento di analisi delle dipendenze del codice, che permette di individuare tutti i rischi associati all'utilizzo di codice non sicuro di terze parti. Questo strumento esegue le analisi utilizzando un file chiamato SBOM (Software Bill Of Material), ovvero un file di tipo JSON o XML in cui sono elencate tutte le dipendenze del software che si sta sviluppando. Grazie a questo file, Dependency Track è in grado di effettuare tutte le analisi necessarie per verificare la sicurezza del codice. Dependency Track fornisce due servizi, un server e un frontend, gestiti con due container Docker. Entrambi i container vengono creati partendo dalle immagini Docker fornite nella pagina ufficiale del repository. Una volta operativi, è necessario accedere alla web app fornita dal frontend per creare le credenziali d'accesso. Create le credenziali si deve impostare un token di accesso al servizio ed i relativi permessi per chi vi accede. Completati questi passaggi basta inserire il token di accesso su Jenkins per permettere l'autenticazione al servizio. Una volta inviato il progetto al server questo mostrerà tutte le vulnerabilità del codice.

2.3 Spiegazione della Pipeline

Una volta che il server Jenkins è operativo, è possibile creare un progetto Pipeline. Per prima cosa deve essere impostato il repository in cui sono presenti il progetto, il branch e il percorso dove è salvato il Jenkinsfile. Qui viene poi impostato un trigger che, effettuando un polling, controlla la presenza di un nuovo commit nel repository remoto. Con la ricezione di un nuovo commit, il progetto viene triggerato, il repository viene clonato all'interno di una directory di Jenkins per cominciare l'esecuzione della pipeline. Gli step della pipeline vengono definiti con un file di testo chiamato Jenkinsfile. All'interno di questo, si utilizza il linguaggio Groovy, un linguaggio dichiarativo che permette di definire tutti i passaggi (stage) che Jenkins deve svolgere. Il Jenkinsfile viene salvato all'interno del progetto sul repository, in modo che, una volta clonato il codice, il servizio possa utilizzarlo per eseguire la pipeline.

Gli stage di una pipeline definiscono ogni funzionalità che deve essere eseguita. Il Jenkinsfile di questo progetto è costituito da tre stage:

1. Stage di compilazione del codice e analisi con Sonarqube: l'ambiente di compilazione e testing è stato gestito con un container Maven. Quando questo stage viene eseguito, si produce la build del progetto clonato e i relativi test, per poi inviare il codice al server Sonarqube per eseguire le analisi.

2. Stage del quality gate: in questo stage la pipeline attende una risposta (negativa o positiva) dal server di Sonarqube per verificare se la scansione ha superato i gate di sicurezza impostati. Nel caso di risposta negativa, la pipeline si blocca e ritorna un errore.

3. Stage di analisi delle dipendenze con Dependency Track: completati gli stage precedenti, è possibile utilizzare il file bom generato in fase di compilazione, per inviarlo al server di Dependency Track.

Completati gli stage, vi è un ultimo step nella pipeline che si utilizza per eseguire tutte le funzionalità post pipeline. In questo progetto, viene inviata una notifica ad un ipotetico team di sviluppo e in caso di successo della pipeline, il codice viene salvato in locale e pushato sul repository nel ramo principale.

Con l'esecuzione di questa pipeline si ottiene un ambiente di sviluppo del software automatizzato e molto sicuro. Infatti, ad ogni esecuzione della pipeline è possibile visionare tutti i risultati dei servizi di analisi della sicurezza del codice, così che lo sviluppatore sia sempre aggiornato sullo stato del software.

2.4 Implementazione del gate di sicurezza

Il gate di sicurezza permette di gestire tutte le vulnerabilità del codice e decidere se accettarle mitigarle o risolverle. Effettuato l'accesso sulla web app di Sonarqube è possibile creare un gate, denominato quality gate, in cui vi sono differenti valori che si possono impostare per gestire la sicurezza del codice.

In questo progetto è stato impostato un gate specifico per il numero di vulnerabilità critiche e bloccanti pari a cinque. Questo indica il numero massimo di vulnerabilità oltre il quale l'analisi fallisce. La configurazione di un gate permette di gestire in modo automatico il proseguimento dell'esecuzione della pipeline. Infatti, nella pipeline utilizzata per questo progetto, se il gate di sicurezza dovesse fallire, l'artefatto non viene salvato ed il progetto non inviato al repository remoto.

2.5 Modifiche effettuate al codice

Le due vulnerabilità più critiche trovate dopo l'analisi del codice sono: l'utilizzo di password hard-coded e di oggetti persistenti.

Nella classe "com.jtspringproject.JtSpringProject.controller.AdminController" i metodi "updateProfile" e "profileDisplay" avviano una connessione con il database, qui la password viene scritta staticamente nel codice. Per ovviare a questo problema, è stato utilizzato il riferimento ad un ipotetico file database.password,

in modo tale da ottenere la password da un file di configurazione o in alternativa da una variabile d'ambiente.

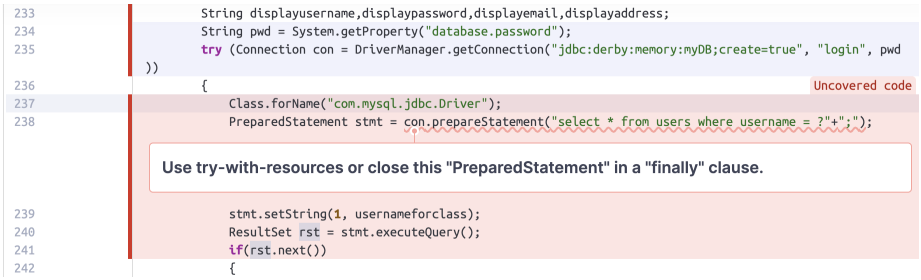
Nella classe "com.jtspringproject.JtSpringProject.controller.UserController" il metodo "newUserRegister" utilizza l'oggetto persistente "user", per inserire i suoi dati all'interno del database. È stata creata la classe "UserDTO" per utilizzare un oggetto che non sia persistente. Quindi all'interno del metodo, viene creato un oggetto di tipo User in cui si inseriscono i dati salvati nell'oggetto "userDTO".

3 Analisi delle vulnerabilità

Di seguito le specifiche delle vulnerabilità riscontrate dalle analisi effettuate al codice.

3.1 Utilizzare un try con le risorse o finally

- Descrizione della vulnerabilità: si presenta quando si utilizzano classi che estendono "Closable" o "AutoClosable" ed è doveroso chiuderle dopo il loro utilizzo.

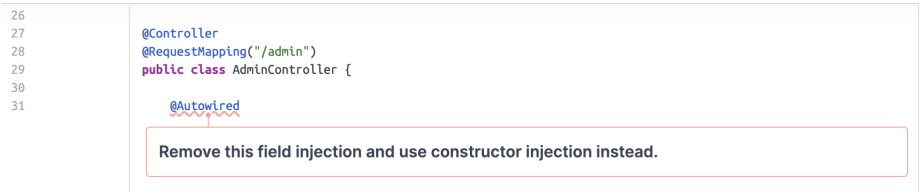


- Gravità e impatti: vulnerabilità ad alto rischio poiché, se sfruttata, compromette il funzionamento dell'applicativo causandone un'interruzione anomala. In alcuni casi è possibile attaccare anche l'infrastruttura sottostante al software.

- Fix del codice: è necessario utilizzare il metodo close della classe all'interno di un blocco finally. In alternativa, è possibile utilizzare un try con le risorse per chiudere la classe che estende "AutoClosable".

3.2 Field Injection

- Descrizione della vulnerabilità: quando si utilizzano framework che permettono di iniettare dipendenze all'interno di una classe, è opportuno implementare il codice con il corretto tipo di "Injection". In questo caso il codice risulta essere vulnerabile dato l'utilizzo di una "Field Injection" con il tag "@Autowierd".



- Gravità e impatti: rende il sistema insicuro poiché può causare malfunzionamenti dovuti all'utilizzo di campi non inizializzati. Utilizzando la field injection si può ottenere durante l'esecuzione, un'eccezione di tipo "NullPointerExecption". Con questo tipo di vulnerabilità all'interno del codice si possono ottenere malfunzionamenti che rendono più insicuro il sistema.

- Fix del codice: modificare il tipo di "Injection". Anziché implementare il codice con una "Field Injection", la modalità corretta da utilizzare è una "Constructor Injection". Così si assicura che quando si crea un nuovo oggetto i campi al suo interno vengano sempre inizializzati dal costruttore, eliminando ogni tipo di inconsistenza.

3.3 Duplicazione di dati

- Descrizione della vulnerabilità: si viene a riscontrare quando lo stesso dato hard-coded è utilizzato in più parti del codice. In questo esempio è stato trovato l'utilizzo di una stessa stringa più volte all'interno dello stesso codice.

- Gravità e impatti: può causare inconsistenze dovute ad errori di trascrizione durante lo sviluppo del codice. In aggiunta, se si dovesse modificare il codice, i dati "hard-coded" renderebbero questo processo estremamente più complicato da realizzare.

- Fix del codice: è buona pratica definire una costante in cui salvare il dato che si vuole utilizzare. Così facendo, se si necessita di modificare il codice, non si rischia di incorrere in errori.

3.4 Campi e metodi con nomi simili

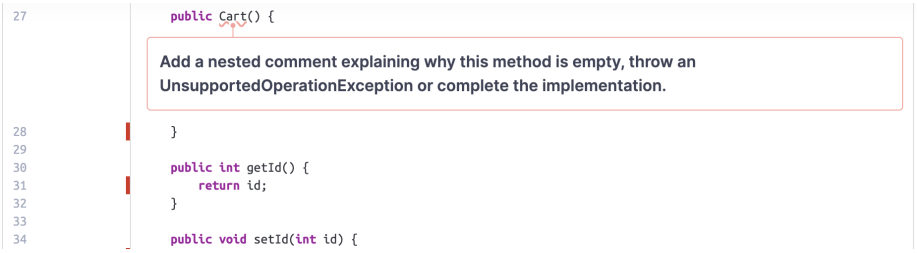
- Descrizione della vulnerabilità: si verifica quando vi sono classi con nomi di metodi o campi simili tra loro, che cambiano ad esempio per una lettera maiuscola/minuscola. Questi tipi di vulnerabilità possono essere riscontrate sia in casi in cui i nomi tra campi e metodi differiscono per una lettera, sia nei casi in cui i nomi si assomigliano tra loro. È inoltre fonte di errore cambiare la visibilità di campi o metodi simili all'interno della classe.

- Gravità e impatti: in questi casi si mette a repentaglio la comprensione del codice durante la lettura. Se il codice fosse modificato, potrebbero insorgere problemi con il sistema. Con questo tipo di implementazione, si rischia di indurre in errore lo sviluppatore quando utilizza i metodi o campi di quella classe erroneamente.

- Fix del codice: denominare campi e metodi con nominativi che siano correlati alla funzione che esercitano e che non siano simili tra loro.

3.5 Metodo vuoto

- Descrizione della vulnerabilità: si verifica quando il corpo di un metodo all'interno di una classe è vuoto. Un metodo vuoto è una pratica scorretta che non apporta nessuna funzionalità aggiuntiva al software, anzi può portare a diversi errori.



- Gravità e impatti: si può facilmente compromettere il lavoro dello sviluppatore quando si utilizza il metodo con la convinzione che questo esegua una funzionalità specifica. Deve essere gesito anche in fase di manutenzione del software, rendendo questa pratica più complicata.

- Fix del codice: è importante evitare pratiche di questo tipo. Nel caso in cui sia necessario adottare una soluzione analoga è fondamentale:
  - invocare un'eccezione al suo interno;
  - indicare con un commento le motivazioni che hanno portato ad implementare il codice con questa struttura.

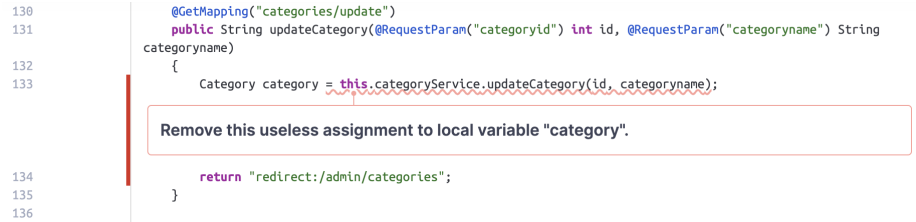
3.6 Logging

- Descrizione della vulnerabilità: si presenta quando non si adottano sistemi di logging o si indirizzano i dati sull'output standard (es. System.out, System.err).

- Owasp top 10: A09:2021 Security Logging and Monitoring Failures
- Gravità e impatti: può causare molti problemi dovuti ad una mancata possibilità di tracciare e analizzare tutte le attività del sistema. In caso di malfunzionamenti del sistema, non è possibile visionare i problemi.
- Fix del codice: è importante quindi utilizzare un accurato sistema di logging e sostituire se presente l'utilizzo di indirizzamento dell'output standard.

3.7 Variabili inutilizzate

- Descrizione della vulnerabilità: pratica scorretta di scrittura del codice in cui si vede la presenza di variabili mai utilizzate o modificate prima del loro utilizzo.



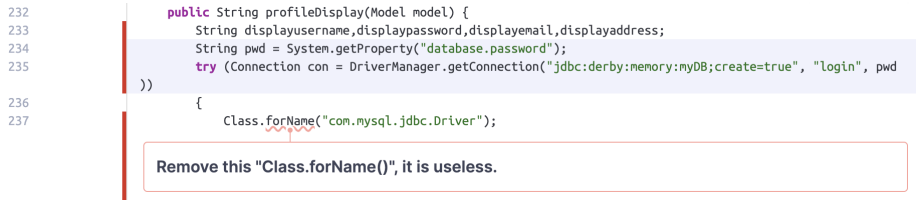
- Gravità e impatti: adottare queste pratiche non porta ad un effettivo aumento dei rischi del sistema. Si ottiene un codice più confuso e difficile da

leggere ed interpretare, con dei possibili incrementi di costi computazionali dell'algoritmo.

- Fix del codice: adottare i principi di sviluppo sicuro del software. In questo caso l'oggetto "category" viene dichiarato e inizializzato ma il suo valore non viene mai utilizzato. Le buone pratiche di sviluppo prevedono la creazione di variabili solo se necessario. È quindi errato creare una variabile senza mai utilizzarla; si può risolvere rimuovendo la variabile inutilizzata.

3.8 Rimuovere funzionalità non necessarie

- Descrizione della vulnerabilità: in generale, è buona prassi adottare funzionalità di dipendenze esterne importate all'interno del codice. Nel caso in cui si dovessero utilizzare metodi specifici che sono stati sostituiti per eseguire la stessa operazione, è buona pratica utilizzare la struttura più aggiornata che la libreria fornisce. In questo caso è stato utilizzato il metodo "Class.forName()" per avviare un "driver" prima di creare una "java.sql.Connection". Con i nuovi "driver" non è più richiesto l'utilizzo di questo metodo prima di avviare la connessione, poiché il "driver" è avviato automaticamente.



- Owasp top 10: A06:2021 Vulnerable and Outdated Components

- Gravità e impatti: l'utilizzo scorretto di dipendenze esterne può portare a contrasti all'interno del codice con altre dipendenze importate, causando possibili malfunzionamenti.

- Fix del codice: è buona prassi eliminare l'utilizzo di queste dipendenze all'interno del codice. In questo caso, poichè non è più necessario l'utilizzo del metodo "Class.forName()", grazie alle nuove versioni dei "driver", è sicuramente buona pratica eliminare la parte di codice che ne fa uso.

3.9 Fornire il tipo parametrico per i generici

- Descrizione della vulnerabilità: si presenta quando non si specifica il tipo di parametro di un generico. Così facendo il compilatore non genera un errore di compilazione poiché verrà effettuato un casting esplicito a run time.

```
43      @Transactional
44      public User getUser(String username,String password) {
45          Query query = sessionFactory.getCurrentSession().createQuery(
"from CUSTOMER where username = :username");
```

Provide the parametrized type for this generic.

```
46          query.setParameter("username",username);
47
48          try {
49              User user = (User) query.getSingleResult();
50              System.out.println(user.getPassword());
```

- Gravità e impatti: può causare problemi al sistema quando il casting esplicito fallisce. Poichè questo processo avviene a run time, il software potrebbe potenzialmente bloccarsi e rendere così il sistema inaffidabile e non sicuro.
- Fix del codice: inserire il tipo parametrico in modo tale che il compilatore possa effettuare il confronto tra i parametri in fase di compilazione. Se il tipo è sconosciuto è comunque una cattiva pratica lasciare il generico vuoto, il modo migliore è inserire la wildcard (?).

3.10 Codice commentato

- Descrizione della vulnerabilità: si presenta quando all'interno del codice vi è una porzione di codice commentato. Questa è una pratica scorretta che rende il sistema vulnerabile. Inoltre, rende più confusionaria la lettura e la modifica del codice.

```
172     //  @GetMapping("carts")
173     //  public ModelAndView  getCartDetail()
174     //  {
```

This block of commented-out lines of code should be removed.

```
175     //      ModelAndView mv= new ModelAndView();
176     //      List<Cart>carts = cartService.getCarts();
177     //  }
178
179     }
180
```

- Owasp top 10: A08:2021 Software and Data Integrity Failures
- Gravità e impatti: forniscono ad un utente malevolo, in possesso del codice sorgente, la possibilità di accedere ad informazioni sensibili o funzionamenti specifici che rendono insicuro il sistema.
- Fix del codice: buona prassi eliminare tutti i codici commentati, all'interno dei sorgenti. Se è necessario accedere a queste informazioni, è possibile farlo grazie ai sistemi di controllo di versione del codice.