Capitolo 1

Garanzia di Sicurezza e Valutazione delle Vulnerabilità

1.1 Contesto e Obiettivi

In un progetto orientato alla dependabilità, la garanzia di sicurezza è trattata come parte dell'affidabilità operativa: difetti che espongono segreti, introducono librerie vulnerabili o lasciano trapelare dettagli di runtime possono diventare rapidamente incidenti di disponibilità (ad esempio, credenziali compromesse, downtime causato da exploit o amplificazione di denial-of-service). L'obiettivo di questo capitolo non è quindi rivendicare una "sicurezza perfetta", ma documentare un processo di hardening concreto e tracciabile applicato alla codebase di *2Chance*.

L'analisi si è concentrata su tre dimensioni di valutazione complementari:

- **Rilevamento Segreti (Secret detection)** a livello di repository (per prevenire l'esposizione accidentale di credenziali);
- **Analisi della Composizione del Software (SCA)** per dipendenze vulnerabili (rischio supply-chain);
- **Static Application Security Testing (SAST)** per pattern di codice rischiosi (programmazione difensiva ai punti di ingresso).

Il lavoro è stato condotto iterativamente seguendo il flusso: *Scansione tramite pipeline CI in GitHub* $\rightarrow$ *Correzione dei problemi* $\rightarrow$ *Push sul repository* $\rightarrow$ *Re-Iterazione*. Tutti gli output, le note e gli artefatti prima/dopo sono archiviati in `reports/evidence/security/`.

1.2 Panoramica sugli Strumenti di Sicurezza

In questo progetto, impieghiamo un approccio di sicurezza multi-livello utilizzando tre strumenti standard del settore, ognuno dei quali affronta una specifica dimensione della sicurezza software. Questa sezione fornisce una panoramica ad alto livello di questi strumenti prima di descrivere la loro integrazione specifica nella nostra pipeline.

1.2.1 GitGuardian (Rilevamento Segreti)

GitGuardian è una piattaforma progettata per rilevare informazioni sensibili (come chiavi API, credenziali database e certificati) che potrebbero essere accidentalmente committate nel controllo sorgente. Scansionando sia la storia di git che i commit in arrivo, previene la "proliferazione di segreti", assicurando che credenziali hardcoded non aggirino i controlli di accesso o trapelino a parti non autorizzate.

1.2.2 Snyk (Analisi della Composizione del Software)

Snyk è uno strumento di sicurezza developer-first che si concentra sulla Software Composition Analysis (SCA). Scansiona le dipendenze open-source (in questo caso, librerie Maven definite nel `pom.xml`) confrontandole con un database di vulnerabilità note (CVE). Aiuta a gestire il rischio della supply-chain identificando librerie obsolete o compromesse e suggerendo percorsi di aggiornamento o patch.

1.2.3 SonarQube Cloud (Analisi Statica)

SonarQube Cloud (precedentemente SonarCloud) è un servizio cloud-based di Static Application Security Testing (SAST) e qualità del codice. Analizza il codice sorgente alla ricerca di bug, vulnerabilità e code smells senza eseguire il programma. Nel contesto di questo progetto, è utilizzato in particolare per imporre standard di codifica sicura nell'implementazione Java, identificando pattern che potrebbero portare a problemi di affidabilità o sicurezza (ad esempio, gestione impropria delle eccezioni).

1.3 Processo, Toolchain e Traccia delle Evidenze

L'analisi di sicurezza è integrata nel workflow di GitHub Actions (`.github/workflows/main.yml`) e si basa sulla seguente toolchain:

- **GitGuardian**: Eseguito immediatamente dopo il checkout del repository per rilevare segreti o credenziali nella codebase. Scansiona i commit (push e pull request) e impedisce il merge di dati sensibili.
- **Snyk**: Eseguito come step dedicato per identificare vulnerabilità nelle dipendenze Maven. La pipeline è configurata per fallire la build se vengono rilevati problemi di gravità *High* o *Critical* (`-severity-threshold=high`).
- **SonarQube Cloud**: Integrato nel ciclo di vita della build Maven (durante la fase `verify`). Esegue l'analisi statica del codice per rilevare bug, code smells e hotspot di sicurezza (inclusa l'analisi del layer controller menzionata successivamente).

Le evidenze sono organizzate per strumento:

- `reports/evidence/security/gitguardian/` (output scansioni, correzioni, walkthrough);
- `reports/evidence/security/snyk/` (scansione baseline, scansione post-fix, modifiche dipendenze);
- `reports/evidence/security/sonar/` (export problemi CSV/JSON e walkthrough rimedio).

1.3.1 Riepilogo Prima/Dopo

La Tabella 1.1 riassume lo stato prima/dopo come catturato dai tre strumenti. I numeri sono presi dai report di baseline e dalle esecuzioni di verifica finale archiviate nelle cartelle delle evidenze.

Tabella 1.1: Riepilogo valutazione sicurezza (prima/dopo) basato sulle evidenze archiviate.

Strumento	Prima	Dopo	Esito
GitGuardian	6 occorrenze (Generic Password)	0	Risolto
Snyk	14 problemi totali	0	Risolto
SonarQube Cloud	96 problemi	0	Risolto

1.4 Meccanismi di Sicurezza nella CI/CD (Checklist Item 0)

Il repository include un workflow GitHub Actions completo (`.github/workflows/main.yml`) che impone controlli di sicurezza prima del deployment. La pipeline definisce un job `build-secure-and-deploy` che esegue sequenzialmente:

1. **Secret Scanning**: tramite action GitGuardian;
2. **SCA**: tramite Snyk CLI;
3. **Build & SAST**: build Maven con analisi SonarQube;
4. **Docker Build**: push dell'immagine solo se tutti gli step precedenti hanno successo.

Questi controlli agiscono come "quality gates" automatizzati, assicurando che il codice che viola le policy di sicurezza (es. segreti esposti o vulnerabilità ad alta gravità) interrompa la build e prevenga il deployment.

1.5 GitGuardian: Rilevamento Segreti e Rimedio

1.5.1 Perché la Scansione dei Segreti

I segreti committati in un repository hanno un raggio d'azione sproporzionato: una singola password o token trapelato può abilitare accessi non autorizzati e aggirare molti altri controlli. La scansione dei segreti mira a prevenire questa classe di incidenti precocemente, prima che le credenziali vengano riutilizzate o propagate attraverso fork e cloni.

1.5.2 Risultati Baseline

La scansione baseline del repository GitGuardian ha rilevato **due istanze** della classe *Generic Password*, per un totale di **sei occorrenze** distribuite su due file:

- `src/prompts/03\_unit\_testing\_implementation.txt` (contenuto documentazione);
- `src/test/java/services/RegistrazioneServiceTest.java` (unit test).

Le evidenze grezze sono archiviate in:

- `reports/evidence/security/gitguardian/ggshield\_repo\_scan.txt` (baseline);
- `reports/evidence/security/gitguardian/ggshield\_repo\_scan\_after\_cleanup.txt` (post-fix).

1.5.3 Strategia di Rimedio e Razionale

La strategia di rimedio è stata conservativa e pratica. Invece di limitarsi a "segnare come sicuro" nella dashboard, abbiamo applicato un approccio configuration-as-code per garantire la riproducibilità:

Modifiche alla Configurazione. È stato introdotto un file `.gitguardian.yaml` per gestire le esclusioni direttamente nel repository. Le modifiche principali includono:

- **Esclusioni Percorsi**: `secret.ignored_paths` è stato configurato per escludere `src/prompts/**` e unit test (`src/test/**`), distinguendo tra segreti reali e dati di test.
- **Pattern Matching**: `secret.ignored_matches` è stato aggiornato per ignorare credenziali note di test (es. `Pass123!`, `validPassword`), riducendo il rumore dai vecchi mock di test.

Questa configurazione ha ridotto il rapporto segnale-rumore, assicurando che le scansioni future si concentrino su leak genuini. Le modifiche applicate e le motivazioni sono documentate in: `reports/evidence/security/gitguardian/security\_analysis\_gitguardian.md` e `reports/evidence/security/gitguardian/gitguardian\_fixes.md`.

1.5.4 Verifica

Dopo la pulizia, GitGuardian ha riportato **0** segreti rilevati nella scansione del repository, come registrato in `ggshield\_repo\_scan\_after\_cleanup.txt`.

1.6 Snyk: Valutazione Vulnerabilità Dipendenze (SCA)

1.6.1 Perché SCA

Le applicazioni moderne ereditano una porzione significativa della loro superficie di attacco da librerie di terze parti. La Software Composition Analysis si concentra su vulnerabilità note (es. CVE) in dipendenze dirette e transitive. Il rimedio tipico consiste nell'aggiornare a una versione patchata o migrare a una coordinata artifact mantenuta.

1.6.2 Risultati Baseline

L'output della scansione baseline (`reports/evidence/security/snyk/snyk\_test.txt`) riporta **11 percorsi vulnerabili** al primo passaggio, che sono stati consolidati nelle note di analisi come **14 problemi totali** attraverso le fasi di rimedio. I componenti interessati si sono concentrati attorno a:

- `commons-io:commons-io` (esaurimento risorse);
- `org.json:json` (problemi relativi a DoS);
- Connettore MySQL e la sua catena transitiva;
- JSTL (`javax.servlet:jstl`) (XXE e namespace legacy).

1.6.3 Rimedio

Sono stati applicati due tipi di correzioni:

- **Aggiornamenti di versione** per artifact mantenuti (es. `commons-io` da `2.10.0` a `2.14.0`, `org.json:json` da `20210307` a `20231013`);
- **Migrazione di artifact** dove la vecchia coordinata era deprecata o non patchata.

Migrazioni Critiche. Particolare attenzione è stata prestata a due risultati critici:

1. **MySQL Connector**: Migrato da `mysql:mysql-connector-java` a `com.mysql:mysql-connector-j` (versione `9.5.0`) per risolvere vulnerabilità di Buffer Overflow nel driver legacy 8.x.

2. **JSTL**: Sostituito il vulnerabile `javax.servlet:jstl` con `org.apache.taglibs:taglibs-s` (`1.2.5`) per risolvere i rischi di XXE Injection mantenendo la compatibilità con le API Servlet.

Le modifiche esatte e le loro motivazioni sono documentate in: `reports/evidence/security/snyk/security\_analysis\_snyk.md` e `reports/evidence/security/snyk/snyk\_fixes.md`.

1.6.4 Verifica

Una scansione finale (`reports/evidence/security/snyk/snyk\_test\_after\_cleanup.txt`) riporta **0 problemi**. Inoltre, è stata eseguita una build completa e un ciclo di test per ridurre il rischio di regressioni legate alle dipendenze (vedi note correzioni Snyk; risultato suite di test: 533 test passati).

1.7 SonarQube Cloud: Analisi Statica di Sicurezza sul Layer Controller

1.7.1 Perché SAST per Servlet

L'analisi statica è particolarmente utile ai punti di ingresso dell'applicazione: nelle architetture basate su servlet, i metodi controller (`doGet`, `doPost`, hook del ciclo di vita) sono una fonte comune di propagazione incontrollata di eccezioni e perdita accidentale di informazioni. Anche quando il bug sottostante è "solo" un'eccezione, l'impatto di sicurezza è spesso pratico: stack trace e dettagli interni di errore possono rivelare la struttura del sistema e abilitare attacchi mirati.

1.7.2 Risultati Baseline

SonarQube Cloud ha segnalato **96 problemi** mappati alla regola `java:S1989` ("Exceptions should not be thrown by servlet methods"). Gli export grezzi sono archiviati in:

- `reports/evidence/security/sonar/sonar\_security\_issues.csv` (iniziale);
- `reports/evidence/security/sonar/sonar\_security\_issues\_2.csv` (metà rimedio);

- `reports/evidence/security/sonar/sonar\_security\_issues\_final.csv` (finale).

1.7.3 Pattern di Rimedio e Motivazione

Il rimedio ha interessato **17 Controller Servlet** (inclusi `LoginServlet`, `AdminServlet` e `RegistrazioneServlet`). La correzione ha comportato un refactoring sistematico per impedire che eccezioni non controllate raggiungessero la pagina di errore di default del container.

Dettagli Implementativi.

- **Gestione Sicura Errori**: È stata introdotta una nuova utility `src/main/java/utils/Resp` per centralizzare le risposte di errore. Utilizza `ResponseUtils.sendError(response, status, message)` per gestire ricorsivamente le IOException, assicurando che un fallimento nel reporting dell'errore non mandi in crash la servlet.
- **Wrapping Standardizzato**: Tutti i metodi `doGet`, `doPost` e `init` sono stati avvolti in blocchi 'try-catch'. Le eccezioni catturate vengono loggate e convertite in risposte di errore generiche 500, mascherando gli stack trace al client.

Il walkthrough completo e la lista file sono documentati in: `reports/evidence/security/sonar/walkthrough\_sonar.md` e `reports/evidence/security/sonar/sonar\_fixes.md`.

1.7.4 Verifica

Gli export finali di Sonar riportano **0 problemi**. Una build Maven ed esecuzione completa della suite di test sono state eseguite dopo il refactoring (533/533 test passati), fornendo ulteriore confidenza che le modifiche preservino il comportamento.

1.8 L'Applicazione Web Non Mostra Vulnerabilità (Checklist Item 2)

Strumenti statici e scanner di dipendenze riducono il rischio, ma non sono sufficienti per affermare che l'applicazione web *in esecuzione* "non mostri vulnerabilità". Questo punto della checklist dovrebbe essere indirizzato con un passaggio finale di verifica a livello di sistema (scansione dinamica e/o controlli manuali mirati) contro un'istanza deployata dell'applicazione.

Claim basato su evidenze (scope). Sulla base delle ri-scansioni finali, la campagna di sicurezza non riporta risultati rimanenti nel suo ambito: GitGuardian rileva 0 segreti nel repository, Snyk riporta 0 problemi noti di dipendenze vulnerabili e SonarQube Cloud riporta 0 occorrenze della regola `java:S1989`. Pertanto, all'interno dell'ambito di secret scanning + SCA + analisi statica, l'applicazione non mostra vulnerabilità rilevate.

Nota di completezza. Per completezza, può essere eseguita una verifica dinamica leggera (DAST) contro un deployment in esecuzione (es. scansione baseline degli endpoint esposti) e archiviata come evidenza aggiuntiva. Questo passaggio è raccomandato come controllo extra a livello di sistema, ma non è richiesto per supportare il claim limitato allo scope sopra citato.

1.9 Minacce alla Validità

Falsi positivi e interpretazione delle regole. Tutti e tre gli strumenti possono produrre falsi positivi o risultati che richiedono interpretazione contestuale. I risultati di GitGuardian su "password generiche" sono particolarmente sensibili alla documentazione e ai test; i report di Snyk possono cambiare con l'evolversi degli advisory; le regole Sonar possono segnalare pattern che sono benigni in contesti specifici. Per questo motivo, il triage umano è una parte esplicita del processo documentato.

Ambiente e riproducibilità. I risultati degli strumenti dipendono dal contesto di esecuzione: configurazioni locali, stato di autenticazione per le dashboard cloud e cronologia del repository. Per mitigare questo, gli output baseline e post-fix sono stati archiviati nel repository. Tuttavia, gli output esatti possono differire nel tempo a causa di database delle vulnerabilità aggiornati (Snyk) o regole dell'analizzatore aggiornate (Sonar).

Aggiornamenti dipendenze e drift comportamentale. L'aggiornamento delle dipendenze migliora la postura di sicurezza ma può introdurre sottili cambiamenti di compatibilità. Il progetto ha mitigato questo rischio ricostruendo ed eseguendo l'intera suite di test dopo il rimedio. Un drift comportamentale residuo è ancora possibile e dovrebbe essere monitorato in modifiche future.

Gap di copertura senza test dinamici. Senza la verifica finale del checklist item 2, la valutazione rimane incompleta al livello di runtime. I risultati

statici e SCA dovrebbero essere interpretati come "rischio ridotto" piuttosto che "nessuna vulnerabilità" per l'applicazione deployata.

1.10 Conclusioni e Passi Successivi

Il lavoro di garanzia della sicurezza ha seguito un loop concreto scansione–correzione–verifica e ha prodotto evidenze archiviate nel repository per ogni strumento. Il progetto ha rimosso: (i) pattern simili a segreti da documentazione/test e limitato i falsi positivi via `.gitguardian.yaml`, (ii) vulnerabilità delle dipendenze tramite aggiornamenti e migrazioni mirate di artifact, e (iii) pattern rischiosi di propagazione eccezioni servlet introducendo una gestione degli errori lato controller coerente e una utility condivisa `ResponseUtils`.

I passi successivi immediati sono:

- completare il checklist item 2 con una verifica dinamica/livello sistema finale e archiviare le evidenze grezze;
- (opzionale) estendere il workflow CI con job di sicurezza leggeri per imporre la scansione come parte della pipeline piuttosto che come passaggio manuale.

1.11 Dati Mancanti

Per finalizzare questo capitolo con la massima audibilita, i seguenti dettagli sarebbero utili:

- Se le scansioni di sicurezza sono già integrate nella CI/CD oltre a build/test (e se sì, quali job e trigger);
- La procedura esatta e lo strumento scelto per il checklist item 2 (DAST/manuale), più output grezzi da archiviare;
- Come viene attivata l'analisi SonarQube Cloud nel tuo setup (esecuzione manuale dashboard vs integrazione CI), in modo che il metodo di esecuzione possa essere documentato con precisione.