Università degli Studi di Milano
FACOLTÀ DI SCIENZE E TECNOLOGIE

DIPARTIMENTO DI INFORMATICA
GIOVANNI DEGLI ANTONI


CORSO DI LAUREA MAGISTRALE IN
INFORMATICA

# STUDIO DI UNA SOLUZIONE DEVSECOPS PER LA SOFTWARE ASSURANCE IN AMBITO IOT

Relatore: Prof. Marco Anisetti
Correlatore: Dott. Antongiacomo Polimeno

Tesi di Laurea di:
Giandonato Inverso
Matr. Nr. 14416A

Anno Accademico 2023-2024

# Indice

# Elenco delle tabelle

# Introduzione

*DevOps is not a goal, but a never-ending process of continual improvement.* [Kim et al., 2016]

Questa frase di Jez Humble riflette perfettamente l'evoluzione delle pratiche **DevOps** [1.1] nel mondo dello sviluppo software moderno, perché l'idea di un "*never-ending process of continual improvement*" implica una mentalità di miglioramento continuo (*continuous improvement*) e apprendimento iterativo.

Questo concetto si basa su pratiche come la *continuous integration* e il *continuous delivery*, che sono diventati elementi fondamentali per accelerare i cicli di sviluppo e migliorare la qualità del software.

Tuttavia, queste pratiche hanno determinato nuove problematiche di sicurezza, portando all'emergere del **DevSecOps** [1.2], un approccio che integra la sicurezza in ogni fase del ciclo di vita del software, a differenza delle metodologie tradizionali, in cui i controlli di sicurezza erano relegati alle fasi finali dello sviluppo.

Eppure nel contesto delle **architetture a microservizi** [A.0.1], le tradizionali metodologie di analisi di sicurezza, consistenti nell'utilizzo di tool di sicurezza [A.0.59] per rilevare vulnerabilità, spesso si rivelano inadeguate.

In particolare, i tool di analisi esistenti, sebbene efficaci nei loro ambiti specifici, spesso mancano di capacità di integrazione e di una visione olistica necessaria per affrontare i rischi emergenti dall'interazione tra *microservizi* [A.0.1] eterogenei, spesso immersi in infrastrutture complesse e ad alta diversità tecnologica.

Questa tesi si propone di colmare questa lacuna sviluppando una metodologia per l'analisi di sicurezza nelle *architetture a microservizi*, progettata per fornire un quadro sistematico e stratificato e in grado di adattarsi e reagire dinamicamente alle specificità di ciascun componente del sistema, garantendo una copertura di sicurezza comprensiva e dettagliata.

Questa metodologia è stata implementata e integrata con **MoonCloud** [3.1], una piattaforma per la valutazione continua di conformità e di *assurance* [1.4] di applicazioni e infrastrutture ICT, ed è stata applicata ad un caso di studio riguardante un sistema *IoT* [A.0.80] complesso denominato **UrbanIoT** [4.1.2], un

software di telegestione degli impianti di illuminazione pubblica e delle smart city.

L'obiettivo ultimo è garantire una copertura di sicurezza completa e dettagliata, in grado di identificare e mitigare i rischi in modo efficace, promuovendo la *resilienza* [A.0.15] e l'affidabilità delle applicazioni moderne.

La tesi è strutturata come segue:

Nel **capitolo 1** viene descritto lo stato dell'arte relativo alle principali metodologie e pratiche attuali nello sviluppo software, con particolare attenzione a *DevOps*, *DevSecOps*, *Security by Design* e *software assurance*, e alle tecniche di analisi e testing del codice.

Nel **capitolo 2** vengono esposte le motivazioni e gli obiettivi della ricerca, evidenziando l'insufficiente applicazione delle tradizionali metodologie di analisi di sicurezza nelle *architetture a microservizi* ed esplicando le motivazioni che hanno condotto alla sviluppo della metodologia proposta.

Viene, inoltre, descritta la metodologia proposta, dettagliando le diverse fasi che includono la definizione dei requisiti di sicurezza, la *gap analysis*, la *threat analysis*, l'analisi della sicurezza per strati, la generazione del report di aggregazione e, infine, l'impostazione del monitoraggio continuo.

Nel **capitolo 3** viene presentata una panoramica delle tecnologie utilizzate, descrivendo *MoonCloud* [3.1] e i tool di sicurezza utilizzati, quali *Gosec* [3.2.1], *Bandit* [3.2.1], *Trivy* [3.2.2] e *Lynis* [3.2.3].

Nel **capitolo 4** viene illustrata un'implementazione della metodologia proposta attraverso l'analisi di un caso di studio riguardante il sistema IoT *UrbanIoT*. Vengono, quindi, descritti i requisiti di sicurezza definiti, le *gap analysis* effettuate, le *threat analysis* condotte e i controlli di sicurezza realizzati attraverso l'uso dei tool di *analisi statica* [1.5] *Gosec* [4.5.1], *Trivy* [4.5.2], *Lynis* [4.5.5] e *Bandit* [4.5.6].

Nel **capitolo 5** vengono presentati i risultati sperimentali ottenuti dall'applicazione della metodologia proposta, tra i quali il numero di vulnerabilità individuate, la loro distribuzione in base alla severità e ai diversi componenti del sistema, nonché i *livelli di rischio* [A.0.37] complessivi e specifici associati alle vulnerabilità identificate.

Nel **capitolo 6** vengono tratte le conclusioni della tesi, evidenziando come la metodologia proposta superi le limitazioni delle tecniche tradizionali di analisi di sicurezza, offrendo un quadro completo e dettagliato delle vulnerabilità e dei rischi associati in *architetture a microservizi* complesse. Vengono inoltre delineati i possibili sviluppi futuri della ricerca.

Infine nell'**appendice A** sono presenti le definizioni relative alla terminologia scientifica e ad altri concetti rilevanti per la comprensione del lavoro.

# Capitolo 1

# Stato dell'arte

Questo capitolo è dedicato allo stato dell'arte relativo alle principali metodologie e pratiche attuali nel campo dello sviluppo software, con particolare attenzione a **DevOps**, **DevSecOps**, **Security by Design** e **software assurance**.

La sezione 1.1 introduce il concetto di *DevOps*, evidenziando come l'integrazione tra *development* e *operations*, insieme all'automazione e alla collaborazione, possa ridurre i tempi di rilascio e migliorare la qualità del software. In particolare, verranno illustrati i principi fondamentali del *DevOps* e le fasi principali di una *pipeline DevOps*.

Successivamente, la sezione 1.2 esplora l'evoluzione del *DevOps* verso il *DevSecOps*, un approccio che integra la sicurezza nell'intero ciclo di vita dello sviluppo del software. Saranno descritti i principi chiave del *DevSecOps* e le differenze rispetto alle pratiche *DevOps* tradizionali, con un focus su come le *pipeline DevSecOps* includano analisi delle vulnerabilità, controlli di configurazione e verifiche di compliance.

La sezione 1.3 è dedicata al concetto di *Security by Design*, un approccio che integra la sicurezza fin dalle prime fasi di progettazione del software. A tal proposito, verranno discusse le pratiche di *secure coding* [A.0.60], *verifica e validazione del software* [A.0.63] e l'importanza dell'*analisi del rischio* [A.0.38] per identificare e mitigare le minacce in modo proattivo.

Infine, la sezione 1.4 descrive vari approcci e tecniche della *software assurance*, che garantisce la qualità, l'affidabilità e la sicurezza dei sistemi software attraverso pratiche di certificazione, formazione, analisi e testing del codice e modellazione delle minacce.

## 1.1 DevOps

**DevOps** è un concetto emergente nel campo dello sviluppo software che mira a ridurre i tempi di rilascio di nuove versioni di applicativi attraverso l'integrazione delle attività di sviluppo (*Development*) e di quelle operative (*IT Operations*) [Srivastav et al., 2023].

Questa metodologia si fonda su due principi cardine: l'**automazione** e la **collaborazione**. L'*automazione* impiega le tecnologie per minimizzare le operazioni manuali, migliorando l'efficienza e la qualità dei processi e riducendo il margine di errore [Bass et al., 2015]. La *collaborazione*, invece, consente di facilitare il flusso di lavoro tra diversi team promuovendo un allineamento agli obiettivi organizzativi [Luz et al., 2018].

*DevOps* si configura come una evoluzione del modello **Agile** [A.0.12], enfatizzando rilasci più frequenti e di maggiore qualità attraverso pratiche di integrazione e distribuzione continua, note come **CI/CD** [Colavita, 2016]. Il **Continuous Integration** (CI) assicura che ogni modifica apportata al codice sia automaticamente testata, identificando rapidamente eventuali errori, mentre il **Continuous Delivery and Deployment** (CD) facilita il rilascio di nuove versioni attraverso distribuzioni automatizzate [Davis and Daniels, 2016].

In particolare, nell'implementazione di pratiche di integrazione e distribuzione continue (CI/CD) all'interno di un ambiente *DevOps*, le **pipeline DevOps** rappresentano un elemento fondamentale.
Una *pipeline DevOps* è una serie automatizzata di passaggi che consentono il flusso continuo del codice sorgente attraverso varie fasi, dalla compilazione al rilascio e al monitoraggio, al fine di favorire l'automazione di attività ripetitive e ridondanti, consentendo un flusso di lavoro più efficiente e affidabile.

Le fasi principali di una *pipeline DevOps* sono:

- **Build**: compilazione del codice sorgente in un *artifact* [A.0.64] eseguibile;
- **Test**: esecuzione di test automatizzati per garantire che il software funzioni correttamente e rispetti i requisiti di qualità;
- **Deploy**: il software viene distribuito in un ambiente di test o produzione, eventualmente includendo il *provisioning di risorse* [A.0.2], la configurazione dell'ambiente e il rilascio dell'applicazione in un ambiente di produzione;

- **Monitoraggio e feedback**: una volta rilasciato, il software viene monitorato per identificare eventuali problemi o degrado delle prestazioni. Il feedback raccolto durante questa fase viene utilizzato per migliorare il processo di sviluppo e distribuzione.

## 1.2 DevSecOps

*La metodologia **DevSecOps** si propone di integrare la sicurezza in ogni fase del ciclo di vita dello sviluppo software, rendendola una responsabilità condivisa dall'inizio alla fine* [RedHat, 2024].

Nelle vecchie metodologie di sviluppo software, i test e i controlli di sicurezza erano relegati alle fasi finali dello sviluppo, tuttavia con l'avvento del *DevOps*, i tempi di sviluppo si sono notevolmente accorciati, rendendo possibili controlli di sicurezza continui e automatizzati lungo l'intero processo di sviluppo [Stol and Fitzgerald, 2022].

In particolare, il paradigma *DevSecOps* si propone di identificare e mitigare in modo proattivo i rischi all'interno dell'intero ciclo di vita dello sviluppo del software attraverso un approccio sistematico e integrato senza ostacolare l'agilità e la velocità delle pratiche *DevOps* esistenti [Rahman, 2016].

Ad esempio nella fase di test di una classica *pipeline DevOps* [1.1] vengono eseguiti esclusivamente test automatici, mentre le *pipeline DevSecOps* sono configurate per includere anche analisi delle vulnerabilità [A.0.30], controlli di configurazione e verifiche di *compliance* [A.0.49].

Da notare che le *pipeline DevSecOps* sono solo una parte di un processo più grande articolato in più fasi, in cui ognuna è pensata per integrare la sicurezza come un elemento centrale del processo. Il fine ultimo è identificare e risolvere rapidamente le vulnerabilità in ogni fase dello sviluppo e migliorare continuamente il prodotto attraverso i feedback e l'analisi dei dati raccolti durante il monitoraggio, riducendo così il tempo e il costo associati alla mitigazione dei rischi di sicurezza post-rilascio.

Le fasi principali di una *pipeline DevSecOps* sono:

1. **Pianificazione:** identificazione dei requisiti di sicurezza e delle metriche di successo [Mohan and Singh, 2018];
2. **Codifica:** applicazione di pratiche di *secure coding* [A.0.60] e *revisione del codice peer-to-peer* [A.0.62] [Mohan and Singh, 2018];
3. **Build:** utilizzo di strumenti di analisi statica e dinamica del codice per rilevare vulnerabilità [Mohan and Singh, 2018];

4. **Testing:** esecuzione di test di sicurezza automatizzati durante la fase di test [Mohan and Singh, 2018];

5. **Rilascio:** implementazione di controlli di sicurezza automatizzati [A.0.41] nel processo di rilascio [Mohan and Singh, 2018];

6. **Monitoraggio:** raccolta continua di dati sull'attività dell'applicazione per rilevare e rispondere tempestivamente a potenziali minacce [Mohan and Singh, 2018].

## 1.3 Security by Design

*Molti dei problemi che minacciano la sicurezza delle applicazioni hanno le loro radici nella progettazione stessa, il che rende fondamentale l'adozione di pratiche di* ***Security by Design****. Questo approccio non solo riduce il rischio di vulnerabilità, ma consente anche di risparmiare tempo e risorse nel lungo periodo.* [CISA, 2023]

Con *Security by Design* (SbD) ci si riferisce alla considerazione della sicurezza come una parte essenziale dei processi di progettazione, sviluppo e manutenzione, piuttosto che una esclusiva correzione post-sviluppo, la quale inizia nelle fasi iniziali di sviluppo di un sistema software e termina con la fase di dismissione [A.0.18], assicurando che i dati sensibili [A.0.48] siano protetti fino al termine dell'uso del sistema [Chattopadhyay et al., 2020].

Uno degli elementi chiave della *SbD* è l'**analisi del rischio** [A.0.38], che deve essere condotta regolarmente per identificare nuove vulnerabilità e minacce emergenti. Ciò permette agli sviluppatori di adeguare le misure di sicurezza in modo proattivo anziché reattivo.

Altre pratiche di *SbD* riguardano il *secure coding* [A.0.60] e la *verifica e validazione del software* [A.0.63], affinché gli errori siano identificati e corretti prima del rilascio del sistema.

L'adozione di questo approccio è particolarmente rilevante in settori dove la sicurezza è critica, come nel settore bancario, nei dispositivi medici, infrastrutture critiche e dispositivi *Internet of Things* [A.0.80], dato che non solo migliora la resilienza dei sistemi contro gli attacchi, ma riduce anche i costi associati alla gestione delle emergenze di sicurezza post-sviluppo.

## 1.4 Software assurance

*La* ***software assurance*** *è il livello di confidenza che il software funzioni come previsto e che sia privo di vulnerabilità, progettate o inserite intenzionalmente o*

*meno nel software durante tutto il suo ciclo di vita* [IICS WG, 2015].

Da notare che la *software assurance* non aggiunge alcun controllo aggiuntivo per contrastare i rischi legati alla sicurezza ma fornisce solo un livello di confidenza che i controlli implementati ridurranno il rischio previsto [ISO, 2008].

In particolare, le pratiche di *software assurance* sono volte a garantire la qualità, l'affidabilità e la sicurezza dei sistemi software.

### 1.4.1 Pratiche di software assurance

#### Certificazione e accreditamento

Insieme di attività formali volte a valutare e confermare che un sistema software soddisfi determinati requisiti stabiliti da enti regolatori o standard internazionali.

La certificazione è solitamente eseguita da terze parti indipendenti, mentre l'accreditamento riguarda l'approvazione di processi e pratiche aziendali da parte di organismi di controllo [IEEE, 2014].

Ad esempio la certificazione **ISO/IEC 27001** è uno standard internazionale che descrive le best practice per avere un sistema di gestione della sicurezza delle informazioni robusto e conforme ai requisiti dello standard, tra le quali figurano l'analisi e la gestione dei rischi, il controllo degli accessi e la protezione dei dati sensibili.

La certificazione richiede un audit [A.0.40] da parte di un ente accreditato, che verifica l'efficacia dei controlli di sicurezza implementati.

#### Formazione ed educazione

Formazione da parte degli sviluppatori in merito a best practise di *secure coding* e tecniche di *software testing* al fine dell'ottenimento di competenze necessarie per sviluppare software sicuri e di elevata qualità [NIST, 2013], al fine di ridurre il rischio di introdurre vulnerabilità e difetti di sicurezza che potrebbero essere sfruttati da attaccanti.

Un esempio pratico di formazione in *secure coding* è la certificazione **Certified Secure Software Lifecycle Professional** erogata da *ISC2*[1] , un ente di formazione internazionale. Questo programma di certificazione copre uno spettro di competenze ampio, tra cui la progettazione sicura, il *secure coding*, il *software testing* e il mantenimento del software, al fine di identificare e mitigare le vulnerabilità del software in ogni fase del suo ciclo di vita.

[1] https://isc2.org

Un altro esempio significativo è la piattaforma di apprendimento **OWASP - Open Web Application Security Project**[2] che offre risorse educative e strumenti per migliorare la sicurezza delle applicazioni web. Inoltre, *OWASP* fornisce esercitazioni pratiche, esempi di codice sicuro e linee guida aggiornate su come affrontare le vulnerabilità di sicurezza più comuni identificate nella classifica *OWASP Top Ten*[3].

### Analisi e test del codice

L'analisi e il test del codice sono processi che utilizzano una combinazione di tecniche avanzate per identificare potenziali difetti, errori, vulnerabilità e per migliorare la qualità e la sicurezza del software.

Tra le tecniche più comuni vi sono l'**analisi statica e dinamica** del codice.

L'*analisi statica* esamina il codice sorgente senza eseguirlo, permettendo di individuare errori di sintassi e potenziali vulnerabilità come i **side channel attack** [A.0.69], ad esempio **buffer overflow** [A.0.70] e **heap overflow** [A.0.71].

Contrariamente, l'*analisi dinamica* richiede l'esecuzione del software per osservare il comportamento del codice in tempo reale, rilevando problemi che possono emergere solo durante l'esecuzione, come le **race conditions** [A.0.72] e i problemi di gestione della memoria.

### Threat modeling e risk assessment

Il **threat modeling** (*modellazione delle minacce*) consiste nell'identificazione e nell'analisi delle potenziali minacce [A.0.29] che possono compromettere la sicurezza di un sistema software, attraverso la creazione di modelli che rappresentano possibili scenari di attacco, identificando le vulnerabilità e i punti deboli del sistema.

Ad esempio, una tecnica popolare è l'approccio **STRIDE** (Spoofing, Tampering, Repudiation, Information disclosure, Denial of Service, Elevation of privilege) [A.0.33], che categorizza le minacce in diverse classi per facilitare l'identificazione di strategie per la loro mitigazione e per la contribuzione alla resilienza agli attacchi, cioè la capacità di un sistema di resistere e recuperare rapidamente dagli attacchi informatici.

Il **risk assessment** (*valutazione dei rischi*) è il processo di identificazione, analisi e prioritizzazione dei rischi associati alle minacce individuate. In particolare, viene calcolata la stima della probabilità che una minaccia si concretizzi e l'impatto potenziale che avrebbe sul sistema. La valutazione dei rischi permette di determinare quali minacce richiedono azioni immediate e quali possono essere gestite con misure di mitigazione meno urgenti.

[2] https://owasp.org

[3] https://owasp.org/www-project-top-ten

Ad esempio si possono usare le tecniche come l'analisi **SWOT** (Strengths, Weaknesses, Opportunities, Threats) [A.0.34] e le *matrici di rischio* [A.0.39] per valutare e classificare i rischi.

### 1.4.2 Tecniche di software testing

Entrando nel merito del **software testing**, di seguito verranno presentate le tipologie di tecniche di test e verifica più comuni, ognuna delle quali ha un ruolo specifico nel rilevare problemi che potrebbero compromettere il funzionamento del software in ambienti reali.

#### Test Funzionale

Questo tipo di test è utilizzato per garantire che il software esegua le funzioni per le quali è stato progettato. Tale test comporta l'esecuzione del software e la verifica delle sue caratteristiche e funzionalità per assicurarsi che funzioni come previsto [Myers et al., 2011].

#### Test delle prestazioni

Si utilizza per misurare la velocità, la reattività e la stabilità del software attraverso la simulazione di scenari realistici per valutare la gestione del carico da parte e il funzionamento in generale a seconda di diverse condizioni [Myers et al., 2011].

#### Test di sicurezza

Si utilizzano diversi tool per identificare vulnerabilità e debolezze nel software che potrebbero essere sfruttate da attaccanti. Questo tipo di test implica l'uso di varie tecniche per tentare di violare la sicurezza del software, come il *penetration testing* e la scansione delle vulnerabilità [Arbaugh et al., 2000].

#### Verifica formale

Uso di metodi matematici o logici per dimostrare che il software si comporta come previsto e che non contiene difetti o vulnerabilità [Clarke and Wing, 1996].

#### Analisi statica

Analisi del codice sorgente senza una vera e propria esecuzione del software, al fine di identificare potenziali difetti o vulnerabilità [Spinellis, 2006].

#### Analisi dinamica

Analisi del comportamento di un'applicazione software durante la sua esecuzione per identificare difetti o vulnerabilità che potrebbero non essere evidenti nel codice sorgente. Gli strumenti di analisi dinamica possono essere utilizzati anche per rilevare problemi legati a prestazioni, uso della memoria e sicurezza [Dowd et al., 2006]. Ciò è altamente significativo per i componenti frontend [A.0.24], specialmente per quelli esposti su internet, i quali accettando input dagli utenti possono essere veicolo di attacchi.

#### Penetration testing

Simulazione di attacchi informatici contro un sistema, una rete o un'applicazione, per identificare vulnerabilità che potrebbero essere sfruttate dagli attaccanti, quali **Sql injection** [A.0.73], **Cross Site Scripting** [A.0.74] e **buffer overflow** [A.0.70] [Arbaugh et al., 2000].

## 1.5 Analisi statica

L'**analisi statica** è una tecnica di verifica del software che permette di esaminare il codice sorgente senza una vera e propria esecuzione del software.

Essa si basa sulla costruzione di un modello matematico dell'esecuzione del programma al fine di analizzare alcune proprietà del software in modo rapido ed estensivo [Clarke et al., 2004]. Infatti un vantaggio di questa tecnica è proprio la velocità, tuttavia, a causa della mancata esecuzione del programma con dati reali, l'*analisi statica* potrebbe non rivelare problemi specifici legati a certi input oppure a configurazioni a runtime [Chess and West, 2007].

Tra le principali tipologie di *analisi statica* figurano:

### 1.5.1 Esecuzione simbolica

Si eseguono simbolicamente i programmi su input simbolici anziché concreti, permettendo di esplorare molti più percorsi di esecuzione rispetto ai test tradizionali, rilevando bug nascosti che potrebbero non emergere durante una normale esecuzione del programma [Cadar and Sen, 2013].

### 1.5.2 Model checking

Verifica delle proprietà di sicurezza e di correttezza di un modello di programma utilizzando tecniche di esplorazione dello *spazio degli stati* [A.0.66]. È particolarmente utile per sistemi concorrenti [A.0.67], dove i metodi tradizionali di testing potrebbero non coprire tutti gli scenari di esecuzione [Clarke and Schlingloff, 2008].

### 1.5.3 Analisi del flusso di dati

Identifica potenziali problemi come *race conditions* [A.0.72], variabili non inizializzate e accessi a memoria non validi analizzando come i dati si muovono attraverso il software. Questa analisi aiuta a garantire che il flusso di controllo e il flusso di dati nel programma non causino errori di esecuzione [Sadowski et al., 2018].

# Capitolo 2

# Metodologia

A causa di un'insufficiente applicazione delle tradizionali metodologie di analisi di sicurezza nelle *architetture a microservizi* [A.0.1], è sorta l'esigenza di una metodologia analitica che possa mitigare i rischi in contesti di elevata complessità e diversità tecnologica.

La metodologia proposta si prefigge di fornire un quadro sistematico e stratificato per l'analisi di sicurezza, che sia in grado di adattarsi e reagire dinamicamente alle specificità di ciascun componente del sistema, garantendo una continua copertura di sicurezza comprensiva e dettagliata per una generica *architettura a microservizi*.

Questo capitolo è suddiviso in quattro parti.

La prima parte confronta le caratteristiche di sicurezza dei *software monolitici* e dei *microservizi*, sottolineando la necessità di un'analisi dettagliata e specifica per ogni *microservizio* a causa delle differenti tecnologie e interazioni tra i servizi.

La seconda parte espone le limitazioni dei tool di sicurezza esistenti, esplicando le motivazioni che hanno condotto alla sviluppo della metodologia proposta.

La terza parte descrive l'approccio metodologico adottato per l'analisi di sicurezza nelle *architetture a microservizi*.

La quarta parte si concentra sulle principali tipologie di vulnerabilità individuabili nel codice sorgente, nelle *immagini Docker* e nei dispositivi hardware.

In particolare, nella sezione 2.3 sono descritte le fasi della metodologia proposta, partendo dalla definizione dei **requisiti di sicurezza** [A.0.35] e della **gap analysis** [A.0.31], passando poi dalla **threat analysis** [A.0.32] per identificare le potenziali minacce, proseguendo con l'**analisi della sicurezza per strati** per esaminare ogni componente in dettaglio, fino alla generazione di un report di aggregazione che sintetizza i risultati delle valutazioni di sicurezza e concludendo con il monitoraggio continuo per verifiche costanti di sicurezza nel tempo.

## 2.1 Analisi di sicurezza di software monolitici e microservizi

I **software monolitici** sono caratterizzati da un'unica base di codice sorgente e da un'architettura centralizzata, infatti i tool di analisi di sicurezza possono esaminare l'intero sistema come un'unica entità poiché le librerie e i moduli sono utilizzati trasversalmente nell'applicazione e ciò facilita l'identificazione di dipendenze interne e di vulnerabilità trasversali.

Tuttavia, questa architettura può rendere arduo l'aggiornamento e la manutenzione in quanto qualsiasi modifica potrebbe potenzialmente compromettere l'intera applicazione, aumentando il costo di riparazione degli errori a seconda della fase del ciclo di vita dello sviluppo del software.

Contrariamente, i software che adottano un'**architettura a microservizi** offrono una modularità che permette ad ogni componente (o servizio) di operare in modo indipendente, offrendo vantaggi significativi in termini di scalabilità [A.0.14], riuso [A.0.13], resilienza del sistema [A.0.15] e costi di manutenzione [A.0.17].

Tuttavia, questa frammentazione del codice comporta complessità aggiuntive nell'analisi delle vulnerabilità, in quanto ogni *microservizio* potrebbe utilizzare *stack tecnologici* [A.0.22] diversi, avere le proprie dipendenze ed essere soggetto a specifiche configurazioni di sicurezza.

Pertanto le analisi di sicurezza dovrebbero estendersi non solo ai singoli servizi ma anche alle interazioni tra essi. Inoltre, al fine di facilitare l'isolamento e la scalabilità, i *microservizi* sono spesso eseguiti in *container* [A.0.5] e ciò introduce la necessità di gestire la sicurezza anche a livello di *container*, eventuali *orchestratori* [A.0.6], configurazioni di rete e politiche di sicurezza.

## 2.2 Limitazioni dei tool di analisi di sicurezza

I tool di analisi di sicurezza esistenti, pur essendo efficaci nei loro specifici ambiti di applicazione, presentano diversi limiti quando si tratta di operare su un insieme diversificato di componenti tecnologici, perché nonostante siano progettati per uno o più specifici campi di applicazione (ad esempio diversi linguaggi di programmazione, *immagini Docker* [A.0.8], *servizi cloud* [A.0.3]), non è possibile sviluppare infinite integrazioni verso altri tool di sicurezza, al fine di avere un risultato aggregato.

A tal proposito esistono degli **orchestratori di tool di sicurezza**, i quali in base ai tool di sicurezza supportati, permettono la loro esecuzione e conversione dell'output prodotto in un formato omogeneo per l'intero sistema; quest'ultimo

aggregabile con altri output al fine di fornire un quadro complessivo dell'intera infrastruttura da analizzare.

Tra gli orchestratori di tool di sicurezza esistenti più noti figura **DefectDojo**[1], una piattaforma *open-source* di aggregazione dei risultati di sicurezza, che tuttavia è in grado di analizzare solo codice sorgente, scritto in diversi linguaggi, escludendo qualsiasi altro componente non legato al codice sorgente, come immagini di *container*, *servizi cloud* o dispositivi.

Un'alternativa a *DefectDojo* è **Kondukto**[2], una piattaforma commerciale che supporta l'analisi di vari target tra cui codice sorgente in diversi linguaggi, *servizi cloud*, immagini di *container*, infrastruttura di rete e dispositivi mobile. Tuttavia, il costo di questa piattaforma è nell'ordine delle decine di migliaia di dollari all'anno, il quale rappresenta una barriera economica significativa per molte aziende.

## 2.3 Metodologia per l'analisi di sicurezza in architetture a microservizi complesse

Alla luce delle limitazioni evidenziate, è emersa la necessità di sviluppare una nuova metodologia che sia in grado di superare i vincoli dei tool di sicurezza e degli orchestratori esistenti.

La metodologia proposta, si configura come una soluzione di **software assurance continua** progettata per infrastrutture a microservizi e che integra generici tool di sicurezza, eseguibili su generici componenti hardware/software, in un processo unificato e sistematico.

Questa metodologia assicura che la sicurezza sia mantenuta attraverso tutte le fasi del ciclo di vita del software, dalla progettazione iniziale fino al monitoraggio continuo post-rilascio, effettuando una valutazione dei rischi in base alle vulnerabilità individuate.

### 2.3.1 Definizione dei requisiti di sicurezza e gap analysis

La prima fase consiste nella mappatura dei *microservizi* e quindi nell'identificazione di tutti i componenti dell'architettura, dei loro linguaggi di programmazione, delle loro dipendenze nonché degli ambienti di esecuzione (ad esempio i container).

Successivamente, vengono determinati i **requisiti di sicurezza** in base a una combinazione di linee guida normative (ad esempio *GDPR* [A.0.53] e *PCI-DSS*

[1] https://defectdojo.org
[2] https://kondukto.io

[A.0.54]) e best practice del settore (ad esempio *OWASP Top Ten* per le applicazioni web), e specifiche esigenze aziendali dettate dalla consultazione con gli *stakeholder* chiave [A.0.36] (ad esempio i responsabili della sicurezza e team di sviluppo), per comprendere le priorità di sicurezza ed eventuali vincoli implementativi legati all'operatività dei *microservizi*.

Una volta definiti i requisiti di sicurezza, viene effettuata una **gap analysis** [A.0.31] per identificare le discrepanze tra l'attuale stato di sicurezza dell'*architettura a microservizi* e i requisiti desiderati.

### 2.3.2 Threat analysis

In questa fase viene condotta una **threat analysis** [A.0.32] per identificare potenziali minacce (o *threats*) che possono compromettere la sicurezza dei *microservizi* identificati.

Successivamente, vengono illustrati esempi concreti che mostrano come tali minacce possano manifestarsi, collegandoli a specifiche *Common Weakness Enumeration* (CWE) [A.0.43], che forniscono una tassonomia standardizzata delle debolezze che possono essere sfruttate dagli attaccanti.

### 2.3.3 Analisi della sicurezza per strati

In questa fase, per ogni strato tecnologico, si applica una procedura di analisi della sicurezza ottimizzata per quel tipo specifico di tecnologia affinché l'analisi possa catturare una più ampia gamma di vulnerabilità possibile, espresse attraverso metriche quali *CVE* [A.0.44], *CWE*, *severity* [A.0.46] e *score* [A.0.47].

L'approccio per strati garantisce che ogni componente dell'*architettura a microservizi* sia esaminato in dettaglio, tenendo conto delle specifiche tecniche e delle peculiarità di ciascun ambiente.

### 2.3.4 Report di aggregazione

In questa fase viene generato un report nel quale vengono aggregati i risultati di tutte le valutazioni di sicurezza in base alle *CWE* e *threats* individuati nella *threat analysis* e nel quale vengono calcolati diversi livelli di rischio [A.0.37] in base alle *CWE* ed ai *threats*, al fine di presentare lo stato di sicurezza globale.

### 2.3.5 Monitoraggio continuo

Nell'ultima fase vengono impostate tecniche di monitoraggio continuo che effettuano verifiche continue di sicurezza al susseguirsi delle versioni dei *microservizi*.

## 2.4 Principali tipologie di vulnerabilità individuabili nel codice sorgente

In questa sezione vengono esplorate le vulnerabilità più rilevanti che possono essere individuate nel codice sorgente delle applicazioni software. Saranno analizzate diverse categorie di vulnerabilità, tra cui *injection*, *memory corruption*, *remote code execution*, *broken authentication*, *sensitive data exposure*, *broken access control*, *password hardcoded*, *race conditions*, *insufficient input validation*, *elevation of privilege*, *Denial of Service* e *path traversal*.

### 2.4.1 Injection

Le vulnerabilità di tipo **injection** (ad esempio *SQL injection* [A.0.73] e *Cross-Site Scripting* (XSS) [A.0.74]), sono causate dall'incorporazione diretta di input malevoli all'interno delle query oppure all'interno dei comandi eseguiti da un sistema.

Questi attacchi sfruttano la mancanza di un'adeguata sanificazione degli input per eseguire comandi arbitrari sul server, manipolare il database o alterare la presentazione dei contenuti web.

Ad esempio gli attacchi di tipo *SQL injection* possono permettere agli aggressori di bypassare autenticazioni, accedere a dati sensibili, modificare o distruggere dati, mentre gli attacchi *XSS* possono modificare il comportamento delle pagine web per eseguire script malevoli oppure rubare *cookies* e sessioni utente.

### 2.4.2 Memory corruption

Le vulnerabilità di tipo **memory corruption** (ad esempio *buffer overflow* [A.0.70] e *heap overflow* [A.0.71]), si verificano quando c'è un errore nella gestione della memoria durante l'esecuzione di un programma.

In particolare, questi errori permettono agli attaccanti di modificare il normale flusso del programma sovrascrivendo la memoria con dati arbitrari, il che può portare a crash del sistema, *undefined behaviour* [A.0.68] o, nei casi più gravi, all'esecuzione di codice arbitrario sotto il controllo dell'attaccante.

### 2.4.3 Memory leaks

Questa categoria riguarda perdite di memoria non volute a causa della mancata deallocazione di una parte della stessa quando non più necessaria. Ciò può causare crash del sistema oppure degradazioni delle prestazioni, le quali sono sfruttabili per *attacchi DoS* [A.0.76].

Ad esempio, nei dispositivi hardware, i *memory leak* sono spesso associati a driver mal progettati oppure a bug nel *firmware*, che gestiscono in modo inefficace l'allocazione e la deallocazione della memoria.

### 2.4.4 Remote code execution

Le vulnerabilità di tipo **remote code execution** permettono l'esecuzione di codice arbitrario da parte di un attaccante su un sistema remoto tramite l'*injection* di un *payload malevolo* [A.0.65], il quale a causa di una mancata sanificazione dell'input o di un difetto nelle procedure di *deserializzazione* [A.0.97], viene interpretato ed eseguito dal sistema vulnerabile.

L'*injection* può partire, ad esempio, da *web form* che interagiscono con il *backend* [A.0.23] di un'applicazione in maniera non sicura, oppure sfruttando vulnerabilità nei protocolli di rete che non implementano adeguati controlli di autenticazione.

### 2.4.5 Broken authentication

Questo tipo di vulnerabilità sono causate da carenze nei meccanismi di autenticazione e gestione delle sessioni, le quali permettono agli aggressori di impersonare altri utenti o di bypassare il processo di autenticazione. Ciò può essere dovuto ad una mancanza di adeguati controlli di sicurezza oppure ad una mancata invalidazione della sessione dopo il logout.

### 2.4.6 Sensitive data exposure

L'**esposizione di dati sensibili** [A.0.48] avviene quando informazioni riservate quali password o dati personali vengono memorizzati, processati o trasmessi senza adeguate misure di sicurezza (ad esempio la crittografia).

Un esempio è la conservazione di dati sensibili in chiaro e non in un formato criptato nei database e/o nei log, oppure l'uso di protocolli non sicuri per la trasmissione dei dati, e in generale di un'inadeguata protezione dei dati sensibili contro l'accesso da parte di utenti non autorizzati.

### 2.4.7 Broken access control

Vulnerabilità di questo tipo si presentano quando sono presenti dei difetti nella verifica dei permessi di lettura/scrittura degli utenti/ruoli e dei sistemi di restrizione, consentendo ad utenti non autorizzati di accedere a funzionalità oppure a dati che dovrebbero essere, invece, nascosti.

### 2.4.8 Password hardcoded

Con l'espressione **password hardcoded** si intende una pratica di sviluppo che implica l'incorporazione diretta di credenziali all'interno del codice sorgente, esponendo le applicazioni a rischi di sicurezza significativi, perché le credenziali diventano facilmente accessibili agli attaccanti che riescono a visualizzare il codice sorgente in maniera lecita o illecita, oppure tramite tecniche di *reverse engineering* [A.0.57].

### 2.4.9 Race conditions

Le **race conditions** si verificano quando il comportamento di un'applicazione è influenzato dall'ordine non deterministico in cui le operazioni concorrenti vengono eseguite.

Questi difetti possono essere sfruttati dagli attaccanti per condurre attacchi di tipo *time-of-check to time-of-use* in cui l'attaccante interviene tra la verifica e l'uso di una risorsa causando comportamenti imprevisti e potenzialmente dannosi.

### 2.4.10 Insufficient input validation

Questa vulnerabilità si verifica quando un'applicazione non filtra adeguatamente l'input fornito dall'utente, consentendo l'inserimento di *payload malevoli* [A.0.65] per eseguire attacchi di tipo *injection* [2.4.1] o in generale altri attacchi basati sull'inserimento di script o comandi dannosi all'interno di campi di input.

### 2.4.11 Elevation of privilege

L'**elevation of privilege** si verifica quando un attaccante acquisisce privilegi superiori a quelli assegnati al fine di eseguire comandi o accedere a dati che dovrebbero essere al di fuori del suo ambito di autorizzazione.

Ciò può avvenire, ad esempio, attraverso l'uso di diverse tecniche che sfruttano vulnerabilità di mancata validazione dell'input [2.4.10].

### 2.4.12 Denial of Service

Gli attacchi di tipo **Denial of Service** (DoS) mirano a rendere una risorsa di sistema non disponibile attraverso un sovraccaricamento del sistema con una grande mole di richieste.

Le vulnerabilità che permettono tali attacchi possono essere causate, ad esempio, da mancate verifiche sui permessi/privilegi prima dell'esecuzione di task computazionalmente costosi, da mancate impostazioni di tecniche di *throttling* per limitare

il numero di richieste effettuabili in un determinato periodo di tempo oppure da un mancato *load balacing* per distribuire il carico di lavoro su più server.

### 2.4.13 Path traversal

Le vulnerabilità di tipo **path traversal** si verificano quando un'applicazione non valida correttamente gli input relativi ai percorsi dei file o delle directory, consentendo agli attaccanti di accedere a file o directory fuori dalla radice prevista del web server.

## 2.5 Principali tipologie di vulnerabilità individuabili in immagini Docker

In questa sezione vengono descritte le vulnerabilità più significative che possono essere rilevate all'interno delle *immagini Docker*, come *vulnerabilità di sistema operativo e dipendenze*, *configurazioni non sicure*, *esposizione di dati sensibili* e rischi introdotti durante il processo di costruzione delle immagini stesse.

### 2.5.1 Vulnerabilità di sistema operativo e dipendenze

Le *immagini Docker* possono includere vulnerabilità ereditate dal sistema operativo di base e dalle dipendenze delle applicazioni.

Ad esempio le immagini di sistema operativo come *Debian*[3], *Ubuntu*[4] o *CentOS*[5], di versioni obsolete, possono contenere vulnerabilità note che possono essere sfruttate da aggressori per ottenere accesso non autorizzato o eseguire codice malevolo all'interno dei *container*.

### 2.5.2 Configurazioni non sicure

Configurazioni di default non sicure nelle *immagini Docker*, come l'esposizione di porte non necessarie, l'uso di credenziali predefinite, e configurazioni inadeguate delle reti interne, possono esporre le applicazioni ad attacchi di diverso tipo.

[3]https://debian.org
[4]https://ubuntu.com
[5]https://centos.org

### 2.5.3 Esposizione di dati sensibili

*Immagini Docker* mal configurate possono involontariamente includere dati sensibili nei *layer* dell'immagine stessa, come *chiavi API*, password o token di accesso e ciò può portare a gravi violazioni se gli attaccanti riescono ad accedere a tali informazioni.

### 2.5.4 Inserimento di vulnerabilità durante la costruzione

Durante il processo di costruzione delle *immagini Docker* possono essere introdotte vulnerabilità a causa di script di costruzione non sicuri oppure a causa dell'uso di *Dockerfile* [A.0.9] non revisionati.

## 2.6 Principali tipologie di vulnerabilità individuabili in dispositivi hardware

In questa sezione, vengono esaminate le vulnerabilità critiche che possono essere individuate nei dispositivi hardware, mettendo in evidenza i rischi per la sicurezza che queste vulnerabilità comportano. Saranno esplorate categorie come *vulnerabilità nel boot loader*, *configurazioni non sicure del kernel*, *processi con privilegi eccessivi*, *attacchi Man-in-the-Middle* (MITM), *exploitation di bug nel firmware*, *configurazioni predefinite insicure* ed *errori nei meccanismi di protezione*.

### 2.6.1 Boot loader insicuro

Le **vulnerabilità nel boot loader** [A.0.87] permettono agli attaccanti di bypassare i meccanismi di sicurezza all'avvio con attacchi di tipo *rootkit* o *bootkit* che modificano il processo di avvio del sistema operativo per eseguire codice malevolo prima che il sistema operativo stesso abbia la possibilità di attivare le proprie misure di sicurezza [Kleissner, 2009].

Questi attacchi sfruttano spesso vulnerabilità note nell'*Unified Extensible Firmware Interface* (UEFI[6]), l'interfaccia firmware standardizzata che ha sostituito il vecchio *Basic Input/Output System* (BIOS) nei computer moderni [Sabt et al., 2015].

### 2.6.2 Software non utilizzati

La presenza di pacchetti software non essenziali alle operazioni ordinarie, magari abilitati all'avvio, aumentano la superficie di attacco del dispositivo perché possono

[6]https://uefi.org

introdurre potenziali vulnerabilità oppure interferire con altri programmi, causando conflitti o rallentamenti.

### 2.6.3 Configurazioni non sicure del kernel

Il *kernel* è il nucleo centrale del sistema operativo ed ha il controllo completo delle risorse del sistema, quindi, qualsiasi vulnerabilità a questo livello può portare a conseguenze devastanti.

In particolare, alcune configurazioni non sicure possono esporre il sistema a *exploit* che consentono la *privilege escalation* [A.0.75].

Un esempio di configurazione non sicura è l'abilitazione delle funzionalità di *debugging nel kernel*, le quali dovrebbero essere disattivate in un ambiente di produzione dato che possono essere sfruttate per eseguire codice arbitrario con *privilegi di root* [Hallinan, 2006].

### 2.6.4 Processi con privilegi eccessivi

L'esecuzione di un software, contenente vulnerabilità, con privilegi superiori a quelli necessari, aumenta il rischio di attacchi di tipo *privilege escalation*, permettendo ad un attaccante di eseguire azioni malevoli a livello di sistema.

### 2.6.5 Man-in-the-Middle

Gli attacchi di tipo **Man-in-the-Middle** (MITM) permettono ad un aggressore di intercettare e manipolare le comunicazioni tra due dispositivi senza che le vittime ne siano consapevoli, compromettendo la riservatezza e l'integrità dei dati.

Un esempio di attacco *MITM* nei dispositivi hardware *IoT* riguarda l'intercettazione di comunicazioni non cifrate di protocolli wireless, ad esempio quelle *Bluetooth*.

### 2.6.6 Exploitation di bug nel firmware

Il *firmware* è il software integrato nei dispositivi per gestire le operazioni a basso livello e le vulnerabilità nel *firmware* possono essere sfruttate per ottenere accesso non autorizzato, eseguire codice arbitrario o compromettere l'integrità del dispositivo.

Tali attacchi sono particolarmente pericolosi perché il *firmware* opera a un livello di privilegio elevato e spesso al di fuori del controllo dell'utente finale, inoltre le tecniche di persistenza del *firmware* permettono agli aggressori di mantenere il controllo del dispositivo anche dopo un riavvio o una reinstallazione del sistema operativo.

### 2.6.7 Configurazioni predefinite insicure

Le **configurazioni predefinite insicure** riguardano le impostazioni di default che non sono ottimizzate per la sicurezza, come credenziali di default (reperibili anche su internet), porte di rete aperte non necessarie (rilevabili con strumenti di *port scanning*) e servizi non essenziali attivi.

Tali impostazioni possono lasciare i dispositivi esposti a una serie di attacchi, compromettendo la sicurezza dell'intero sistema.

### 2.6.8 Errori nei meccanismi di protezione

Errori nella configurazione di funzionalità di sicurezza, come configurazioni errate di **AppArmor**[7] o **SELinux**[8], possono compromettere i controlli di accesso obbligatori per il kernel Linux.

[7] https://apparmor.net
[8] http://selinuxproject.org

# Capitolo 3

# Tecnologie utilizzate

Questo capitolo fornisce una panoramica delle principali tecnologie impiegate nel progetto, descrivendo il ruolo e le funzionalità di ciascuna nel contesto dell'analisi di sicurezza delle *architetture a microservizi*.

La sezione 3.1 introduce **MoonCloud**, una piattaforma per la valutazione continua di conformità e di *assurance* [1.4] per applicazioni e infrastrutture ICT, descrivendo la metodologia adottata dalla piattaforma, le sue feature principali e il funzionamento generale.

Infine, la sezione 3.2 esamina i tool di *analisi statica* [1.4.2] utilizzati, quali **Gosec** [3.2.1], **Bandit** [3.2.1], **Trivy** [3.2.2] e **Lynis** [3.2.3], descrivendone il funzionamento e l'applicazione nel contesto del progetto.

## 3.1 MoonCloud

**MoonCloud**[1] è una piattaforma per la valutazione continua di conformità e di *assurance* [1.4] per applicazioni e infrastrutture ICT, nata per rispondere al crescente bisogno di analizzare in maniera trasparente gli strati sempre più nascosti delle infrastrutture ICT moderne [MoonCloud, 2024].

### 3.1.1 Metodologia

L'implementazione della piattaforma si basa sull'attività di ricerca "*A continuous certification methodology for devops*" [Anisetti et al., 2019].

In particolare, vista la bassa trasparenza e incertezza degli ambienti *cloud* è stata proposta una metodologia di certificazione continua che permette di valutare

[1] https://moon-cloud.eu

le proprietà non funzionali come la sicurezza, la privacy e l'affidabilità in maniera continua, cioè in ogni fase del ciclo di vita del software [Anisetti et al., 2019].

Il suo funzionamento consiste nell'integrazione di un processo di verifica continua nel processo di sviluppo *DevOps*, sincronizzando la verifica del software con la *Continuous Integration* (CI) e il *Continuous Deployment* (CD) assicurando, quindi, una valutazione costante e aggiornata del sistema software [Anisetti et al., 2023].

### 3.1.2 Feature

*MoonCloud* offre una valutazione continua della conformità e dell'*assurance* in diversi ambiti. Ad esempio per le infrastutture ICT moderne che fanno un uso massiccio di microservizi, fornisce controlli specifici per cloud pubblici come *AWS*[2] e *Azure*[3], controlli ad-hoc per *infrastrutture on-premises* [A.0.4], regole di conformità per standard rilevanti (ad esempio *Agid*[4] e *GDPR* [A.0.53]) e monitoraggio delle minacce basato su controlli di *vulnerability assessment* e *penetration testing* [1.4.2].

Per le applicazioni basate su *artificial intelligence* e *machine learning*, *MoonCloud* offre controlli specifici che monitorano i modelli in tempo reale garantiscono la conformità dei processi secondo standard come *CapAI* [A.0.55] e *ALTAI* [A.0.56].

Infine, nell'ambito dell'*Edge Cloud Continuum* [A.0.81], la piattaforma permette la valutazione su larga scala di infrastrutture composte da dispositivi e domini eterogenei e controlli di sicurezza per dispositivi *IoT*, *reti 5G* e *nodi edge*.

### 3.1.3 Funzionamento

La piattaforma *MoonCloud* è composta da un cluster *Kubernetes* [A.0.7] che gestisce diverse **probe**, cioè controlli di sicurezza che vengono lanciati contro un target.

In particolare, una generica probe è una classe *Python*[5] che eredita da una classe base, chiamata *MoonCloud driver*, metodi per costruire i controlli di sicurezza secondo una precisa struttura, definita appunto dal driver, e che produce tre risultati:

- **Integer result**: un numero intero per indicare se la probe ha riscontrato o meno discordanze con le proprietà che deve asserire oppure per indicare che si è verificato un errore specifico;
- **Pretty result**: una breve stringa che descrive dettagliatamente il motivo per cui la probe ha restituito tale numero intero;

[2]https://aws.amazon.com
[3]https://azure.microsoft.com
[4]https://agid.gov.it
[5]https://python.org

- **Extradata**: un insieme arbitrario di valori contenenti le evidenze raccolte dalla probe.

Attraverso una dashboard dedicata è possibile, definire uno o più target ai quali associare una o più *probe* sviluppate e visualizzare i risultati al termine delle valutazioni.

Fisicamente la probe è un container *Docker* che viene eseguito a seguito del suo inserimento in una catena di *microservizi*.

In particolare, l'input viene iniettato dal *Kubernetes manager* [A.0.7] e l'output viene passato all'*evidence writer*, un componente che si occupa di scrivere i risultati della probe sul database in ordine temporale.

A livello implementativo la probe è una macchina a stati finiti con due catene definite come segue:

- La **main chain** (o **forward chain**) è quella che viene eseguita normalmente se non si verificano errori;
- La **backward chain** (o **rollback chain**) è quella che viene eseguita se si verifica un errore e che si occupa di compiere *azioni di rollback* rispetto alle azioni di scrittura/modifica compiute sul target.

  Da notare che se questa catena è definita deve esistere una corrispondenza tra uno stato nella catena principale e uno nella catena di *rollback*.

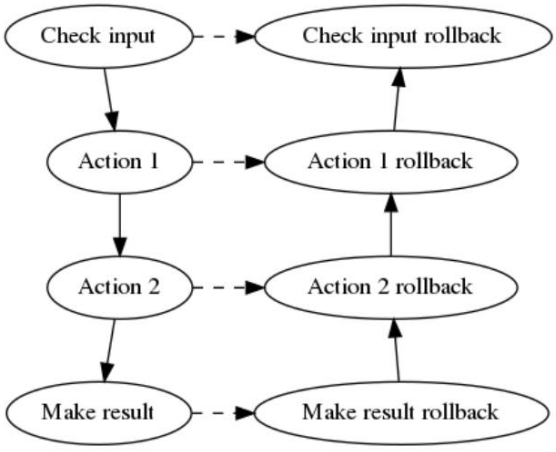


Figura 1: Architettura *probe MoonCloud*

In particolare, ogni catena è composta da più stati (metodi *Python*) che rappresentano i passi del controllo di sicurezza. Tipicamente, ci sono almeno i seguenti tre stati:

1. Fase iniziale che riceve l'input, ne verifica la correttezza ed eventualmente compie alcune azioni per l'inizializzazione della probe (ad esempio settaggio di alcuni attributi per i passaggi successivi);

2. Fase che esegue l'azione effettiva per la quale la probe è stata progettata, ad esempio l'esecuzione di un tool esterno;

3. Fase che valuta i risultati ottenuti e restituisce l'output secondo il formato già descritto [3.1.3].

## 3.2 Tool di analisi statica utilizzati

Questa sezione esamina diversi strumenti di analisi statica utilizzati per identificare e mitigare vulnerabilità di sicurezza nel software e nei sistemi, sottolineando le metodologie e le capacità specifiche di ciascuno strumento.

### 3.2.1 Gosec e Bandit

**Gosec**[6] e **Bandit**[7] sono due strumenti di analisi statica *open source* [A.0.19] progettati per identificare potenziali vulnerabilità di sicurezza, rispettivamente, nel codice *Go* e nel codice *Python*.

Entrambi operano allo stesso modo, cioè analizzano il codice sorgente utilizzando l'*Abstract Syntax Tree* (AST) del linguaggio, la struttura sintattica del codice in una forma ad albero, dove ogni nodo corrisponde a una costruzione sintattica del linguaggio, come espressioni, dichiarazioni e blocchi di codice.

In particolare, durante il processo di analisi, vengono eseguite le seguenti operazioni:

1. **Parsing del codice sorgente:** lettura del codice sorgente e costruzione dell'*AST* corrispondente;

2. **Applicazione delle regole di sicurezza:** lo strumento applica un insieme predefinito di regole di sicurezza all'*AST*. Queste regole sono progettate per identificare costrutti potenzialmente pericolosi o vulnerabili nel codice;

[6] https://github.com/securego/gosec
[7] https://pypi.org/project/bandit

3. **Segnalazione delle vulnerabilità:** se viene rilevata una vulnerabilità, viene generato un report dettagliato che descrive il problema, posizione nel codice sorgente, *Common Vulnerabilities and Exposures* (CVE) [A.0.44] e *score*.

### 3.2.2 Trivy

**Trivy**[8] è uno strumento per l'analisi di sicurezza progettato per identificare vulnerabilità note, basandosi sul database di vulnerabilità *CVE*, in diverse tipologie di target, tra cui i pacchetti software e loro dipendenze, immagini Docker e *file system*.

#### Immagini Docker

Per l'analisi dei *layer delle immagini Docker*, *Trivy* prende in considerazione i file e i metadati delle stesse al fine di individuare:

- **Vulnerabilità** nei pacchetti software installati e nelle dipendenze;
- **Configurazioni errate**, convertendo l'immagine in un *Dockerfile* [A.0.9] per individuare impostazioni di configurazione che potrebbero rappresentare un rischio per la sicurezza come permessi di file non sicuri e impostazioni di rete inadeguate;
- **Secrets**, convertendo la configurazione dell'immagine in un file *JSON* [A.0.26] per individuare, ad esempio, variabili d'ambiente che potrebbero contenere credenziali, *chiavi API* o altre informazioni sensibili.

#### File system

*Trivy* esegue scansioni su generici *file system* a partire da una directory specifica oppure dalla loro radice, al fine di analizzare interi sistemi operativi, individuando:

- **Vulnerabilità** in pacchetti software basandosi su file che descrivono dipendenze, ad esempio, *go.mod*, *package-lock.json* e *Gemfile.lock*;
- **Configurazioni errate** [3.2.2];
- **Secrets** [3.2.2].

[8] https://trivy.dev

### 3.2.3 Lynis

**Lynis**[9] è uno strumento *open source* [A.0.19] di *audit* e *hardening*, progettato per sistemi *Unix-like*, il quale consente di identificare vulnerabilità e fornire raccomandazioni per risolvere le problematiche di sicurezza.

Le scansioni di *Lynis* sono modulari, cioè vengono testati solo i componenti che vengono individuati nel sistema, eseguendo scansioni su misura a seconda del sistema specifico.

In particolare, *Lynis* effettua una serie di test di sicurezza che coprono varie aree, ad esempio i servizi di autenticazione, firewall, servizi di rete e controlli di sistema.

Inoltre analizza configurazioni di sistema e software per identificare potenziali configurazioni errate o vulnerabili, ad esempio impostazioni del kernel, permessi dei file, patch di sicurezza mancanti, configurazioni di sicurezza di software come *SSH* [A.0.91] e altri software.

Al termine della scansione, *Lynis* produce un report dettagliato che elenca gli eventuali punti deboli trovati e nel quale compare una metrica denominata **hardening index score**.

Questa metrica consente di valutare il livello di *hardening* di un sistema *Unix-like* che può variare tra 0 e 100, dove un punteggio più alto indica un livello di sicurezza migliore.

[9] https://cisofy.com/lynis

# Capitolo 4

# Implementazione

Questo capitolo descrive l'applicazione pratica della metodologia di analisi di sicurezza precedentemente delineata, presentando i risultati ottenuti e le soluzioni implementate per garantire la sicurezza delle *architetture a microservizi*.

La sezione 4.1 presenta lo scenario di applicazione, descrivendo **UrbanIoT** [4.1.2], un software di telegestione di impianti di illuminazione pubblica e smart city basato su **Mainflux** [4.1.1], un *framework open source* per lo sviluppo di soluzioni *IoT*. Lo scenario di applicazione è stato oggetto della sperimentazione dell'implementazione della metodologia, tuttavia per quanto riguarda il software *UrbanIoT*, si precisa che ne verrà presentata una versione semplificata per questioni di riservatezza.

La sezione 4.2 fornisce un confronto e le motivazioni dietro la scelta dei tool di analisi di sicurezza adottati, evidenziando i criteri di selezione e le ragioni per cui ciascuno strumento è stato preferito rispetto ad altri strumenti alternativi.

La sezione 4.3 è dedicata alla definizione dei *requisiti di sicurezza* e alla *gap analysis* per i componenti *Mainflux* e *UrbanIoT*, analizzando le caratteristiche di sicurezza dichiarate e identificando eventuali lacune da colmare per raggiungere un livello di sicurezza ottimale.

Successivamente, nella sezione 4.4, relativa alla *threat analysis*, vengono identificate e valutate le potenziali minacce che possono compromettere la sicurezza dei sistemi, utilizzando la tassonomia ufficiale di **ENISA** [A.0.51] e del progetto **H2020-CONCORDIA** [A.0.50] per classificare i vari tipi di minacce e debolezze.

La sezione 4.5, dedicata all'analisi della sicurezza per strati, descrive l'applicazione di procedure di analisi specifiche per ciascun livello tecnologico dell'*architettura a microservizi*, utilizzando tool di *analisi statica* come *Gosec*, *Trivy*, *Bandit* e *Lynis* per valutare la sicurezza del codice sorgente, delle *immagini Docker* e del *dispositivo edge*.

La sezione 4.6 presenta un'aggregazione e un'analisi complessiva dei risultati delle varie valutazioni di sicurezza, calcolando livelli di rischio complessivi e specifici per ogni *CWE* e *threat* identificato e fornendo una visione d'insieme della sicurezza del sistema.

Infine, la sezione 4.7, relativa al monitoraggio continuo, illustra come le probe sviluppate possano essere integrate nelle *pipeline DevOps* [1.1] attraverso le *API* di *MoonCloud*, al fine di avere una valutazione costante e aggiornata della sicurezza dei *microservizi* durante l'intero ciclo di vita del software.

## 4.1 Scenario

### 4.1.1 Mainflux

**Mainflux**[1] è un *framework open source* scritto in *Go*[2], estendibile e integrabile con applicazioni di terze parti, che fornisce una serie di funzionalità fondamentali per lo sviluppo di soluzioni *IoT* attraverso un insieme di *microservizi containerizzati con Docker*, al fine di garantire alte prestazioni, scalabilità e *fault tolerance* [A.0.16].

La piattaforma *Mainflux* funge, quindi, da infrastruttura software e *middleware* con le seguenti feature principali:

- **Gestione dei dispositivi**: registrazione, autenticazione e monitoraggio, attraverso la gestione remota dei *dispositivi IoT*;
- **Raccolta dei dati**: raccolta e archiviazione dei dati strutturati e non strutturati provenienti dai *dispositivi IoT*, in maniera scalabile e con il supporto di meccanismi di prioritizzazione dei dati per flussi improvvisi di grandi quantità di dati;
- **Analisi dei dati**: analisi dei dati in tempo reale e visualizzazione in una dashboard dedicata di *insight* significativi come report, grafici, mappe con la posizione dei dispositivi. Supporto di *algoritmi di machine learning* per l'estrazione automatica di informazioni rilevanti;
- **Sicurezza dei dati**: integrità dei dati sensibili garantita attraverso un insieme di misure di sicurezza avanzate quali, autenticazione basata su ruoli con chiavi *API* [A.0.28] e *token JWT* [A.0.27] con *scope* di accesso personalizzabili, crittografia dei dati in transito attraverso mutua autenticazione *TLS* [A.0.88], per dispositivi e servizi, tramite *certificati X.509* [A.0.89], e un *reverse proxy Nginx* [4.1.1] per garantire la sicurezza e il bilanciamento del carico.

[1] https://mainflux.com
[2] https://go.dev

Nella figura 2 è rappresentata l'architettura e i componenti principali del framework, i quali saranno successivamente descritti:

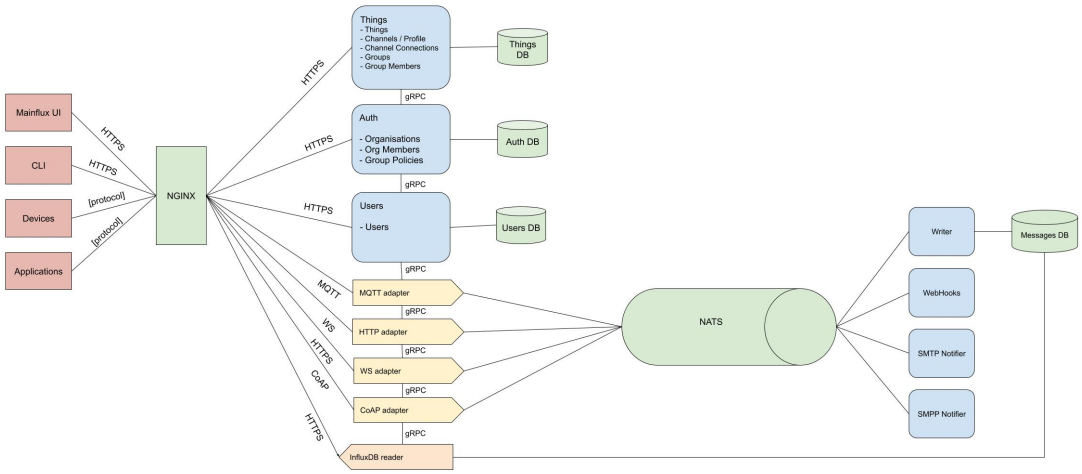


Figura 2: Architettura e componenti principali del framework *Mainflux*

### Users

Gestisce gli utenti e l'autenticazione sulla piattaforma.

### Auth

Funzionalità di autenticazione tramite *API* per gestire le chiavi di autenticazione e di amministrare *things* e *users*.

### Things

Gestisce le *things*, i *channels* e le politiche di accesso.

- Una *thing* rappresenta un dispositivo connesso a *Mainflux* e che usa la piattaforma per scambiare messaggi con altre *things*.
- Un *channel* rappresenta un canale di comunicazione, raggruppa i messaggi che possono essere consumati da tutte le *things* collegate al canale.

### Protocol adapters

Gestisce le *things*, i *channels* e le politiche di accesso della piattaforma con diverse interfacce a seconda dei protocolli di comunicazione implementati.

**NATS**[3]

Sistema di messaggistica scalabile *open source* per lo scambio di messaggi all'interno dei *Mainflux*.

**Jaeger**[4]

Sistema *open source* per il monitoraggio delle transazioni distribuite in *architetture a microservizi* per tracciare e analizzare il flusso delle richieste attraverso i componenti di un'applicazione.

**Nginx**[5]

Proxy che funge da interfaccia di ingresso ed uscita. Riceve le richieste da parte dei client esponendo delle *API* ed inoltra le richieste ai singoli componenti.

**PostgreSQL**[6]

Database relazionale per archiviare metadati (*users*, *things*, *channels* e rispettivi token di autenticazione).

**Redis**[7]

*Data store* in memoria utilizzato come database e cache.

**InfluxDB**[8]

Database *open source* progettato per gestire in maniera ottimizzata dati come serie temporali, metriche e dati di eventi, tipici dei *sistemi IoT*.

**Grafana**[9]

Piattaforma *open source* per il monitoraggio e l'osservazione dei dati al fine di visualizzare e analizzare metriche tramite dashboard grafiche interattive che si integrano con varie fonti di dati come *InfluxDB*.

[3]https://nats.io
[4]https://jaegertracing.io
[5]https://nginx.org
[6]https://postgresql.org
[7]https://redis.io
[8]https://influxdata.com
[9]https://grafana.com

### 4.1.2 UrbanIoT

**UrbanIoT** è il nome utilizzato in questa tesi per riferirsi ad un reale software di telegestione degli impianti di illuminazione pubblica e delle smart city, che tuttavia per ragioni di riservatezza industriale, non è possibile descrivere con ulteriori dettagli.

Questo software è scritto in *Go* ed è basato sul framework *Mainflux*, infatti come descritto in figura 3, *UrbanIoT* è costituito da 4 strati progettuali:

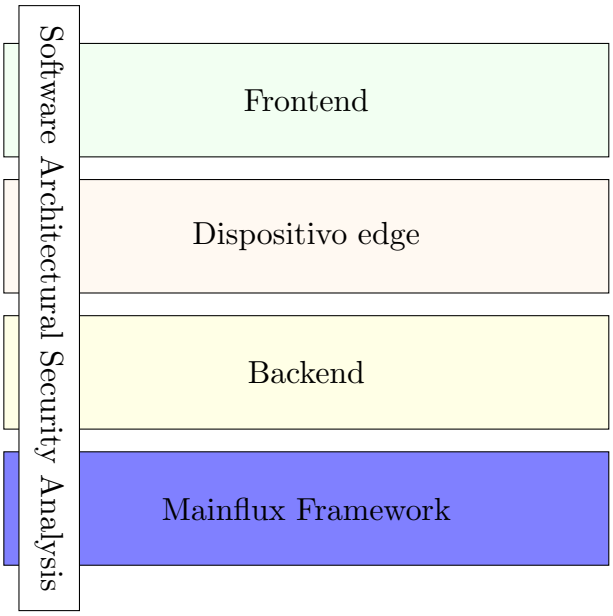


Figura 3: Livelli dell'architettura di UrbanIoT

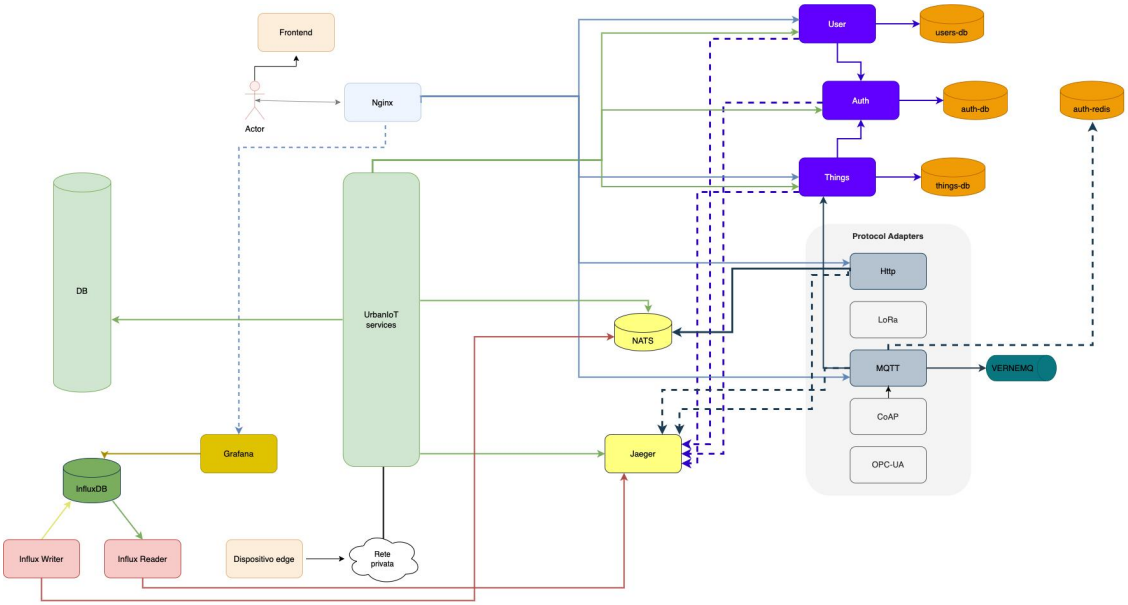


Figura 4: Architettura e componenti principali di *UrbanIoT*

### Backend

Nell'architettura di *UrbanIoT* mostrata nella figura 4 sono presenti, oltre ai componenti proprietari anche i componenti chiave di *Mainflux*, i quali sono componenti generici molto usati in tantissimi contesti ed hanno una ottima solidità e team di sviluppo consolidati.

Per tali motivi sono anche stati individuati come servizi su cui basare anche lo sviluppo dei componenti proprietari di *UrbanIoT* in completa sinergia quindi con il framework.

Tale scelta ha ottimi impatti sul design della soluzione nella sua interezza e fornisce una minore esposizione a problematiche di sicurezza data la miglior possibilità di controllare la superficie di attacco.

Da notare che l'architettura di *UrbanIoT* è un'architettura complessa, dato che è composta da diversi microservizi per il backend, diverse tipologie di database, un microservizio per il frontend e un *dispositivo edge*.

A tal proposito risulta altrettanto complesso effettuare analisi di sicurezza con i tool esistenti, per i motivi già descritti nella sezione 2.2, pertanto questa architettura è stata oggetto di sperimentazione per la metodologia proposta [2.3].

#### Frontend

Il frontend di *UrbanIoT* è costituito da una singola applicazione, sviluppata utilizzando il framework *AngularJS* [A.0.25], la quale opera come un *microservizio* indipendente che non è direttamente integrato nel progetto *UrbanIoT*, bensì ci si interfaccia attraverso le *API* disponibili per ottenere dati in lettura e inviare comandi al sistema per eseguire varie operazioni.

Da notare che, a differenza dei componenti precedentemente descritti, la dashboard di *UrbanIoT*, essendo basata su *AngularJS*, presenta un numero significativamente maggiore di dipendenze esterne e ciò introduce maggiori potenziali rischi di sicurezza, poiché ogni dipendenza esterna può rappresentare una possibile fonte di vulnerabilità gravi.

#### Dispositivo edge

Il **dispositivo edge** di *UrbanIoT* è un dispositivo *Raspberry Pi* [A.0.86] basato su *Linux Raspbian* con *architettura ARM* [A.0.85].

La funzione del *dispositivo edge* è quella di utilizzare un collegamento dati *IP* verso il proprio server di controllo centrale per ricevere i parametri di funzionamento con i quali comandare il driver dell'apparecchio di illuminazione e fornire i dati delle misure effettuate in campo.

## 4.2 Confronto e motivazioni sulle scelte dei tool

Questa sezione confronta i diversi strumenti di analisi di sicurezza e fornisce le motivazioni dietro le scelte effettuate per ciascun contesto di analisi di sicurezza.

### 4.2.1 Frontend - AngularJS

Durante la scelta dei tool, è stata riscontrata una mancanza di strumenti dedicati esclusivamente all'analisi delle vulnerabilità per progetti **AngularJS**, a differenza di *Go* per il quale esiste uno strumento specifico come *Gosec*.

A tal proposito, è stato utilizzato *Trivy* nella modalità *file system* per eseguire la scansione della cartella del progetto *AngularJS*, analizzando il file *package.json* e le relative dipendenze.

### 4.2.2 Backend - Go

Per l'*analisi statica* del codice *Go* sono stati confrontati i seguenti tool: *golint*[10], *govet*[11], *errcheck*[12] e *Gosec*[13].

E' stato scelto di implementare nei controlli di sicurezza **Gosec** perché per quanto riguarda i primi tre è stato riscontrato che sono strumenti adatti per la rilevazione di errori di sintassi, problemi di stile e bug nel codice e non offrono la stessa profondità nell'individuazione di vulnerabilità di sicurezza di *Gosec*, dato che nonostante siano utili per migliorare la qualità del codice, mancano delle capacità di associare i problemi rilevati con specifiche vulnerabilità di sicurezza e *CVE*.

### 4.2.3 Script Python

Per l'*analisi statica* del codice *Python* sono stati presi in considerazione i seguenti tool: *Pylint*[14], *MyPy*[15], *Bandit*[16], *Safety*[17], *Snyk*[18] e *SonarQube*[19].

E' stato scelto di implementare nei controlli di sicurezza **Bandit** perché per quanto riguarda i primi due è stato riscontrato lo stesso problema visto per *Gosec* [4.2.2].

Inoltre, sebbene *Safety*, *Snyk* e *SonarQube* analizzino i progetti e le loro dipendenze per trovare vulnerabilità di sicurezza note, parte delle funzionalità sono a pagamento, essendo piattaforme *SaaS* [A.0.20] e solo in parte *open source*.

### 4.2.4 Immagini Docker

Per le valutazioni di sicurezza delle *immagini Docker*, sono stati confrontati *Grype*[20] e *Trivy* e dopo un'analisi approfondita, la scelta è ricaduta su *Trivy* per diverse ragioni chiave che lo rendono preferibile rispetto a *Grype*:

- **Versatilità**: sebbene entrambi gli strumenti siano in grado di individuare un numero comparabile di *CVE*, *Trivy* offre una maggiore versatilità perché, come già detto, non si limita a scansionare solo le *immagini Docker*, ma può

[10] https://golangci-lint.run
[11] https://pkg.go.dev/cmd/vet
[12] https://github.com/kisielk/errcheck
[13] https://golangci-lint.run
[14] https://pypi.org/project/pylint
[15] https://mypy-lang.org
[16] https://github.com/PyCQA/bandit
[17] https://safetycli.com
[18] https://snyk.io
[19] https://sonarsource.com
[20] https://github.com/anchore/grype

essere utilizzato anche per analizzare i *file system* locali, infatti è stato anche per altri target;

- **Community**: *Trivy* beneficia di una *community* più grande e attiva, il che significa un accesso più rapido a risorse, tutorial e risoluzioni di problemi;
- **Sistemi operativi supportati**: *Trivy* supporta un numero di sistemi operativi, usati come *immagini Docker* di base, maggiore e ciò aumenta il grado di compatibilità con *immagini Docker* eterogenee;
- **Maggiori sorgenti di vulnerabilità**: *Trivy* si appoggia ad un numero maggiore di database di vulnerabilità, permettendo una copertura delle vulnerabilità più completa e aggiornata.

### 4.2.5 Dispositivo edge - Lynis

Per l'*hardening del dispositivo edge* sono stati presi in considerazione i seguenti tool: *Lynis*, *OpenVAS*[21] e *Nessus*[22].

*Nessus* è un software proprietario disponibile solo come parte di un'offerta commerciale, quindi è stato scartato a priori.

E' stato scelto **Lynis** perché rispetto ad *OpenVAS*, offre una maggiore copertura degli sistemi operativi supportati e una profondità di audit superiore, inoltre a differenza di *OpenVAS*, che scansiona il target tramite connessione di rete, *Lynis* opera direttamente sull'host esclusivamente in modalità lettura e ciò riduce il rischio di compromissione dei processi aziendali in produzione.

Infine, operando direttamente sull'host, i log di connessione sono privi di tentativi di connessione e richieste errate che potrebbero allertare sistemi di difesa quali *IPS* [A.0.94] e *IDS* [A.0.93].

## 4.3 Definizione dei requisiti di sicurezza e gap analysis

Questa sezione esamina i requisiti di sicurezza definiti per *Mainflux*, analizzando le caratteristiche dichiarate e confrontandole con le pratiche effettivamente implementate. Successivamente, viene presentata una una *gap analysis* per identificare eventuali lacune di sicurezza e potenziali rischi associati ai requisiti dichiarati.

[21] https://openvas.org
[22] https://tenable.com/products/nessus

Da notare che anche per *UrbanIoT* si è proceduto ad identificare i requisiti di sicurezza e condurre la *gap analysis*, tuttavia per i motivi già esplicati [4] non è possibile includere i risultati in questa tesi.

### 4.3.1 Requisiti di sicurezza

Di seguito vengono analizzate le caratteristiche di sicurezza dichiarate da *Mainflux* ed il loro rapporto con i componenti, inclusi i componenti esterni infrastrutturali utilizzati, definiti nella figura 2.

In particolare *Mainflux* dichiara le seguenti caratteristiche:

**Rx1** **Autenticazione e autorizzazione attraverso il controllo degli accessi sulle *API***: garantite dai componenti *auth* e *users* con autenticazione attraverso *JWT token* [A.0.27];

**Rx2** **Confidenzialità del canale**: garantita con *certificati X.509* [A.0.89]. Inoltre tutti i componenti consentono l'abilitazione di *mutual TLS authentication* [A.0.88] per crittografare la comunicazione fra i singoli componenti e con l'esterno;

**Rx3** **Load balancing [A.0.11]**: *Nginx reverse proxy* [4.1.1] per il load-balancing posizionato come gateway davanti a tutti i componenti esposti con il client;

**Rx4** **Unico accesso al perimetro di sicurezza**: *Nginx reverse proxy* per la sicurezza e terminazione delle connessioni TLS e DTLS.

### 4.3.2 Gap analysis

In questa sezione si identificano eventuali *gap di sicurezza* di *Mainflux*, distinguendo tra *warning tecnici* sui requisiti e *gap teorici*.

Il colore indica la severità del gap [bassa, media, alta].

**Rx1** La libreria utilizzata per gestire i *token JWT* è *dgrijalva/jwt-go* [23] e il suo utilizzo deve essere monitorato, in quanto il *repository* che ospita il codice della versione utilizzata da *Mainflux* non è attualmente mantenuto. La libreria di riferimento dovrebbe essere: *golang-jwt/jwt* [24].

**Rx2** La versione del protocollo crittografico utilizzata è l'ultima disponibile ovvero la 1.3 però è da controllare nel tempo.

[23] https://github.com/dgrijalva/jwt-go
[24] https://github.com/golang-jwt/jwt

**Rx3, Rx4** Sebbene *Nginx* non sia dell'ultima versione disponibile al momento della scrittura (1.25.5), la versione utilizzata di default da Mainflux (1.23.3) è conforme alle richieste dei requisito.

**Gm1 Confidentiality** a livello di log.

Nella fase di registrazione, i log non intaccano la privacy dell'utente sebbene per motivi tecnici, lo username sia registrato insieme al messaggio che ne attesta l'avvenuta registrazione.

**Gm2 Confidentiality** a livello di dato salvato.

*Mainflux* non esprime alcun requisito in merito, demandando all'applicazione specifica. Ad ogni modo si segnala come *Mainflux* includa servizi per il salvataggio dei dati (*PostgreSQL*, *InfluxDB*, *Redis*) e che questi non siano configurati di default per garantire la confidenzialità.

**Gm3 Auditability** attraverso l'uso dei log per audit di sicurezza.

Al momento sembra non siano espressivi a sufficienza pertanto si rende necessario un'analisi protratta nel tempo e più approfondita.

**Gm4 Availability** a livello di *reverse proxy*.

La configurazione di *Nginx* non prevede meccanismi di *high availability* pertanto potrebbe essere un *single point of failure* e un possibile target di *dDOS* [A.0.76].

## 4.4 Threat analysis

Questa sezione espone la *threat analysis* per *Mainflux* e *UrbanIoT*, utilizzando come riferimento la tassonomia di **ENISA** [A.0.51] e il framework definito dal progetto **H2020-CONCORDIA** [A.0.50].

Le minacce sono categorizzate in base alle aree di interesse, cioè quelle relative ai servizi di sicurezza forniti da *Mainflux* per *UrbanIoT* e quelle dirette a *Mainflux* stesso.

### 4.4.1 Mainflux

Le famiglie di *threats* individuate per Mainflux, sono:

- **Interception and unauthorized acquisition**: a livello di comunicazione tra componenti base e servizi Mainflux.

  - **Networking**: *eavesdropping [A.0.77], interception [A.0.78], hijacking [A.0.79]*
    * **CWE-200 - Information exposure**: questa debolezza si verifica quando informazioni sensibili vengono esposte a entità che non sono autorizzate ad accedervi.
    * **CWE-300 - Channel and path vulnerabilities**: riguarda le vulnerabilità nei canali di comunicazione che possono essere sfruttate per intercettare o dirottare i dati.
  - **System**: acquisizione non autorizzata di informazioni (data breach)
    * **CWE-284 - Improper access control**: questa debolezza si verifica quando un sistema non applica correttamente le politiche di controllo degli accessi, permettendo a utenti non autorizzati di accedere a risorse riservate.

- **Nefarious activity/abuse** a livello di servizi Mainflux.
  - **Networking**: *remote activities (execution)*
    * **CWE-94 - Improper control of generation of code**: questa debolezza si verifica quando un sistema permette l'inserimento e l'esecuzione di codice non autorizzato.
  - **System**: *identity theft or identity fraud, Denial of Service, malicious code/software/activity, code execution and injection.*
    * **Identity theft or identity fraud**
      · **CWE-522 - Insufficiently protected credentials**: questa debolezza si verifica quando le credenziali non sono adeguatamente protette, consentendo a malintenzionati di rubarle o utilizzarle in modo fraudolento.
    * **Denial of Service**
      · **CWE-400 - Uncontrolled resource consumption**: questa debolezza si verifica quando un sistema non riesce a controllare il consumo delle risorse, rendendosi vulnerabile agli attacchi di *Denial of Service*.
    * **Malicious code/software/activity**
      · **CWE-94 - Improper control of generation of code (code injection)**: [4.4.1]
    * **Code execution and injection**

· **CWE-20 - Improper input validation**: questa debolezza si verifica quando un sistema non valida correttamente l'input, permettendo l'inserimento di dati malformati che possono causare comportamenti imprevisti o dannosi.

· **CWE-77 - Improper neutralization of special elements used in a command (command injection)**: questa debolezza si verifica quando un sistema non neutralizza correttamente elementi speciali utilizzati in un comando, permettendo l'iniezione di comandi non autorizzati.

### 4.4.2 UrbanIoT

Di seguito vengono elencate le famiglie di *threats* per *UrbanIoT* e il *dispositivo edge*, le quali sono associate a diversi livelli: *IoT*, *system*, *network*, *data*, *application* e *user*.

- **Intentional physical damage**:
  - **IoT**: modifica del *dispositivo edge*
    - **CWE-1252 - Improperly controlled modification of object prototype attributes**: questa debolezza si verifica quando gli attributi di un oggetto possono essere modificati in modo non controllato, permettendo alterazioni non autorizzate.
    - **CWE-200 - Information exposure**: [4.4.1]
- **Unintentional damage/loss of information or IT assets**:
  - **System**: progettazione inadeguata
    - **CWE-250 - Execution with unnecessary privileges**: questa debolezza si verifica quando il software viene eseguito con privilegi superiori a quelli necessari.
    - **CWE-710 - Improper adherence to coding standards**: questa debolezza si verifica quando il codice sorgente non aderisce agli standard di codifica appropriati, portando a potenziali vulnerabilità.
  - **Data**: perdita o condivisione di informazioni a causa di errori umani
    - **CWE-199 - Information management errors**: questa debolezza si verifica quando si verificano errori nella gestione delle informazioni, portando alla perdita o alla divulgazione involontaria di dati.
  - **Network**: gestione impropria di dispositivi e sistemi

        * **CWE-277 - Insecure inherited permissions**: questa debolezza si verifica quando i permessi ereditati non sono sicuri, portando a configurazioni di sicurezza errate.
    - **Application**: errore di configurazione della sicurezza
        * **CWE-16 - Configuration**: questa debolezza si verifica quando le configurazioni di sicurezza non sono corrette, permettendo potenziali exploit.

- **Poisoning**:
    - **IoT**: alterazione del *dispositivo edge*
        * **CWE-494 - Download of code without integrity check**: questa debolezza si verifica quando il codice viene scaricato senza un controllo di integrità, permettendo alterazioni non autorizzate.
    - **Data**: manomissione dei dati
        * **CWE-502 - Deserialization of untrusted data**: questa debolezza si verifica quando i dati vengono deserializzati [A.0.97] senza essere adeguatamente verificati, permettendo la manipolazione dei dati.

- **Failures and malfunctions**
    - **IoT**: a livello del *dispositivo edge*
        * **CWE-385 - Covert timing channel**: questa debolezza si verifica quando un canale temporale nascosto permette la trasmissione non autorizzata di informazioni.
    - **Network**: a livello di *networking*
        * **CWE-118 - Improper access of indexable resource**: questa debolezza si verifica quando una risorsa indicizzabile viene acquisita in modo non appropriato, portando a malfunzionamenti.

- **Interception and unauthorized acquisition**:
    - **IoT**: accesso al *dispositivo edge* di *UrbanIoT*
        * **CWE-294 - Authentication bypass by capture-replay**: questa debolezza si verifica quando un attaccante può bypassare l'autenticazione catturando e riutilizzando credenziali valide.
    - **Networking**: *eavesdropping [A.0.77], interception [A.0.78], hijacking [A.0.79]*

    - **CWE-300 - Channel and path vulnerabilities**: [4.4.1]
  - **Data**: acquisizione di informazioni non autorizzata (data breach)
    - **CWE-284 - Improper access control**: [4.4.1]
  - **Application**: esposizione di dati sensibili
    - **CWE-200 - Information exposure**: [4.4.1]

- **Nefarious activity and abuse**:
  - **System**: *identity theft or identity fraud, Denial of Service, malicious code/software/activity, code execution and injection (unsecured APIs), abuse of information leakage*
    - **CWE-522 - Insufficiently protected credentials**: [4.4.1]
    - **CWE-400 - Uncontrolled resource consumption**: [4.4.1]
    - **CWE-94 - Improper control of generation of code (code injection)**: [4.4.1]
    - **CWE-20 - Improper input validation**: [4.4.1]
    - **CWE-77 - Improper neutralization of special elements used in a command (command injection)**: [4.4.1]
    - **CWE-359 - Exposure of private information (privacy violation)**: questa debolezza si verifica quando informazioni private vengono esposte in modo non autorizzato.
  - **Application**: autenticazione e controllo degli accessi compromessi
    - **CWE-287 - Improper authentication**: questa debolezza si verifica quando un sistema non autentica correttamente gli utenti, permettendo l'accesso non autorizzato.
    - **CWE-285 - Improper authorization**: questa debolezza si verifica quando un sistema non autorizza correttamente gli utenti, permettendo l'accesso a risorse senza i dovuti permessi.

- **Organizational threats**:
  - **IoT**: *malicious operator*
    - **CWE-1242 - Use of a risky cryptographic algorithm**: questa debolezza si verifica quando vengono utilizzati algoritmi crittografici rischiosi, permettendo a operatori malintenzionati di comprometterli.
  - **Application**: *malicious insider*

    - **CWE-706 - Use of incorrectly-resolved name or reference**: questa debolezza si verifica quando vengono utilizzati nomi o riferimenti risolti in modo errato, permettendo a insider malintenzionati di sfruttare la vulnerabilità.

- **User**: *skill shortage*
    - **CWE-399 - Resource management errors**: questa debolezza si verifica quando si verificano errori nella gestione delle risorse, spesso dovuti a una carenza di competenze adeguate.

## 4.5 Analisi della sicurezza per strati

Come definito nella metodologia 2.3.3, per ogni strato tecnologico di *UrbanIoT* si applica una procedura di analisi della sicurezza ottimizzata per quel tipo specifico di tecnologia.

A tal proposito sono state sviluppate apposite probe *MoonCloud* [3.1] per ogni tool già presentato nella sezione 3.2, le quali saranno descritte secondo il loro flusso di esecuzione.

### 4.5.1 Gosec

La probe clona un *repository git* [A.0.21] pubblico o privato (utilizzando un *token Oauth* [A.0.90]) contenente il codice sorgente in *Go* da analizzare, in questo caso il codice sorgente di backend di *UrbanIoT* e il codice sorgente di *Mainflux*.

Successivamente lancia il tool *Gosec* su tale target, escludendo la cartella "*test*" in quanto spesso contiene codice che può essere percepito da *Gosec* come fonte di potenziali vulnerabilità e ciò può portare a un numero elevato di falsi positivi, riducendo l'efficacia dell'analisi.

*Gosec* genera un report *JSON* contenente:

- **Golang errors**: riporta errori di compilazione o problemi che *Gosec* ha incontrato durante l'analisi del codice. Questi errori suggeriscono, ad esempio, che ci sono riferimenti a variabili o funzioni non definite nel codice sorgente e questo potrebbe essere dovuto a dipendenze mancanti oppure ad errori di sintassi;
- **Issues**: elenco di problemi di sicurezza rilevati, ognuno composto da diversi campi tra cui:
    - **file**: il percorso del file in cui è stata rilevata la vulnerabilità;

  - **severity**: livello di gravità della vulnerabilità (*CRITICAL, HIGH, MEDIUM, LOW*);
  - **confidence**: livello di confidenza che *Gosec* ha nella precisione della segnalazione (*HIGH, MEDIUM, LOW*);
  - **cwe**: identificatore CWE [A.0.43] della vulnerabilità;
  - **details**: descrizione dettagliata del problema di sicurezza rilevato;
  - **line and column**: numero di riga e colonna nel file in cui è stata rilevata la vulnerabilità.

- **Stats**: statistiche sull'analisi eseguita, come il numero di file analizzati, il numero di linee di codice esaminate e il tempo totale impiegato per l'analisi.

Tuttavia, il file *JSON* generato contiene anche dati che non si riferiscono direttamente a vulnerabilità (*Golang errors* e *stats*), inoltre la sezione *issues* contiene dati duplicati, ad esempio una stessa *CWE* con stessa *severity* e *confidence* può ripetersi più volte a causa di diverse occorrenze nell'intero progetto.

A tal proposito è stato realizzato uno script *Python* che prende in input un report *JSON* di *Gosec* e ne restituisce un altro più corto e con dati più significativi.

In particolare, vengono rimosse le sezioni *Golang errors* e *stats* e per ogni *CWE* vengono raggruppate tutte le occorrenze che si presentano nei vari file.

Il risultato finale è un file *JSON* composto, ad esempio, da queste occorrenze:

```
{
  "Issues": [
    {
      "CWE": "79",
      "Targets": {
        "file1.go": [{"line": 10, "column": 5}],
        "file2.go": [{"line": 20, "column": 15}]
      },
      "severity": "HIGH",
      "confidence": "MEDIUM",
      "details": "XSS vulnerability"
    },
    {
      "CWE": "89",
      "Targets": {
        "file1.go": [{"line": 30, "column": 25}]
      },
```

```
            "severity": "CRITICAL",
            "confidence": "HIGH",
            "details": "SQL injection vulnerability"
        }
    ]
}
```

Infine, la determinazione del successo o fallimento della *probe* è basata sulla valutazione di una soglia del numero di occorrenze di una determinata *severity*.

Ad esempio, se al termine della valutazione è presente un numero di occorrenze con *severity MEDIUM* maggiore di una determinata soglia, allora essa fallisce e nella dashboard di *MoonCloud* sarà segnalato con il colore rosso, viceversa con il colore verde.

### 4.5.2 Trivy per le immagini Docker

La probe scarica un file *Docker compose* [A.0.10], in questo caso quello di *UrbanIoT*, ed eventualmente un file contenente i valori delle variabili dichiarate nel file *Docker compose* al fine di effettuare opportune sostituzioni.

Successivamente la *probe* estrae dal file *Docker compose* l'elenco delle immagini dichiarate nei *service* di tale file e procede a lanciare il tool *Trivy* su ogni immagine, il quale si occuperà di eseguire una scansione focalizzata esclusivamente sulla ricerca di vulnerabilità note nei pacchetti software e nelle dipendenze.

In particolare, per ogni immagine, *Trivy* genera un report *JSON* contenente:

- **Metadata**: informazioni sul contesto e configurazione della scansione eseguita;
- **Vulnerabilities**: elenco di problemi di sicurezza rilevati, ognuno composto da diversi campi tra cui:
  - **VulnerabilityID**: identificatore *CVE* [A.0.44] della vulnerabilità;
  - **Title**: titolo descrittivo della vulnerabilità;
  - **Description**: descrizione dettagliata della vulnerabilità;
  - **severity**: livello di gravità della vulnerabilità (*CRITICAL, HIGH, MEDIUM, LOW, UNKNOWN, NONE*);
  - **cwe**: identificatore *CWE* [A.0.43] della vulnerabilità;

- **VendorSeverity**: severità della vulnerabilità determinata da diverse fonti ognuna delle quali può valutare la severità in base ai propri criteri e metodi di valutazione;
- **CVSS**: punteggi e vettori *Common Vulnerability Scoring System* (CVSS) V2 e V3 [A.0.45] forniti da diverse fonti.

Tuttavia, la sezione *Vulnerabilities* contiene anche metadati superflui per le valutazioni di sicurezza.

A tal proposito è stato realizzato uno script *Python* che prende in input un report *JSON* di *Trivy* e ne restituisce un altro più corto e con dati più significativi.

Inoltre per evitare *CVE* duplicate per diverse immagini, lo script raggruppa più target per una determinata *CVE*.

Il risultato finale è un file *JSON* composto, ad esempio, da queste occorrenze:

```
{
    "CVE-2021-33194": {
        "Targets": [
            "auth:0.12.1",
            "influxdb-writer:0.12.1",
            "http:0.12.1"
        ],
        "Details": {
            "Title": "golang: x/net/html: infinite loop in ParseFragment",
            "Description": "...",
            "Severity": "HIGH",
            "CWE": [
                "CWE-835"
            ],
            "V2Score": 5.0,
            "V3Score": 7.5
        }
    },
    ...
}
```

Infine, come già definito per altre probe, la determinazione del successo o fallimento della probe è basata sulla valutazione di una soglia del numero di occorrenze di una determinata *severity* [4.5.1].

### 4.5.3 Trivy per il codice sorgente

La probe clona un *repository git* pubblico o privato (utilizzando un *token Oauth* [A.0.90]) contenente il codice sorgente in un linguaggio supportato da *Trivy*, in questo caso il codice sorgente di backend (*Go*) e frontend (*AngularJS*) di *UrbanIoT* e il codice sorgente di *Mainflux* (*Go*).

Successivamente lancia il tool *Trivy* su tale target e, allo stesso modo di *Trivy* per le immagini docker [4.5.2], viene generato un output *JSON* analogo e vengono effettuate le stesse valutazioni per determinare il successo o il fallimento della probe.

### 4.5.4 Trivy per il dispositivo edge

#### Emulazione del dispositivo

Per l'effettuazione delle analisi di sicurezza, non potendo operare sul *dispositivo edge* in maniera fisica oppure tramite connessione, è stato utilizzato l'emulatore *open source* **QEMU**[25], il quale consente di eseguire sistemi operativi compilati per un'architettura hardware (ad esempio *x86* e *ARM*) su un'altra architettura.

A tal proposito è stato utilizzato per emulare il processore *ARM* e le periferiche associate, nella sua versione per *Docker* in modo da avere i seguenti vantaggi:

- **Isolamento**: i container *Docker* sono isolati dal sistema host, permettendo di evitare conflitti di dipendenze;
- **Portabilità**: i container possono essere eseguiti su qualsiasi sistema che supporta *Docker*;
- **Automatizzazione**: è possibile definire l'intero ambiente di emulazione in un *Dockerfile*, rendendo semplice la sua riproducibilità.

Tuttavia prima di procedere con l'emulazione vera e propria sono state effettuate alcune modifiche al sistema operativo da emulare al fine di creare un nuovo utente con *privilegi di amministratore* (necessari per alcune fasi dell'analisi) e di modificare alcune impostazioni esistenti che avrebbero interferito con l'emulazione.

In particolare, con uno script *Bash*, l'immagine del sistema operativo del *Raspberry Pi* è stata montata su una directory temporanea (*/mnt*) e sono state eseguite le seguenti operazioni:

- Modifica al file */mnt/etc/ld.so.preload* per evitare il caricamento delle librerie condivise specificate in esso, le quali avrebbero interferito con il processo di emulazione;

[25] https://qemu.org

- Creazione di un nuovo utente nel file */mnt/etc/passwd* e concessione dei privilegi di amministratore nel file */mnt/etc/sudoers*;
- Modifica della modalità di funzionamento del *file system* nel file */mnt/etc/fstab* da lettura a scrittura;
- Sostituzione del file */mnt/etc/ssh/sshd_config* con una configurazione che consente l'accesso *SSH* tramite meccanismo chiave pubblica/privata;
- Creazione del file */mnt/home/.ssh/authorized_keys* contenente una chiave pubblica corrispondente alla chiave privata usata per la successiva connessione *SSH*.

Successivamente, per l'emulazione con *QEMU*, è stato configurato un *Dockerfile* che partendo da un'immagine di base di *Ubuntu 20.04* installa alcuni pacchetti necessari: *qemu-system-aarch64*[26] per l'emulazione dell'*architettura ARM*, *fdisk*[27], *wget*[28], *mtools*[29], e *xz-utils*[30] necessari per gestire e manipolare l'immagine del sistema operativo *Raspberry Pi*.

In seguito, l'immagine del *Raspberry Pi* è stata copiata nella directory di lavoro del container e ridimensionata alla potenza di due successiva, garantendo una dimensione ottimale per l'emulazione con *QEMU*.

Poi, utilizzando *fdisk* e *mtools*, il *Dockerfile* ha identificato la partizione *FAT32* [A.0.82] nell'immagine e configurato l'ambiente per estrarre i file necessari.

Sono stati, quindi, estratti il *device tree blob* [A.0.83] e l'immagine del kernel (*kernel8.img*) necessari per l'avvio del sistema emulato.

Infine, per consentire l'accesso remoto al sistema emulato è stato configurato *SSH* specificando il *forwarding* della porta per consentire l'accesso *SSH* dal sistema host al sistema emulato.

Da notare che il *Raspberry Pi 3B+*[31], utilizzato dal *dispositivo edge*, è dotato di 1 GB di *RAM* e un processore *quad-core ARM Cortex-A53*[32], pertanto al fine di ottenere un'emulazione del sistema più fedele possibile all'hardware reale ed evitare *undefined behaviors*, *QEMU* impone tali limitazioni anche nella configurazione per l'emulazione. Tali limitazioni comportano, tuttavia, rallentamenti generali non trascurabili durante l'esecuzione dei tool di sicurezza nel sistema emulato.

[26] https://qemu.org/docs/master/system/target-arm.html
[27] https://www.gnu.org/software/fdisk
[28] https://gnu.org/software/wget
[29] https://gnu.org/software/mtools
[30] https://tukaani.org/xz
[31] https://raspberrypi.com/products/raspberry-pi-3-model-b
[32] https://arm.com/products/silicon-ip-cpu/cortex-a/cortex-a53

**Esecuzione della probe**

La probe si connette via *SSH* al *dispositivo edge* emulato con *QEMU*, scarica una versione di *Trivy*, specificata da utente, procede con l'installazione con il tool *dpkg*[33] e lo esegue nella modalità *rootfs*[34] specificando come path di inizio la radice dell'intero sistema operativo (/).

Successivamente, la probe estrae dal sistema il report *JSON* generato e, come già descritto, utilizza lo script *Python* per generare un report finale più significativo, sul quale vengono effettuate le valutazioni già descritte per determinare il successo o il fallimento della probe.

Da notare che a seguito di diverse esecuzioni sperimentali e considerate le già descritte limitazioni dell'emulazione, sono stati esclusi alcune directory e file con determinate estensioni per evitare un tempo di esecuzione totale nell'ordine delle ore, non di certo compatibile con i tempi accettabili per verifiche di sicurezza continue.

In particolare le directory e i file esclusi riguardano:

- La directory */mnt/ramdisk*;
- I file con le estensioni *jar, war, par, ear*.

### 4.5.5 Lynis per il dispositivo edge

La probe si connette via *SSH* al *dispositivo edge* emulato con *QEMU* [4.5.4], scarica una versione di *Lynis*, specificata da utente, e lo esegue con l'opzione *nocolors* al fine di salvare l'output senza la formattazione dei colori per evitare un inutile *overhead*.

Da notare che le opzioni di formattazione e formato dell'output di *Lynis* sono disponibili solo nella versione a pagamento, pertanto è stato realizzato uno script *Python* che prende in input il report in formato *txt* generato da *Lynis* e genera un report *JSON* analogo contenente le informazioni già descritte [3.2.3].

Infine la probe valuta il valore *hardening index* con una soglia definita da utente per determinare il successo o fallimento della stessa.

Da notare che per le limitazioni già descritte nella sezione 4.5.4, sono stati esclusi alcuni *test case*:

- **PKGS-7345**: verifica la presenza di pacchetti non rimossi, i quali determinano la presenza di file non utilizzati;

[33] https://wiki.debian.org/it/dpkg
[34] https://aquasecurity.github.io/trivy/v0.52/docs/target/rootfs/

- **PKGS-7392**: verifica la presenza di pacchetti software vulnerabili nel sistema, identificando quelli con falle di sicurezza note per le quali sono già disponibili aggiornamenti;
- **NETW-2600**: verifica la configurazione di *IPv6* [A.0.92] per determinare se il protocollo è abilitato o disabilitato;
- **HTTP-6708**: esamina le impostazioni di configurazione di *Nginx* per identificare eventuali errori di configurazione o impostazioni non sicure che potrebbero compromettere la sicurezza del server;
- **CRYP-7902**: verifica la data di scadenza dei certificati *SSL*[35] per evitare l'uso improprio di certificati scaduti;
- **KRNL-6000**: confronto tra i valori correnti delle variabili di sistema e una lista di valori considerati best practice.

### 4.5.6 Bandit per gli script Python

La probe clona un *repository git* pubblico o privato (utilizzando un *token Oauth* [A.0.90]) contenente il codice sorgente in *Python* da analizzare, in questo caso un *repository* contenente alcuni script contenuti nel *dispositivo edge*.

Successivamente lancia il tool *Bandit* su tale target, il quale genera un report *JSON* contenente:

- **errors**: riporta errori di compilazione o problemi che *Bandit* ha incontrato durante l'analisi del codice;
- **metrics**: distribuzione della *confidence* e *severity* sia a livello generale, che per ogni singolo file analizzato, numero totale di linee di codice analizzate, linee di codice ignorate e il numero di test saltati;
- **results**: elenco dei problemi di sicurezza rilevati, ognuno composto da diversi campi tra cui:
  - **filename**: il percorso del file in cui è stata rilevata la vulnerabilità;
  - **issue_severity**: livello di gravità della vulnerabilità (*CRITICAL, HIGH, MEDIUM, LOW*);
  - **issue_confidence**: livello di confidenza che *Bandit* ha nella precisione della segnalazione (*HIGH, MEDIUM, LOW*);

[35] https://ssl.com

- **issue_cwe**: identificatore *CWE* [A.0.43] della vulnerabilità;
- **issue_text**: descrizione dettagliata della vulnerabilità;
- **line and column**: numero di riga e colonna nel file in cui è stata rilevata la vulnerabilità.

Tuttavia, il file *JSON* generato contiene anche dati che non si riferiscono direttamente a vulnerabilità (*errors* e *metrics*), inoltre la sezione *results* può contenere dati duplicati, ad esempio una stessa *CWE* con stessa *confidence* e *severity* può ripetersi più volte a causa di diverse occorrenze nell'intero progetto.

A tal proposito è stato realizzato uno script *Python*, simile a quello sviluppato per *Gosec* [4.5.1], che prende in input un report *JSON* di *Bandit* e ne restituisce un altro più corto e con dati più significativi.

In particolare vengono rimosse le sezioni *errors* e *metrics* e per ogni *CWE* vengono raggruppate tutte le occorrenze che si presentano nei vari file.

Il risultato finale è un file *JSON* composto, ad esempio, da queste occorrenze:

```
{
    "Issues": [
        {
            "CWE": "79",
            "Targets": {
                "file1.go": [{"line": 10, "column": 5}],
                "file2.go": [{"line": 20, "column": 15}]
            },
            "severity": "HIGH",
            "confidence": "MEDIUM",
            "details": "XSS vulnerability"
        },
        {
            "CWE": "89",
            "Targets": {
                "file1.go": [{"line": 30, "column": 25}]
            },
            "severity": "CRITICAL",
            "confidence": "HIGH",
            "details": "SQL injection vulnerability"
        }
    ]
}
```

Infine, come già definito per altre probe, la determinazione del successo o fallimento della probe è basata sulla valutazione di una soglia del numero di occorrenze di una determinata *severity* [4.5.1].

## 4.6 Report di aggregazione

Ottenuti i report generati dai diversi tool su diversi target, si procede ad aggregare le vulnerabilità individuate basandosi sui *threats* individuati nella *threat analysis* [4.4].

A tal proposito è stato sviluppato uno script *Python* che si occupa di produrre il report sull'aggregazione partendo da un unico file *JSON* di configurazione composto da uno o più macro componenti definiti nella *threat analysis* (cioè *Mainflux* e *UrbanIoT*), i percorsi nel *file system* dove si trovano i file di report generati dai tool utilizzati e un elenco di *threats* con relative *CWE*.

Il report in output è suddiviso nelle seguenti sezioni:

### 4.6.1 Analisi complessiva delle vulnerabilità e delle minacce

- **CVE e CWE univoche**: numero di *CVE* e *CWE* univoche individuabili in tutti i file di report, considerando che gli strumenti di analisi statica possono individuare la stessa vulnerabilità più volte;
- **Distribuzione delle severità**: distribuzione delle severità delle vulnerabilità individuate in cinque livelli (*CRITICAL, HIGH, MEDIUM, LOW, UNKNOWN*);
- **Threat analysis**: riepilogo della *threat analysis* con l'elenco dei *threats* e *CWE* univoche ricercate in base a quanto definito nella *threat analysis*, dato che diversi *threat* possono essere rappresentati da medesime *CWE*;
- **CWE e CVE univoche individuate**: elenco delle *CWE* univoche individuate tra quelle ricercate, suddivise per i macro componenti definiti e distribuzione delle *CVE* ad esse appartenenti in cinque livelli (*CRITICAL, HIGH, MEDIUM, LOW, UNKNOWN*).

### 4.6.2 Livello di rischio complessivo

Definita una scala di valutazione per le *severity*: *CRITICAL: 9, HIGH: 7, MEDIUM: 5, LOW: 3, UNKNOWN: 1*, il livello di rischio complessivo è calcolato

in base a tutte le vulnerabilità individuate moltiplicando i valori della scala di valutazione per il *V3Score* e dividendo per rischio teorico massimo:

$$risk\_score = \frac{\sum_{i=1}^{n} (s_i \cdot v_i)}{(n_v \cdot s_{\max} \cdot v_{\max}) + (n_{nv} \cdot s_{\max} \cdot 1)} \cdot 100 \quad (1)$$

dove:

- $s_i$: punteggio di severità della vulnerabilità $i$;
- $v_i$: *V3Score* della vulnerabilità $i$, se presente, in quanto tale valore potrebbe non essere registrato nel database *NIST* [A.0.52] oppure potrebbe non essere supportato dallo strumento che ha effettuato l'analisi statica;
- $n_v$: numero di vulnerabilità con *V3Score*;
- $n_{nv}$: numero di vulnerabilità senza *V3Score*;
- $s_{\max}$: valore massimo della scala di valutazione per le *severity* (9);
- $v_{\max}$: *V3Score* massimo (10).

Il rischio teorico massimo si calcola come il prodotto tra il punteggio di severità massimo (9) e il valore *V3Score* massimo (10).

Tuttavia, poiché non tutte le vulnerabilità individuate dispongono di un *V3Score*, il calcolo del rischio teorico massimo è stato adattato per tenere conto di questa variazione.

Pertanto, il rischio teorico massimo è suddiviso in due componenti: una per le vulnerabilità che hanno un *V3Score*, calcolata utilizzando il *V3Score* massimo, e una per quelle che ne sono prive, per le quali il *V3Score* massimo è considerato pari a 1.

### 4.6.3 Livello di rischio per ogni CWE

Il livello di rischio per ogni *CWE* è calcolato in maniera simile al livello di rischio complessivo, con la differenza che il calcolo viene effettuato iterativamente per ogni insieme di vulnerabilità delle *CWE* identificate e in seguito ogni valore di rischio per *CWE* viene normalizzato sulla base della distribuzione delle vulnerabilità tra le *CWE* identificate:

$$risk\_score\_CWE_j = \frac{\left( \frac{\sum_{i=1}^{n_j} (s_{ij} \cdot v_{ij})}{(n_{vj} \cdot s_{\max} \cdot v_{\max}) + (n_{nvj} \cdot s_{\max} \cdot 1)} \right) \cdot n_j}{n_{\max}} \cdot 100 \quad (2)$$

dove:

- $s_{ij}$: punteggio di severità della vulnerabilità $i$ appartenente alla *CWE* $j$;
- $v_{ij}$: *V3Score* della vulnerabilità $i$ appartenente alla *CWE* $j$, se presente;
- $n_{vj}$: numero di vulnerabilità con *V3Score* nella *CWE* $j$;
- $n_{nvj}$: numero di vulnerabilità senza *V3Score* nella *CWE* $j$;
- $n_j$: numero di vulnerabilità nella *CWE* $j$;
- $n_{\max}$: numero massimo di vulnerabilità tra le *CWE*;
- $s_{\max}$: valore massimo della scala di valutazione per le *severity* (9);
- $v_{\max}$: *V3Score* massimo (10).

Da notare che tale normalizzazione produce un livello di rischio proporzionato alla quantità di vulnerabilità presenti in ciascuna *CWE* garantendo che il punteggio di rischio rifletta accuratamente le severità, gli score e la frequenza delle vulnerabilità.

Ciò evita, ad esempio, che le *CWE* con un numero ridotto di vulnerabilità abbiano un rischio eccessivamente elevato rispetto alle *CWE* con un numero maggiore di vulnerabilità, solo a causa della quantità numerica.

### 4.6.4 Livello di rischio per ogni threat

Il livello di rischio per ogni *threat* è calcolato in maniera simile al livello di rischio per *CWE*, con la differenza che il calcolo viene effettuato iterativamente per ogni *CWE* di ogni *threat* individuato nella *threat analysis*, e che la normalizzazione coinvolge il numero massimo di vulnerabilità tra i valori delle somme delle vulnerabilità per ogni *CWE* di un determinato *threat*:

$$risk\_score\_threat_k = \frac{\left( \frac{\sum_{j=1}^{m_k} \sum_{i=1}^{n_{jk}} (s_{ijk} \cdot v_{ijk})}{(n_{vjk} \cdot s_{\max} \cdot v_{\max}) + (n_{nvjk} \cdot s_{\max} \cdot 1)} \right) \cdot N_k}{N_{\max}} \cdot 100 \tag{3}$$

dove:

- $s_{ijk}$: punteggio di severità della vulnerabilità $i$ appartenente alla *CWE* $j$ del *threat* $k$;
- $v_{ijk}$: *V3Score* della vulnerabilità $i$ appartenente alla *CWE* $j$ del *threat* $k$, se presente;
- $n_{vjk}$: numero di vulnerabilità con *V3Score* nella *CWE* $j$ del *threat* $k$;

- $n_{nvjk}$: numero di vulnerabilità senza *V3Score* nella *CWE j* del *threat k*;
- $n_{jk}$: numero di vulnerabilità nella *CWE j* del *threat k*;
- $m_k$: numero di *CWE* nel *threat k*;
- $N_k$: somma delle vulnerabilità per tutte le *CWE* del *threat k*;
- $N_{\max}$: somma massima delle vulnerabilità tra tutti i *threat*;
- $s_{\max}$: valore massimo della scala di valutazione per le *severity* (9);
- $v_{\max}$: *V3Score* massimo (10).

## 4.7 Monitoraggio continuo

Le *probe* sviluppate possono essere facilmente integrate nella *pipeline* di sviluppo di *UrbanIoT* sfruttando le *API* di *MoonCloud* che consentono l'invocazione di una determinata *probe* e la restituzione del risultato in modalità bloccante/non bloccante.

A seconda di tale modalità, in caso di fallimento della *probe*, la *pipeline* può stopparsi oppure proseguire e in ogni caso è possibile visualizzare i dettagli delle analisi di sicurezza nella dashboard di *MoonCloud*, rendendo più efficace la consultazione di report dettagliati rispetto a normali *artifact* di una *pipeline*.

# Capitolo 5

# Analisi dei risultati

Questo capitolo è suddiviso in cinque sezioni.

La prima sezione fornisce un riepilogo dei risultati ottenuti durante l'analisi delle vulnerabilità, con l'obiettivo di presentare un quadro complessivo delle vulnerabilità individuate, categorizzate per *severità* [A.0.46] e classi di *CWE* [A.0.43].

La seconda sezione si concentra sulle vulnerabilità suddivise per target specifici, elencando le vulnerabilità rilevate per ciascun target analizzato, suddivise per *severità* [A.0.46] e classi di *CWE* [A.0.43].

La terza sezione presenta i risultati delle vulnerabilità suddivise per strumenti di analisi, evidenziando come i diversi strumenti abbiano contribuito all'individuazione delle vulnerabilità, fornendo un confronto tra i risultati ottenuti dai vari tool utilizzati.

La quarta sezione riguarda i risultati dell'aggregazione, dove vengono sintetizzate le vulnerabilità individuate e aggregate in base alle famiglie di *CWE* definite nella *threat analysis* [A.0.32], inoltre, viene mostrata la distribuzione delle severità delle vulnerabilità appartenenti alle diverse famiglie di *CWE*.

Infine, la quinta sezione è dedicata all'analisi del rischio. Questa parte illustra l'applicazione delle formule per il calcolo del livello di rischio complessivo e per ogni *CWE*, nonché il livello di rischio per ciascuna minaccia individuata, al fine di fornire una valutazione quantitativa del rischio associato alle vulnerabilità rilevate.

Da notare che la lista dettagliata delle vulnerabilità individuate non è volutamente inclusa nell'appendice per ragioni di sicurezza e privacy.

In particolare, essendo che l'intero codice sorgente analizzato è software proprietario, concesso esclusivamente per le analisi di sicurezza, anche i dettagli specifici delle vulnerabilità rilevate non possono essere divulgati, dato che la pubblicazione di tali informazioni potrebbe compromettere la sicurezza dei sistemi coinvolti e violare accordi di riservatezza con i proprietari del codice.

## 5.1 Riepilogo dei risultati individuati

A seguito dell'esecuzione delle probe sui target specificati, attraverso lo script di aggregazione definito nell'implementazione 4.6, è stato rilevato un totale di **924** *vulnerabilità univoche*, cioè occorrenze univoche di vulnerabilità che possono essere presenti in più report dei tool di sicurezza utilizzati.

Le vulnerabilità individuate appartengono, complessivamente, a **143** *CWE univoche*, con la seguente distribuzione delle *severity*:

- **CRITICAL**: 67/924
- **HIGH**: 260/924
- **MEDIUM**: 227/924
- **LOW**: 360/924
- **UNKNOWN**: 10/924

## 5.2 Vulnerabilità suddivise per target

In questa sezione vengono presentati i risultati relativi al numero di vulnerabilità individuate per ciascun target.

Le vulnerabilità sono categorizzate in base alla loro *severity*, distinguendo tra il numero totale di vulnerabilità rilevate e il numero di quelle specificamente associate alle *CWE* identificate nella *threat analysis*.

### 5.2.1 Immagini Docker di Mainflux

Nella Tabella 1 sono riportate le vulnerabilità individuate attraverso l'analisi delle *immagini Docker* pubbliche dei componenti di *Mainflux*.

Questa analisi è stata condotta sulle versioni effettivamente utilizzate da *UrbanIoT*, anziché sulle ultime versioni disponibili, per garantire una maggiore accuratezza e rilevanza dei risultati.

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | 1 | - |
| High | 15 | 6 |
| Medium | 7 | 4 |
| Low | - | - |
| Unknown | - | - |

Tabella 1: Analisi delle *immagini Docker* di *Mainflux* con *Trivy*

### 5.2.2 Immagini Docker di UrbanIoT

Nella tabella 2 sono elencate le vulnerabilità riscontrate analizzando le *immagini Docker* di *UrbanIoT* che comprendono immagini del backend e servizi base di *Mainflux* usati anche dal backend.

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | 49 | 3 |
| High | 131 | 16 |
| Medium | 91 | 20 |
| Low | - | - |
| Unknown | - | - |

Tabella 2: Analisi delle *immagini Docker* di *UrbanIoT* con *Trivy*

### 5.2.3 Codice sorgente di Mainflux e UrbanIoT

Di seguito viene presentata l'analisi delle vulnerabilità del codice sorgente di *Mainflux* e del backend e frontend di *UrbanIoT*.

#### Mainflux

Nelle tabelle 3 e 4 sono elencate le vulnerabilità del codice sorgente *Go* relativo ai componenti di *Mainflux* delle stesse versioni usate in produzione.

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | 2 | - |
| High | 24 | 12 |
| Medium | 31 | 11 |
| Low | 4 | 1 |
| Unknown | - | - |

Tabella 3: Analisi del codice sorgente di *Mainflux* con *Trivy*

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | - | - |
| High | 3 | - |
| Medium | 3 | - |
| Low | 1 | - |
| Unknown | - | - |

Tabella 4: Analisi del codice sorgente di *Mainflux* con *Gosec*

**UrbanIoT - backend**

Nelle tabelle 5 e 6 sono elencate le vulnerabilità del codice sorgente *Go* del backend di *UrbanIoT* individuate, rispettivamente, da *Trivy* e *Gosec*

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | 2 | 2 |
| High | 9 | 8 |
| Medium | 9 | 8 |
| Low | 1 | - |
| Unknown | - | - |

Tabella 5: Analisi del codice sorgente del backend di *UrbanIoT* con *Trivy*

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | - | - |
| High | 4 | - |
| Medium | 5 | - |
| Low | - | - |
| Unknown | - | - |

Tabella 6: Analisi del codice sorgente del backend di *UrbanIoT* con *Gosec*

#### UrbanIoT - frontend

Nella tabella 7 sono elencate vulnerabilità del codice sorgente *AngularJS* del frontend di *UrbanIoT*.

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | 1 | - |
| High | 1 | 1 |
| Medium | - | - |
| Low | - | - |
| Unknown | - | - |

Tabella 7: Analisi del codice sorgente del frontend di *UrbanIoT* con *Trivy*

### 5.2.4 Dispositivo edge UrbanIoT

#### Sistema operativo

Nella tabella 8 sono elencate vulnerabilità individuate nel *file system* del *dispositivo edge* di *UrbanIoT*.

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | 18 | 1 |
| High | 134 | 11 |
| Medium | 134 | 36 |
| Low | 357 | 19 |
| Unknown | 10 | 4 |

Tabella 8: Analisi del *file system* del *dispositivo edge* di *UrbanIoT* con *Trivy*

Nella tabella 9 sono presenti il numero di test eseguiti e l'*hardening index score* di *Lynis*:

| Metrica | Valore |
|---|---|
| Numero test eseguiti | 242 |
| Hardening index score | 60 |

Tabella 9: Analisi del sistema operativo del *dispositivo edge* di *UrbanIoT* con *Lynis*

**Script Python**

Nella tabella 10 sono elencate vulnerabilità individuate negli script *Python* presenti nel *file system* del *dispositivo edge*.

| Severity | Vulnerabilità | Vulnerabilità threat analysis |
|---|---|---|
| Critical | - | - |
| High | - | - |
| Medium | 1 | - |
| Low | 4 | - |
| Unknown | - | - |

Tabella 10: Analisi degli script *Python* del *dispositivo edge* di *UrbanIoT* con *Bandit*

## 5.3 Vulnerabilità suddivise per strumenti di analisi

In questa sezione vengono presentati i risultati relativi al numero di vulnerabilità, per ogni *CWE*, individuate dai diversi strumenti di analisi

### 5.3.1 Mainflux

| CWE | Trivy Go | Gosec | Trivy Docker | Totale |
|---|---|---|---|---|
| CWE-20 | 3 | - | - | 3 |
| CWE-77 | - | - | - | - |
| CWE-94 | - | - | - | - |
| CWE-200 | - | - | - | - |
| CWE-284 | - | - | - | - |
| CWE-300 | - | - | - | - |
| CWE-400 | 7 | - | 3 | 10 |
| CWE-522 | - | - | - | - |

Tabella 11: Vulnerabilità *CWE* nei target di *Mainflux* suddivise per tool

### 5.3.2 UrbanIoT

| CWE | Bandit | Gosec | Trivy Angular | Trivy Docker | Trivy Go | Trivy Edge | Total |
|---|---|---|---|---|---|---|---|
| CWE-20 | - | - | - | 5 | - | 14 | 19 |
| CWE-77 | - | - | - | - | - | 2 | 2 |
| CWE-118 | - | 1 | - | - | - | - | 1 |
| CWE-200 | - | - | - | 8 | - | 26 | 34 |
| CWE-284 | - | - | - | - | - | 3 | 3 |
| CWE-285 | - | - | - | 1 | - | 1 | 2 |
| CWE-287 | - | - | - | 4 | 3 | 9 | 16 |
| CWE-300 | - | - | - | 1 | - | 1 | 2 |
| CWE-359 | - | - | - | 1 | - | - | 1 |
| CWE-385 | - | - | - | 3 | - | - | 3 |
| CWE-399 | - | - | - | - | - | 1 | 1 |
| CWE-400 | - | - | 1 | 7 | 4 | 19 | 31 |
| CWE-502 | 1 | - | - | 1 | 2 | 1 | 5 |
| CWE-522 | - | - | - | - | - | 1 | 1 |
| CWE-706 | - | - | - | 1 | 2 | - | 3 |

Tabella 12: Vulnerabilità *CWE* nei target di *UrbanIoT* suddivise per tool

## 5.4 Risultati dell'aggregazione

A partire dalle **25** *famiglie di CWE* da ricercare, definite nella *threat analysis*, ne sono state individuate complessivamente **15**, di cui **7/15** riguardano *Mainflux* e **15/15** *UrbanIoT* e alle quali afferiscono, complessivamente, **97** vulnerabilità univoche. Inoltre, la distribuzione delle *severity* delle vulnerabilità, appartenenti alle **15** *famiglie di CWE* individuate, è la seguente:

- **CRITICAL**: 3/97
- **HIGH**: 25/97
- **MEDIUM**: 45/97
- **LOW**: 20/97
- **UNKNOWN**: 4/97

## 5.5 Analisi del rischio

Applicando la formula del livello di rischio complessivo [1] ai dati sperimentali, è stato ottenuto un valore pari a **38/100**.

Applicando la formula del livello di rischio per ogni *CWE* [2] ai dati sperimentali, sono stati ottenuti i seguenti livelli di rischio:

| **CWE** | **Livello di rischio (%)** |
|---|---|
| **CWE-400** | 40/100 |
| **CWE-20** | 32/100 |
| **CWE-287** | 22/100 |
| **CWE-200** | 15/100 |
| **CWE-502** | 6/100 |
| **CWE-284** | 4/100 |
| **CWE-706** | 3/100 |
| **CWE-385** | 3/100 |
| **CWE-118** | 2/100 |
| **CWE-285** | 2/100 |
| **CWE-77** | 2/100 |
| **CWE-300** | 1/100 |
| **CWE-399** | 1/100 |
| **CWE-359** | 0/100 |
| **CWE-522** | 0/100 |

Tabella 13: Livello di rischio per ogni *CWE*

Applicando la formula del livello di rischio per ogni *threat* [3] ai dati sperimentali, sono stati ottenuti i seguenti livelli di rischio:

| Threat | Livello di rischio (%) |
|---|---|
| **Nefarious activity/abuse** | 40/100 |
| **Interception and unauthorized acquisition** | 8/100 |
| **Intentional physical damage** | 6/100 |
| **Poisoning** | 2/100 |
| **Organizational threats** | 2/100 |
| **Failures/malfunction** | 1/100 |
| **Unintentional damage/loss of information or IT assets** | 0/100 |

Tabella 14: Livello di rischio per ogni *threat*

# Capitolo 6

# Conclusioni

L'obiettivo principale di questa tesi è stato sviluppare una metodologia analitica per l'analisi di sicurezza nelle *architetture a microservizi*, rispondendo all'esigenza di affrontare i rischi in contesti di elevata complessità e diversità tecnologica.

La metodologia proposta è stata progettata per fornire un quadro sistematico e stratificato in grado di adattarsi e reagire dinamicamente alle specificità di ciascun componente del sistema, garantendo una copertura di sicurezza comprensiva e dettagliata.

Lo stato dell'arte da cui è partita la ricerca ha evidenziato un'insufficiente applicazione delle tradizionali metodologie di analisi di sicurezza nelle architetture a microservizi, dato che i tool di analisi esistenti spesso mancano di una capacità di integrazione e di una visione globale necessaria per identificare e mitigare i rischi che emergono dall'interazione tra i vari *microservizi* di architetture complesse.

Per risolvere questo problema, è stata proposta una nuova metodologia che integra strumenti di analisi continua all'interno di *MoonCloud* [3.1], una piattaforma per la valutazione continua di conformità e di *assurance* [1.4] per applicazioni e infrastrutture ICT, che permette di sincronizzare la verifica del software con la *Continuous Integration* (CI) e il *Continuous Deployment* (CD), assicurando una valutazione costante e aggiornata del sistema software.

L'implementazione della metodologia è stata illustrata attraverso l'analisi di un caso di studio riguardante un sistema *IoT* complesso denominato *UrbanIoT* [4.1.2], un software di telegestione degli impianti di illuminazione pubblica e delle smart city.

In particolare, per ogni strato del sistema sono stati definiti i *requisiti di sicurezza* [A.0.35], è stata effettuata una *gap analysis* [A.0.31], ed è stata condotta una *threat analysis* dettagliata.

Sono stati, successivamente, utilizzati diversi strumenti di *analisi statica* [1.4.2], quali *Gosec* [3.2.1], *Bandit* [3.2.1], *Trivy* [3.2.2] e *Lynis* [3.2.3], per valutare la sicurezza dei componenti software e hardware coinvolti.

Inoltre, sono stati sviluppati strumenti per l'aggregazione dei risultati delle analisi di sicurezza, permettendo di calcolare livelli di rischio complessivi e specifici per ciascuna vulnerabilità identificata.

Dai risultati sperimentali ottenuti, emergono diverse osservazioni rilevanti sull'efficacia della metodologia proposta.

In particolare, l'analisi delle *immagini Docker* di *Mainflux* e *UrbanIoT* ha rilevato un numero significativo di vulnerabilità di elevata gravità (*CRITICAL* e *HIGH*), evidenziando la necessità di un monitoraggio continuo e approfondito.

Ad esempio, l'analisi delle *immagini Docker* di *UrbanIoT* ha identificato un totale di **49 vulnerabilità CRITICAL** e **131 vulnerabilità HIGH**, rispetto a **1 vulnerabilità CRITICAL** e **15 vulnerabilità HIGH** nelle *immagini Docker* di *Mainflux*.

L'analisi del codice sorgente ha rivelato che i componenti di *Mainflux* e *UrbanIoT* presentano vulnerabilità distribuite tra diverse classi di severità. Per il backend di *UrbanIoT*, *Trivy* ha identificato **2 vulnerabilità CRITICAL** e **9 vulnerabilità HIGH**, mentre *Gosec* ha rilevato **4 vulnerabilità HIGH** e questi dati confermano l'importanza di utilizzare strumenti multipli per ottenere una copertura più ampia delle potenziali vulnerabilità, dato che *Trivy* si concentra sulle dipendenze mentre *Gosec* sul codice sorgente.

Il *dispositivo edge* di *UrbanIoT* ha mostrato una concentrazione elevata di vulnerabilità di gravità variabile, infatti l'analisi con *Trivy* ha rilevato **18 vulnerabilità CRITICAL** e **134 vulnerabilità HIGH** nel *file system* del dispositivo, mentre *Lynis* ha prodotto un **hardening index score [A.0.61] pari a 60**, suggerendo margini significativi di miglioramento nelle configurazioni di sicurezza del dispositivo.

Le vulnerabilità individuate dai vari strumenti di *analisi statica* hanno evidenziato la prevalenza di alcune categorie di *CWE*, con una particolare concentrazione nelle **CWE-400** (*unrestricted resource consumption*), **CWE-20** (*improper input validation*), e **CWE-287** (*improper authentication*), categorie che, quindi, rappresentano le principali aree di rischio per il sistema analizzato, richiedendo interventi specifici per la mitigazione.

Infine l'analisi del **rischio complessivo** ha prodotto un valore di **38/100**, indicando un livello moderato di rischio per l'intero sistema.

Tuttavia, il rischio specifico associato alle *CWE* e ai *threats* identificati varia significativamente, con alcune categorie come **CWE-400** e il *threat nefarious activity/abuse* che raggiungono punteggi di rischio relativamente alti (**40/100**).

I risultati ottenuti dimostrano che la metodologia proposta consente di determinare un quadro completo e dettagliato delle vulnerabilità e dei rischi associati ad *architetture a microservizi* complesse in maniera continua, rispettando, quindi, la definizione di *software assurance* [1.4] per quanto riguarda la verifica continua delle proprietà di sicurezza definite a monte, attraverso anche i risultati dei rischi calcolati.

Ciò supera le limitazioni delle tecniche tradizionali di analisi, consistenti in esecuzioni di uno o più tool di sicurezza su singoli target, le quali non forniscono un risultato aggregato significativo, il quale è determinante per effettuare le giuste scelte di prioritizzazione delle *remediation delle vulnerabilità* [A.0.42], le quali hanno un impatto altamente significativo nella prevenzione agli attacchi.

Alla luce dei risultati ottenuti, la metodologia può essere sicuramente estesa con alcuni sviluppi futuri che possono riguardare:

- Integrazione della metodologia proposta nelle pipeline di sviluppo esistenti utilizzando le *API di MoonCloud*;
- Estensione della metodologia con tecniche di *analisi dinamica* (ad esempio *fuzzing* [A.0.58] e *penetration testing* [1.4.2]) in grado di individuare tipologie di vulnerabilità non individuabili attraverso l'*analisi statica*;
- Ricerca ed integrazione di altri tool di analisi di sicurezza progettati per altre tipologie di target (ad esempio *dispositivi mobile*, *smart contract* [A.0.96], infrastrutture di rete e sistemi di controllo industriale);
- Potenziamento del *mapping* tra le vulnerabilità e i *threat* attraverso altre metriche di calcolo del rischio e integrazione di altri metodi di visualizzazione dei dati.

# Appendice A

# Definizioni

**Definizione A.0.1.** I **microservizi** sono un'architettura di sviluppo software in cui un'applicazione è suddivisa in una serie di servizi piccoli e indipendenti che comunicano tra loro tramite *API*. Ogni microservizio è progettato per eseguire una singola funzione e può essere sviluppato, distribuito e scalato in modo indipendente dagli altri servizi, migliorando la modularità, facilitando la manutenzione, l'aggiornamento e la scalabilità delle applicazioni complesse [Newman, 2015].

**Definizione A.0.2.** Il **provisioning di risorse** è il processo di configurazione e gestione automatizzata delle risorse informatiche, come server, storage e reti [et al., 2016].

**Definizione A.0.3.** Il **cloud** si riferisce a un *modello di computing* che permette l'accesso a un insieme condiviso di risorse informatiche configurabili (come reti, server, storage, applicazioni e servizi) che possono essere avviate rapidamente al fine di offrire scalabilità, elasticità e flessibilità, consentendo alle organizzazioni di adattare rapidamente le risorse alle proprie esigenze [Mell and Grance, 2011].

**Definizione A.0.4.** Le **infrastrutture on-premises** sono risorse IT, come server, storage e reti, che sono fisicamente situate all'interno dei locali di un'organizzazione. Queste infrastrutture sono gestite e mantenute dall'organizzazione stessa, offrendo controllo diretto e completo sulle risorse hardware e software. Le *soluzioni on-premises* sono spesso preferite per motivi di sicurezza, conformità e prestazioni, ma comportano costi iniziali elevati e necessitano di personale qualificato per la gestione e la manutenzione [Miller et al., 2016].

**Definizione A.0.5.** Un **container** è un'unità standard di software che raggruppa il codice e tutte le sue dipendenze, permettendo che un'applicazione venga eseguita in modo rapido, isolata e affidabile da un ambiente di elaborazione all'altro. I container virtualizzano il sistema operativo e sono più leggeri rispetto alle macchine virtuali [Merkel, 2014].

**Definizione A.0.6.** Gli **orchestratori di container** sono strumenti e piattaforme software che automatizzano il *deployment*, la gestione, il ridimensionamento e il *networking* di *applicazioni containerizzate*. Essi gestiscono cluster di container, distribuendo carichi di lavoro e garantendo alta disponibilità e resilienza [Bernstein, 2014].

**Definizione A.0.7. Kubernetes**[1] è una piattaforma *open source* per l'automazione della gestione, del *deployment* e dello *scaling* di *applicazioni containerizzate*. *Kubernetes* fornisce un sistema per orchestrare container su cluster di macchine, facilitando il bilanciamento del carico, la gestione delle risorse e la resilienza delle applicazioni [Kubernetes Authors, 2021].

**Definizione A.0.8. Docker**[2] è una piattaforma *open source* che automatizza la distribuzione di applicazioni all'interno di container software. *Docker* consente di separare le applicazioni dall'infrastruttura, facilitando il *delivery continuo* e migliorando la portabilità delle applicazioni tra diversi ambienti [Merkel, 2014].

**Definizione A.0.9.** Un **Dockerfile** è un file di testo contenente una serie di istruzioni che definiscono i passaggi per creare un'immagine *Docker*, come configurazioni dell'ambiente, fasi di installazione delle dipendenze e copia di file dalla macchina host [Turnbull, 2014].

**Definizione A.0.10. Docker compose**[3] è uno strumento per la definizione e la gestione di *applicazioni Docker multi-container*. Utilizzando un file *YAML*[4], *Docker Compose* consente di configurare i servizi, le reti e i volumi necessari per eseguire un'applicazione completa in ambienti di sviluppo, test e produzione [Turnbull, 2014].

**Definizione A.0.11.** Il **load balancing** è una tecnica di distribuzione uniforme del traffico di rete o delle richieste di elaborazione tra diversi server, risorse o nodi in un ambiente informatico. L'obiettivo del load balancing è ottimizzare l'utilizzo delle risorse, massimizzare il throughput, minimizzare la latenza e garantire l'affidabilità e la disponibilità dei servizi [Awduche et al., 2002].

**Definizione A.0.12. Agile** è un insieme di metodologie di sviluppo software che promuovono un approccio iterativo e incrementale al fine di favorire la collaborazione tra team cross-funzionali e la flessibilità nel rispondere ai cambiamenti attraverso l'interazione costante con il cliente [Beck et al., 2001].

[1] https://kubernetes.io
[2] https://docker.com
[3] https://docs.docker.com/compose
[4] https://yaml.org

**Definizione A.0.13.** Il **riuso**, in informatica, si riferisce alla pratica di utilizzare componenti software esistenti, come librerie, moduli, codice sorgente o architetture, in nuovi contesti o applicazioni. L'obiettivo del riuso è ridurre il tempo di sviluppo, i costi e migliorare la qualità e la manutenzione del software, evitando la duplicazione di sforzi e sfruttando soluzioni già collaudate [Krueger, 1992].

**Definizione A.0.14.** La **scalabilità** è la capacità di un sistema, rete o processo di gestire una quantità crescente di lavoro o la sua potenziale capacità di essere ampliato per accogliere tale crescita. Un sistema è considerato scalabile se può aumentare le proprie prestazioni proporzionalmente all'incremento delle risorse aggiunte, come CPU, memoria o nodi di rete, senza compromettere la funzionalità o l'efficienza [Bondi, 2000].

**Definizione A.0.15.** La **resilienza del sistema** è la capacità di un sistema di continuare a funzionare correttamente e di riportarsi nello stato di funzionamento normale a seguito di eventuali guasti, attacchi o altre perturbazioni. Un *sistema resiliente* è progettato per mantenere un livello accettabile di servizio anche in condizioni avverse, minimizzando l'impatto delle interruzioni e ripristinando le funzionalità normali nel più breve tempo possibile [Laprie, 2008].

**Definizione A.0.16.** La **fault tolerance** è la capacità di un sistema di continuare a funzionare correttamente anche in presenza di guasti o malfunzionamenti di alcuni dei suoi componenti. In particolare, attraverso la ridondanza, la rilevazione dei guasti e l'adozione di strategie di ripristino, permettono al sistema di mantenere la continuità operativa e ridurre al minimo l'impatto dei malfunzionamenti. [Johnson, 1989].

**Definizione A.0.17.** I **costi di manutenzione** sono le spese associate al processo di aggiornamento, miglioramento e riparazione del software dopo il suo rilascio. Ad esempio, questi costi riguardano la correzione di bug, l'adattamento a nuovi ambienti hardware o software, l'aggiunta di nuove funzionalità e il miglioramento delle prestazioni e della sicurezza [Somerville, 2011].

**Definizione A.0.18.** La **fase di dismissione** è il processo di ritiro dal servizio di sistemi, applicazioni o componenti IT, garantendo che tutte le informazioni e i dati sensibili vengano trattati in modo sicuro e che le risorse vengano recuperate o eliminate correttamente per prevenire accessi non autorizzati o perdita di dati [ISO, 2013].

**Definizione A.0.19.** Il termine **open source** si riferisce a un modello di sviluppo del software in cui il codice sorgente è reso disponibile al pubblico sotto una licenza che permette a chiunque di vedere, modificare e distribuire il codice. Questo approccio promuove la collaborazione e l'innovazione, consentendo agli sviluppatori di migliorare continuamente il software [Raymond, 2001].

**Definizione A.0.20.** Il **Software as a Service** (abbreviato SaaS) è un modello di distribuzione del software in cui le applicazioni sono ospitate da un fornitore di servizi e rese disponibili agli utenti attraverso Internet. Gli utenti accedono al software tramite un browser web, senza necessità di installazioni locali o gestione dell'infrastruttura bensì solo attraverso un abbonamento, riducendo i costi di gestione e di aggiornamento del software [Choudhary, 2007].

**Definizione A.0.21.** Un **repository git** è un archivio che contiene tutti i file e la cronologia delle revisioni di un progetto gestito con il sistema di controllo delle versioni *git*[5]. I *repositoryg git* possono essere *locali* o *remoti* e consentono agli sviluppatori di collaborare, tenere traccia delle modifiche al codice e gestire versioni multiple del software [Chacon and Straub, 2014].

**Definizione A.0.22.** Gli **stack tecnologici** sono combinazioni di tecnologie, strumenti e *framework* utilizzati insieme per sviluppare e gestire un'applicazione software [Fowler, 2002].

**Definizione A.0.23.** Il **backend** è la parte di un'applicazione o di un sistema informatico che gestisce la logica di business, l'elaborazione dei dati e l'interazione con il database. Esso opera "dietro le quinte" per garantire che le operazioni richieste dall'utente, attraverso l'interfaccia frontend, vengano eseguite correttamente [Bass et al., 2003].

**Definizione A.0.24.** Il **frontend** è la parte di un'applicazione o di un sistema informatico con cui gli utenti interagiscono direttamente ed è responsabile della visualizzazione dei dati provenienti dal *backend* e dell'elaborazione degli input degli utenti. Esso include l'*user interface* (UI) e l'*user experience* (UX) al fine di presentare le informazioni in modo accessibile e intuitivo [Bass et al., 2003].

**Definizione A.0.25. AngularJS**[6] è un *framework open-source* per lo sviluppo di applicazioni web mantenuto da *Google* e da una comunità di sviluppatori indipendenti. Basato su *JavaScript*, *AngularJS* estende il linguaggio *HTML* con nuove direttive e offre strumenti per il data binding bidirezionale, l'iniezione di dipendenze e la gestione di applicazioni a pagina singola [AngularJS Authors, 2021].

**Definizione A.0.26.** Il **JavaScript Object Notation** (abbreviato JSON[7]) è un formato di interscambio di dati leggero, facile da leggere e scrivere sia per gli esseri umani che per le macchine. *JSON* è basato su un sottoinsieme del linguaggio di programmazione *JavaScript* e viene utilizzato principalmente per trasmettere dati strutturati tra un server e un client web, come parte della comunicazione tra applicazioni web [Crockford, 2006].

[5]https://git-scm.com
[6]https://angularjs.org
[7]https://json.org

**Definizione A.0.27. JSON Web Token** (abbreviato JWT[8]) è uno standard *open source* (RFC 7519) che definisce un metodo compatto e autonomo per la trasmissione sicura di informazioni tra le parti. Queste informazioni possono essere verificate perché sono firmate digitalmente, infatti i *token JWT* sono comunemente usati per l'autenticazione e l'autorizzazione in applicazioni web, permettendo la trasmissione di richieste di autenticazione tra client e server in modo sicuro [Jones et al., 2015].

**Definizione A.0.28.** Un'**Application Programming Interface** (abbreviato API) è un insieme di regole e definizioni che permette a diverse applicazioni software di comunicare tra loro. Le *API* definiscono i metodi e i dati che le applicazioni possono utilizzare per richiedere servizi, scambiare informazioni e interagire con altre applicazioni, sistemi operativi o librerie [Fielding, 2000].

**Definizione A.0.29.** Una **minaccia** (o **threat**) è una potenziale causa di un incidente indesiderato che può danneggiare un sistema o un'organizzazione [ISO, 2013].

**Definizione A.0.30.** Una **vulnerabilità** è una debolezza di un asset o di un controllo che può essere sfruttata da una o più minacce [ISO, 2013].

**Definizione A.0.31.** La **gap analysis** è una tecnica utilizzata per confrontare le prestazioni attuali di un'organizzazione, un progetto o un processo con le prestazioni desiderate o attese. Questo processo identifica le differenze (o *gap*) tra lo stato attuale e quello desiderato, consentendo di sviluppare strategie per colmare tali lacune [ISO, 2018c].

**Definizione A.0.32.** La **threat analysis** è il processo di valutazione e comprensione delle potenziali minacce che potrebbero compromettere la sicurezza di un sistema. Questo processo include l'identificazione delle minacce, la valutazione delle loro capacità e potenziali impatti [ISO, 2013].

**Definizione A.0.33.** L'**approccio STRIDE** è un modello di minaccia utilizzato per identificare potenziali minacce alla sicurezza di un sistema informatico. *STRIDE* è un acronimo che rappresenta sei categorie di minacce: **spoofing** (falsificazione dell'identità), **tampering** (manomissione dei dati), **repudiation** (ripudio), **information disclosure** (divulgazione non autorizzata di informazioni), **Denial of Service** (negazione del servizio) e **elevation of privilege** (elevazione dei privilegi) [Hernan et al., 2006].

**Definizione A.0.34.** L'**analisi SWOT** è uno strumento strategico utilizzato per identificare e analizzare i punti di forza (**strengths**), le debolezze (**weaknesses**), le opportunità (**opportunities**) e le minacce (**threats**) di un'organizzazione o di

[8] https://jwt.io

un progetto. Questo approccio consente di valutare sia gli aspetti interni (forze e debolezze) che quelli esterni (opportunità e minacce), fornendo una panoramica completa della situazione attuale e facilitando la pianificazione strategica [Gürel and Tat, 2017].

**Definizione A.0.35.** I **requisiti di sicurezza** sono specifiche dettagliate che definiscono le misure e i controlli necessari per proteggere un sistema informatico, una rete o un'applicazione da minacce e vulnerabilità. Questi requisiti includono aspetti come la riservatezza, l'integrità, la disponibilità, l'autenticazione, l'autorizzazione e la conformità alle normative. [Shostack, 2014].

**Definizione A.0.36.** Uno **stakeholder** è una persona o un'organizzazione che può influenzare, essere influenzata, o percepirsi come influenzata da una decisione, attività o risultati di un progetto o processo [ISO, 2018b].

**Definizione A.0.37.** Il **rischio** è l'effetto dell'incertezza sugli obiettivi, spesso quantificato come una combinazione della probabilità di un evento e delle sue conseguenze. Nel contesto della sicurezza delle informazioni il rischio è legato alla possibilità che una minaccia sfrutti una vulnerabilità causando un impatto negativo sull'organizzazione [ISO, 2013].

**Definizione A.0.38.** L'**analisi del rischio** è il processo di identificazione, valutazione e prioritizzazione dei rischi, seguito dall'applicazione coordinata di risorse per minimizzare, monitorare e controllare la probabilità e/o l'impatto degli eventi avversi [ISO, 2013].

**Definizione A.0.39.** Le **matrici di rischio** sono strumenti utilizzati per valutare e visualizzare i rischi associati a un progetto, un'attività o un'organizzazione, aiutando a prioritizzarli al fine di facilitare la decisione su quali rischi devono essere gestiti per primi e quali possono essere monitorati e gestiti in un secondo momento. In particolare, una matrice di rischio rappresenta i rischi su una griglia bidimensionale dove un asse rappresenta la probabilità di occorrenza del rischio e l'altro asse rappresenta l'impatto del rischio [ISO, 2018b].

**Definizione A.0.40.** L'**audit** è un processo sistematico, indipendente e documentato per ottenere evidenze e valutarle obiettivamente al fine di determinare in che misura i criteri di *audit* siano soddisfatti. Gli *audit* possono essere interni o esterni e sono utilizzati per garantire che le pratiche, le procedure e i controlli di un'organizzazione siano conformi agli standard, alle normative e ai requisiti stabiliti [ISO, 2018a].

**Definizione A.0.41.** I **controlli di sicurezza** sono misure, politiche, procedure e tecniche implementate per ridurre i rischi per la sicurezza delle informazioni.

Questi controlli mirano a proteggere la riservatezza, l'integrità e la disponibilità delle informazioni, prevenendo accessi non autorizzati, uso improprio, divulgazione, distruzione o modifica delle stesse [ISO, 2013].

**Definizione A.0.42.** La **remediation delle vulnerabilità** è il processo di identificazione delle vulnerabilità di sicurezza nei sistemi informatici, valutazione del rischio associato e implementazione di misure correttive per eliminare o mitigare tali vulnerabilità [ISO, 2013].

**Definizione A.0.43.** Il **Common Weakness Enumeration** (abbreviato CWE[9]) è un dizionario di debolezze di sicurezza del software e delle vulnerabilità sviluppato per aiutare a identificare, mitigare e prevenire i difetti nei software. Il *CWE* fornisce una tassonomia standardizzata delle debolezze del software, permettendo ai professionisti della sicurezza, agli sviluppatori e agli auditor di comunicare in modo efficace sui problemi di sicurezza e di migliorare la qualità del software attraverso l'uso di best practice e metodologie di sviluppo sicure [The MITRE Corporation, 2021b].

**Definizione A.0.44.** Il **Common Vulnerabilities and Exposures** (abbreviato CVE[10]) è un elenco di informazioni di riferimento standardizzate riguardanti le vulnerabilità di sicurezza note nei sistemi software e hardware. Esso è gestito dal *MITRE Corporation* e *National Institute of Standards and Technology* (NIST). In particolare, ogni *CVE* contiene un identificatore univoco, una descrizione della vulnerabilità e riferimenti a informazioni di mitigazione e risoluzione [The MITRE Corporation, 2021a].

**Definizione A.0.45.** Il **Common Vulnerability Scoring System** (abbreviato CVSS[11]) è uno standard aperto per l'assegnazione di punteggi alla gravità delle vulnerabilità di sicurezza informatica. *CVSS* quantifica la gravità delle vulnerabilità basandosi su metriche come l'accessibilità, la complessità dell'attacco e l'impatto sulla riservatezza, integrità e disponibilità [Mell et al., 2006].

*CVSS* esiste in due versioni: V2 e V3. La versione V2 utilizza metriche base per determinare la gravità delle vulnerabilità, mentre la versione V3 introduce una maggiore granularità delle metriche, metriche temporali e ambientali e una scala di punteggio più ampia, rendendo la valutazione più accurata e rappresentativa delle vulnerabilità moderne [Mell et al., 2007].

**Definizione A.0.46.** La **severity** (o gravità) è una misura della serietà o dell'impatto potenziale di una vulnerabilità di sicurezza. Essa valuta il livello di danno che una vulnerabilità potrebbe causare se sfruttata, tenendo conto di fattori

[9] https://cwe.mitre.org
[10] https://cve.mitre.org
[11] https://nvd.nist.gov/vuln-metrics/cvss

come l'integrità, la riservatezza e la disponibilità dei sistemi e dei dati coinvolti [FIRST, 2019].

**Definizione A.0.47.** Lo **score** (o punteggio) è un valore numerico assegnato a una vulnerabilità per rappresentarne il livello di rischio complessivo. Questo punteggio è spesso calcolato utilizzando metriche standardizzate, come il *Common Vulnerability Scoring System* (CVSS), che prendono in considerazione fattori come l'accessibilità, la complessità dell'attacco, l'impatto e la severità [FIRST, 2019].

**Definizione A.0.48.** I **dati sensibili** sono informazioni quali informazioni personali, dati finanziari, segreti commerciali e altre informazioni riservate che, se divulgate, possono causare danni o rischi significativi [ISO, 2013].

**Definizione A.0.49.** La **compliance** è il rispetto di leggi, regolamenti, standard e linee guida pertinenti applicabili a un'organizzazione. Nel contesto della sicurezza delle informazioni, la *compliance* implica l'adozione di misure e controlli necessari per garantire che le pratiche aziendali soddisfino i requisiti legali e normativi [ISO, 2013].

**Definizione A.0.50. H2020-CONCORDIA**[12] è un progetto di ricerca finanziato dal programma *Horizon 2020*[13] dell'*Unione Europea*[14] (UE), finalizzato a sviluppare e migliorare le capacità di *cybersecurity* in Europa. Il progetto *CONCORDIA* riunisce esperti accademici, industriali e governativi per collaborare su soluzioni innovative e per promuovere l'educazione e la consapevolezza sulla sicurezza informatica [CONCORDIA, 2021].

**Definizione A.0.51.** L'**Agenzia dell'Unione Europea per la Cybersecurity** (abbreviato ENISA[15]) è un'agenzia dell'UE che ha il compito di migliorare la sicurezza delle reti e delle informazioni nell'*Unione Europea*, fornendo competenze, supporto e risorse per aiutare gli Stati membri e le istituzioni dell'UE a prevenire, rilevare e rispondere agli incidenti di sicurezza informatica [ENISA, 2021].

**Definizione A.0.52.** Il **National Institute of Standards and Technology** (abbreviato NIST[16]) è un'agenzia del *Dipartimento del Commercio* degli Stati Uniti, che fornisce standard, linee guida e best practice per migliorare la sicurezza delle informazioni e delle tecnologie informatiche [NIST, 2021].

[12] https://concordia-h2020.eu
[13] https://horizon2020.apre.it
[14] https://european-union.europa.eu
[15] https://enisa.europa.eu
[16] https://nist.gov

**Definizione A.0.53.** Il **General Data Protection Regulation** (abbreviato GDPR[17]) è un regolamento dell'UE che disciplina la protezione dei dati personali e la privacy dei cittadini dell'UE. Entrato in vigore il 25 maggio 2018, esso stabilisce rigide norme su come le organizzazioni devono raccogliere, trattare, conservare e proteggere i dati personali, garantendo ai cittadini maggiori diritti di controllo sui propri dati [Voigt and Von dem Bussche, 2017].

**Definizione A.0.54.** Il **Payment Card Industry Data Security Standard** (abbreviato PCI-DSS[18]) è uno standard di sicurezza delle informazioni istituito per proteggere i dati delle carte di pagamento sviluppato dal *Payment Card Industry Security Standards Council*. Il *PCI-DSS* stabilisce un insieme di requisiti tecnici e operativi per garantire che tutte le organizzazioni che gestiscono, elaborano o trasmettono informazioni sulle carte di credito mantengano un ambiente sicuro [Council, 2022].

**Definizione A.0.55.** Il **Capability Assessment for AI** (abbreviato CapAI[19]) è un framework progettato per valutare le capacità e la maturità delle soluzioni di intelligenza artificiale. Questo framework fornisce linee guida e criteri per analizzare vari aspetti delle tecnologie di intelligenza artificiale, come la loro efficacia, sicurezza, eticità e conformità a normative e standard, al fine di implementare e gestire soluzioni di intelligenza artificiale in modo responsabile e sostenibile [European Commission, 2020b].

**Definizione A.0.56.** L'**Assessment List for Trustworthy Artificial Intelligence** (abbreviato ALTAI[20]) è un insieme di linee guida e strumenti sviluppati dalla *Commissione Europea*[21] per promuovere l'uso responsabile e affidabile dell'intelligenza artificiale. *ALTAI* fornisce un elenco di criteri per valutare se una soluzione di intelligenza artificiale rispetta i principi etici fondamentali quali trasparenza, responsabilità, non discriminazione e sicurezza [European Commission, 2020a].

**Definizione A.0.57.** Il **reverse engineering** è il processo di analisi di un sistema, un componente o un dispositivo per comprenderne il funzionamento interno, la struttura e il design originario. Questo processo implica l'esame dettagliato del prodotto finale per ricostruire il suo progetto originale, il codice sorgente o le specifiche tecniche [Chikofsky and Cross, 1990].

[17] https://gdpr.eu
[18] https://pcisecuritystandards.org
[19] https://papers.ssrn.com/sol3/papers.cfm?abstract_id=4064091
[20] https://digital-strategy.ec.europa.eu/it/policies/european-approach-artificial-intelligence
[21] https://commission.europa.eu

**Definizione A.0.58.** Il **fuzzing** è una tecnica di analisi dinamica utilizzata per individuare vulnerabilità e bug nei software che consiste nell'inviare dati di input casuali oppure non validi a un programma, osservandone il comportamento per rilevare eventuali errori, crash, *memory leaks* o altri problemi di sicurezza [Sutton et al., 2007].

**Definizione A.0.59.** Un **tool di sicurezza** è un software o uno strumento utilizzato per proteggere sistemi informatici, reti e dati dalle minacce di sicurezza. Questi strumenti possono svolgere una varietà di funzioni, tra cui la rilevazione di *malware*, la gestione delle vulnerabilità, la cifratura dei dati, il monitoraggio del traffico di rete, e l'esecuzione di *penetration test* [1.4.2] [Stallings, 2015].

**Definizione A.0.60.** Le **pratiche di codifica sicura** (oppure secure coding practices) sono un insieme di linee guida e metodologie progettate per aiutare gli sviluppatori a scrivere codice privo di vulnerabilità che potrebbero essere sfruttate da attaccanti per compromettere la sicurezza del software. Queste pratiche includono l'uso di tecniche di validazione degli input, la gestione sicura delle risorse, la protezione dei dati sensibili e l'adozione di controlli di accesso appropriati [Open Web Application Security Project, 2010].

**Definizione A.0.61.** L'**hardening** è il processo di miglioramento della sicurezza di un sistema informatico per minimizzare la superficie di attacco mediante la riduzione delle sue vulnerabilità. Questo processo include l'applicazione di patch e aggiornamenti, la configurazione sicura di software e hardware, la rimozione di servizi e applicazioni non necessari e l'implementazione di controlli di accesso rigorosi [Lynch, 2006].

**Definizione A.0.62.** La **revisione del codice peer-to-peer** è il processo mediante il quale sviluppatori esaminano reciprocamente il codice sorgente per individuare errori, migliorare la qualità del software e garantire la conformità agli standard di codifica [McConnell, 2004].

**Definizione A.0.63.** La **verifica e validazione del software** sono processi distinti ma correlati volti a garantire che il software soddisfi i requisiti specificati e funzioni correttamente. La **verifica** si concentra sull'assicurare che il software sia stato costruito correttamente, mentre la **validazione** si concentra sull'assicurare che il software costruito sia quello giusto per gli scopi previsti [IEEE, 2016].

**Definizione A.0.64.** Un **artifact** (o artefatto), nel contesto dello sviluppo software, è un qualsiasi prodotto tangibile che viene creato durante il ciclo di vita del software. Ad esempio, tra gli *artifact* rientrano il codice sorgente, la documentazione, i modelli di design, le specifiche dei requisiti e gli eseguibili [Pressman, 2005].

**Definizione A.0.65.** Un **payload malevolo** è la parte di un attacco informatico che esegue fisicamente le azioni dannose su un sistema bersaglio, come la cancellazione di dati, il furto di informazioni sensibili, l'installazione di malware, o la compromissione della funzionalità del sistema [Skoudis and Zeltser, 2004].

**Definizione A.0.66.** Lo **spazio degli stati** è una rappresentazione astratta di tutte le possibili configurazioni di un sistema in cui ogni punto nello spazio degli stati corrisponde a un possibile stato del sistema, definito dai valori delle variabili di stato che descrivono il sistema stesso [Bellman, 1957].

**Definizione A.0.67.** I **sistemi concorrenti** sono sistemi in cui diversi processi o thread vengono eseguiti simultaneamente, potenzialmente interagendo tra loro, attraverso la gestione della sincronizzazione e della comunicazione tra processi, per migliorare l'efficienza e le prestazioni mediante l'esecuzione parallela delle operazioni [Tanenbaum and Bos, 2007].

**Definizione A.0.68. Undefined behaviour** è un termine utilizzato in informatica per descrivere il comportamento di un programma che non è specificato dal linguaggio di programmazione in determinate condizioni. Quando un programma presenta un *undefined behaviour*, il risultato può variare a seconda del compilatore, del sistema operativo o dell'architettura hardware, rendendo il programma potenzialmente instabile o insicuro [ISO, 2018d].

**Definizione A.0.69.** Un **side channel attack** è un tipo di attacco basato su informazioni supplementari che possono essere raccolte a causa del modo in cui un protocollo o un algoritmo informatico è implementato oppure a causa di difetti nel design del protocollo o dell'algoritmo stesso [Kocher, 1996].

**Definizione A.0.70.** Un **buffer overflow** è un tipo di *side channel attack* [A.0.69] che si verifica quando un programma scrive più dati di quelli che un buffer può contenere, sovrascrivendo dati adiacenti in memoria. Ciò può corrompere l'esecuzione di un programma e consentire l'esecuzione di codice dannoso [Cowan et al., 2000].

**Definizione A.0.71.** Un **heap overflow** è un tipo specifico di *buffer overflow* che si verifica nell'*heap*, l'area di memoria dinamica utilizzata per l'allocazione dinamica di memoria a tempo di esecuzione, causando comportamenti imprevisti, crash di sistema o esecuzione di codice arbitrario [Seacord, 2005].

**Definizione A.0.72.** Le **race conditions** sono situazioni che si verificano quando il comportamento di un sistema informatico dipende dall'ordine o dalla tempistica delle operazioni in esecuzione, causando potenzialmente comportamenti imprevisti o errori. Questo avviene spesso in contesti di programmazione concorrente, quando più thread o processi accedono e manipolano risorse condivise senza un'adeguata sincronizzazione [Bishop, 2005].

**Definizione A.0.73.** La **SQL injection** è una tecnica di attacco informatico che consente a un attaccante di interferire con le *query SQL* che un'applicazione esegue sul suo database. Questa tecnica di attacco sfrutta l'inserimento di codice *SQL* malevolo nei campi di input, permettendo all'attaccante di visualizzare, modificare o eliminare dati non autorizzati [Open Web Application Security Project, 2017b].

**Definizione A.0.74.** Il **Cross-Site Scripting** (abbreviato XSS) è una vulnerabilità di sicurezza che consente a un attaccante di iniettare script maligni nelle pagine web visualizzate da altri utenti. Questi script possono essere utilizzati per rubare dati sensibili, come cookie di sessione, sessioni di autenticazione, o per eseguire altre azioni dannose a nome della vittima [Open Web Application Security Project, 2017a].

**Definizione A.0.75.** La **privilege escalation** è una tecnica di attacco informatico in cui un utente ottiene livelli di accesso superiori rispetto a quelli assegnati originariamente. Questo può avvenire sfruttando vulnerabilità nel software, errori di configurazione o altre debolezze di sicurezza [Hoglund and McGraw, 2004].

**Definizione A.0.76.** Gli **attacchi Denial of Service** (abbreviato DoS) e **Distributed Denial of Service** (abbreviato DDoS) mirano a rendere inaccessibili servizi, risorse o reti a utenti legittimi sovraccaricando il sistema bersaglio con un'eccessiva quantità di richieste da una singola fonte. Gli *attacchi DDoS* utilizzano molteplici fonti distribuite per amplificare l'impatto, rendendo più difficile mitigare l'attacco [Mirkovic and Reiher, 2004].

**Definizione A.0.77.** L'**eavesdropping** è un tipo di attacco di sicurezza in cui un malintenzionato intercetta e ascolta comunicazioni private tra due parti senza il loro consenso, con l'obiettivo di raccogliere informazioni sensibili come credenziali di accesso, dati personali e dettagli finanziari [Stallings, 2015].

**Definizione A.0.78.** L'**interception** è un attacco in cui un malintenzionato intercetta, e potenzialmente altera, i dati in transito tra due parti. A differenza dell'*eavesdropping*, che si limita all'ascolto passivo, l'*interception* può includere la modifica attiva dei messaggi per compromettere l'integrità e la riservatezza delle comunicazioni [Stallings, 2015].

**Definizione A.0.79.** L'**hijacking** è un attacco informatico in cui un malintenzionato prende il controllo di una sessione di comunicazione attiva tra due parti senza il loro consenso. Ad esempio, nel *session hijacking*, l'attaccante ruba una sessione autenticata, mentre nel *clickjacking*, l'utente è ingannato a cliccare su elementi nascosti che eseguono azioni non desiderate [Stallings, 2015].

**Definizione A.0.80.** L'**Internet of Things** (abbreviato IoT) è un sistema di dispositivi fisici interconnessi, veicoli, elettrodomestici e altri oggetti che utilizzano sensori, software e altre tecnologie per connettersi e scambiare dati con altri dispositivi e sistemi attraverso Internet [Atzori et al., 2010].

**Definizione A.0.81.** L'**Edge Cloud Continuum** è un paradigma informatico che integra risorse di elaborazione *cloud* centralizzate con risorse di elaborazione distribuite ai margini della rete (edge). Questo continuum permette di sfruttare la potenza del *cloud computing* per operazioni intensive e di bassa latenza, mentre le risorse edge gestiscono l'elaborazione locale, riducendo la latenza e migliorando la reattività delle applicazioni. Questo approccio combina i vantaggi del *cloud* e dell'*edge computing*, offrendo flessibilità, scalabilità e prestazioni ottimizzate [Satyanarayanan, 2017].

**Definizione A.0.82. File Allocation Table 32** (abbreviato FAT32) è un *file system* sviluppato da *Microsoft* nel 1996 come evoluzione dei precedenti *file system* *FAT12* e *FAT16*.

*FAT32* utilizza una tabella di allocazione dei file a 32 bit, permettendo di gestire partizioni di disco rigido di dimensioni fino a 2 terabyte e file individuali fino a 4 gigabyte [Microsoft, 1996].

**Definizione A.0.83.** Il **Device Tree Blob** (abbreviato DTB) è una rappresentazione binaria della struttura ad albero del dispositivo (*Device Tree*), che descrive l'hardware di un sistema al *kernel* del sistema operativo. Il *Device Tree Blob* contiene informazioni sulle periferiche hardware, come i bus, i dispositivi e le loro proprietà, permettendo al *kernel* di configurare e gestire correttamente l'hardware durante l'avvio [Brown and Hallinan, 2015].

**Definizione A.0.84. Reduced Instruction Set Computer** (RISC) è un'architettura di microprocessori che utilizza un insieme ridotto e altamente ottimizzato di istruzioni per migliorare la velocità e l'efficienza del processore mediante l'esecuzione di un numero limitato di istruzioni semplici, piuttosto che un'ampia gamma di istruzioni complesse [Hennessy and Patterson, 2011].

**Definizione A.0.85.** L'**architettura ARM** (Advanced RISC Machine) è una famiglia di architetture per microprocessori basata su un design di tipo *RISC* [A.0.84], sviluppata dalla *ARM Holdings*[22]. Caratterizzata da un'elevata efficienza energetica e prestazioni ottimali, l'*architettura ARM* è ampiamente utilizzata in dispositivi mobili, *dispositivi embedded* e sistemi a bassa potenza [Furber, 2000].

**Definizione A.0.86.** Il **Raspberry Pi**[23] è un microcomputer a basso costo e di dimensioni ridotte sviluppato dalla *Raspberry Pi Foundation* nel Regno Unito, composto da componenti minimani, quali il processore, la memoria, porte *USB* e *HDMI* [Upton and Halfacree, 2014].

[22] https://arm.com
[23] https://raspberrypi.com

**Definizione A.0.87.** Il **boot loader** è un software che viene eseguito all'avvio di un computer o di un *dispositivo embedded* con lo scopo di inizializzare l'hardware del sistema, caricare il sistema operativo in memoria, preparare l'*ambiente di runtime* e trasferire il controllo al *kernel* del sistema operativo [Levine, 2009].

**Definizione A.0.88.** La **mutua autenticazione TLS** (Transport Layer Security) è un processo di sicurezza in cui entrambi i lati di una comunicazione *TLS*, cioè il client e il server, si autenticano reciprocamente utilizzando certificati digitali. Durante la *fase di handshake* di una connessione *TLS*, il server fornisce il proprio certificato al client e, a sua volta, richiede al client di presentare il proprio certificato. Questo processo garantisce che entrambe le parti siano autentiche e autorizzate a stabilire la connessione, migliorando così la sicurezza delle comunicazioni [Dierks and Rescorla, 2008].

**Definizione A.0.89.** I **certificati X.509** sono standard di certificati digitali definiti dalla raccomandazione *ITU-T X.509*. Essi sono utilizzati per gestire chiavi pubbliche e per certificare l'identità di una o più entità in una rete. Un *certificato X.509* contiene informazioni come la chiave pubblica del soggetto, il nome del soggetto, il nome dell'autorità di certificazione che ha emesso il certificato, una data di scadenza e altre informazioni necessarie per la crittografia e l'autenticazione [Cooper et al., 2008].

**Definizione A.0.90. Open Authorization** (abbreviato OAuth[24]) è un protocollo *open source* che consente a un'applicazione di ottenere accesso limitato alle risorse di un utente su un altro servizio senza esporre le credenziali dell'utente [Hardt, 2012].

**Definizione A.0.91.** Il **Secure Shell** (abbreviato SSH[25]) è un protocollo di rete crittografico utilizzato per operare servizi di rete in modo sicuro su una rete non sicura. *SSH* fornisce un canale sicuro su una rete insicura in un'*architettura client-server*, permettendo di eseguire comandi a distanza, trasferire file e gestire in modo sicuro le chiavi di autenticazione [Ylonen, 1996].

**Definizione A.0.92.** L'**Internet Protocol version 6** (abbreviato IPv6) è la versione più recente dell'*Internet Protocol* (IP), progettata per sostituire la versione *IPv4* a causa della continua crescita di dispositivi connessi a Internet con conseguente diminuzione dello spazio degli indirizzi disponibili. *IPv6* offre uno spazio di indirizzamento molto più ampio, miglioramenti nella gestione della sicurezza e supporto per l'autoconfigurazione degli indirizzi [Hinden and Deering, 2006].

[24]https://oauth.net
[25]https://ssh.org

**Definizione A.0.93.** Un **Intrusion Detection System** (abbreviato IDS) è un dispositivo o software progettato per monitorare reti o sistemi informatici al fine di rilevare attività sospette o malintenzionate. Gli *IDS* analizzano il traffico di rete, i log di sistema e altre risorse per identificare potenziali attacchi o violazioni di sicurezza, avvisando gli amministratori di sistema in caso di rilevamento di comportamenti anomali [Scarfone and Mell, 2007].

**Definizione A.0.94.** Un **Intrusion Prevention System** (abbreviato IPS) è un dispositivo di sicurezza di rete che monitora il traffico di rete per rilevare e prevenire attività sospette o dannose, analizzando i pacchetti di dati ed, eventualmente, bloccando il traffico che corrisponde a firme di attacchi noti [Scarfone and Mell, 2007].

**Definizione A.0.95.** La **blockchain** è una tecnologia di registro distribuito che consente di registrare transazioni in maniera sicura, trasparente ed immutabile. Ogni transazione è raccolta in un *blocco* e i blocchi sono concatenati in ordine cronologico, formando una catena. Ogni *blocco* contiene un riferimento crittografico al *blocco* precedente, al fine di garantire l'integrità della catena [Nakamoto, 2008].

**Definizione A.0.96.** Uno **smart contract** è un contratto auto-esecutivo in cui i termini dell'accordo tra le parti sono scritti direttamente nel codice memorizzato in una *blockchain* [A.0.95], al fine di permettere l'esecuzione automatica e sicura delle transazioni e degli accordi senza la necessità di intermediari [Buterin, 2014].

**Definizione A.0.97.** La **deserializzazione** è il processo di conversione di una rappresentazione di dati in formato seriale (come un file o un flusso di dati) in un oggetto o una struttura di dati utilizzabile in memoria. Questo processo è l'opposto della serializzazione, che converte un oggetto in un formato che può essere facilmente memorizzato o trasmesso [Fowler, 2002].

# Bibliografia

[AngularJS Authors, 2021] AngularJS Authors (2021). Angularjs documentation.

[Anisetti et al., 2023] Anisetti, M., Ardagna, C. A., and Bena, N. (2023). Continuous certification of non-functional properties across system changes. In *Service-Oriented Computing*, pages 3–18, Cham. Springer Nature Switzerland.

[Anisetti et al., 2019] Anisetti, M., Ardagna, C. A., Gaudenzi, F., and Damiani, E. (2019). A continuous certification methodology for devops.

[Arbaugh et al., 2000] Arbaugh, W. A., Farber, D. J., and Smith, J. M. (2000). *Windows of Vulnerability: A Case Study Analysis*. IEEE.

[Atzori et al., 2010] Atzori, L., Iera, A., and Morabito, G. (2010). The internet of things: A survey. *Computer Networks*, 54(15):2787–2805.

[Awduche et al., 2002] Awduche, D., Chiu, A., Elwalid, A., Widjaja, I.-S., and Xiao, X. (2002). Overview and principles of internet traffic engineering.

[Bass et al., 2003] Bass, L., Clements, P., and Kazman, R. (2003). *Software Architecture in Practice*. Addison-Wesley Professional.

[Bass et al., 2015] Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, Boston, MA.

[Beck et al., 2001] Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J., and Thomas, D. (2001). Manifesto for agile software development.

[Bellman, 1957] Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

[Bernstein, 2014] Bernstein, D. (2014). Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84.

[Bishop, 2005] Bishop, M. (2005). *Introduction to Computer Security*. Addison-Wesley Professional.

[Bondi, 2000] Bondi, A. B. (2000). Characteristics of scalability and their impact on performance. *Proceedings of the second international workshop on Software and performance*, pages 195–203.

[Brown and Hallinan, 2015] Brown, M. and Hallinan, C. (2015). Device tree documentation.

[Buterin, 2014] Buterin, V. (2014). A next-generation smart contract and decentralized application platform. *Ethereum White Paper*.

[Cadar and Sen, 2013] Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90.

[Chacon and Straub, 2014] Chacon, S. and Straub, B. (2014). *Pro Git*. Apress.

[Chattopadhyay et al., 2020] Chattopadhyay, A., Lam, K. Y., et al. (2020). Autonomous vehicle: Security by design. *IEEE Transactions on Dependable and Secure Computing*.

[Chess and West, 2007] Chess, B. and West, J. (2007). *Secure programming with static analysis*. Addison-Wesley Professional.

[Chikofsky and Cross, 1990] Chikofsky, E. J. and Cross, J. H. (1990). Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17.

[Choudhary, 2007] Choudhary, V. (2007). Software as a service: Implications for investment in software development. *2007 40th Annual Hawaii International Conference on System Sciences (HICSS)*, pages 209a–209a.

[CISA, 2023] CISA (2023). Shifting the balance of cybersecurity risk: Principles and approaches for secure by design software. https://www.cisa.gov/secure-by-design.

[Clarke et al., 2004] Clarke, E. M., Kroening, D., and Lerda, F. (2004). Tool support for producing reliable software. *Oxford University Computing Laboratory, Tech. Rep.*

[Clarke and Schlingloff, 2008] Clarke, E. M. and Schlingloff, H. (2008). The birth of model checking. *25 Years of Model Checking*, pages 1–26.

[Clarke and Wing, 1996] Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. In *ACM Computing Surveys (CSUR)*, volume 28, pages 626–643. ACM.

[Colavita, 2016] Colavita, F. (2016). Devops movement of enterprise agile breakdown silos, create collaboration, increase quality, and application speed. In *Proceedings of 4th International Conference in Software Engineering for Defence Applications*. Springer.

[CONCORDIA, 2021] CONCORDIA (2021). H2020-concordia project.

[Cooper et al., 2008] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and Polk, T. (2008). Internet x.509 public key infrastructure certificate and certificate revocation list (crl) profile.

[Council, 2022] Council, P. C. I. S. S. (2022). Payment card industry (pci) data security standard (dss) v4.0.

[Cowan et al., 2000] Cowan, C., Pu, C., Maier, D., Hinton, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., and Zhang, Q. (2000). Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 2, pages 119–129. IEEE.

[Crockford, 2006] Crockford, D. (2006). *JSON: The JavaScript Object Notation Data Interchange Format*. ECMA International.

[Davis and Daniels, 2016] Davis, J. and Daniels, R. (2016). *Effective DevOps: building a culture of collaboration, affinity, and tooling at scale*. O'Reilly Media, Inc.

[Dierks and Rescorla, 2008] Dierks, T. and Rescorla, E. (2008). The transport layer security (tls) protocol version 1.2.

[Dowd et al., 2006] Dowd, M., McDonald, J., and Schuh, J. (2006). The art of software security assessment: Identifying and preventing software vulnerabilities. *Pearson Education*.

[ENISA, 2021] ENISA (2021). European union agency for cybersecurity (enisa).

[et al., 2016] et al., S. (2016). Resource provisioning and scheduling in clouds: Qos perspective. *The Journal of Supercomputing*, 72(3):926–960.

[European Commission, 2020a] European Commission (2020a). Assessment list for trustworthy artificial intelligence (altai).

[European Commission, 2020b] European Commission (2020b). Capability assessment for ai.

[Fielding, 2000] Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine.

[FIRST, 2019] FIRST (2019). Common vulnerability scoring system v3.1: Specification document.

[Fowler, 2002] Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

[Furber, 2000] Furber, S. (2000). *ARM System-on-Chip Architecture*. Addison-Wesley.

[Gürel and Tat, 2017] Gürel, E. and Tat, M. (2017). Swot analysis: A theoretical review. *Journal of International Social Research*, 10(51):994–1006.

[Hallinan, 2006] Hallinan, C. (2006). Kernel debugging techniques. *Linux Journal*.

[Hardt, 2012] Hardt, D. (2012). The oauth 2.0 authorization framework.

[Hennessy and Patterson, 2011] Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

[Hernan et al., 2006] Hernan, S., Lambert, S., Ostwald, T., and Shostack, A. (2006). Threat modeling: Stride.

[Hinden and Deering, 2006] Hinden, R. and Deering, S. (2006). Internet protocol, version 6 (ipv6) specification.

[Hoglund and McGraw, 2004] Hoglund, G. and McGraw, G. (2004). *Exploiting Software: How to Break Code*. Addison-Wesley Professional.

[IEEE, 2014] IEEE (2014). *IEEE Std 730-2014, IEEE Standard for Software Quality Assurance Processes*. IEEE.

[IEEE, 2016] IEEE (2016). Ieee standard for system and software verification and validation.

[IICS WG, 2015] IICS WG (2015). Supplemental information for the interagency report on strategic u.s. government engagement in international standardization to achieve u.s. objectives for cybersecurity. Technical report, NIST.

[ISO, 2008] ISO (2008). Iso/iec 21827:2008. Standard, ISO.

[ISO, 2013] ISO (2013). Iso/iec 27001:2013 information technology — security techniques — information security management systems — requirements.

[ISO, 2018a] ISO (2018a). Iso 19011:2018 guidelines for auditing management systems.

[ISO, 2018b] ISO (2018b). Iso 31000:2018 risk management — guidelines.

[ISO, 2018c] ISO (2018c). Iso 9004:2018 quality management — quality of an organization — guidance to achieve sustained success.

[ISO, 2018d] ISO (2018d). Iso/iec 9899:2018 information technology — programming languages — c.

[Johnson, 1989] Johnson, B. W. (1989). *Fault-Tolerant Computer System Design*. Prentice Hall.

[Jones et al., 2015] Jones, M., Bradley, J., and Sakimura, N. (2015). Json web token (jwt).

[Kim et al., 2016] Kim, G., Humble, J., Debois, P., and Willis, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, Security in Technology Organizations*. IT Revolution Press.

[Kleissner, 2009] Kleissner, P. (2009). Stoned bootkit. *Black Hat USA*.

[Kocher, 1996] Kocher, P. C. (1996). Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer.

[Krueger, 1992] Krueger, C. W. (1992). Software reuse. *ACM Computing Surveys (CSUR)*, 24(2):131–183.

[Kubernetes Authors, 2021] Kubernetes Authors (2021). Kubernetes documentation.

[Laprie, 2008] Laprie, J.-C. (2008). From dependability to resilience. In *2008 38th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages G8–G9. IEEE.

[Levine, 2009] Levine, R. (2009). *Linux System Programming*. O'Reilly Media.

[Luz et al., 2018] Luz, W. P., Pinto, G., and Bonifácio, R. (2018). Building a collaborative culture: a grounded theory of well succeeded devops adoption in practice. In *Proceedings of the 12th ACM/IEEE International Symposium*.

[Lynch, 2006] Lynch, J. M. (2006). *Hardening Network Security*. McGraw-Hill Osborne Media.

[McConnell, 2004] McConnell, S. (2004). *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press.

[Mell and Grance, 2011] Mell, P. and Grance, T. (2011). The nist definition of cloud computing.

[Mell et al., 2006] Mell, P., Scarfone, K., and Romanosky, S. (2006). Common vulnerability scoring system.

[Mell et al., 2007] Mell, P., Scarfone, K., and Romanosky, S. (2007). The common vulnerability scoring system (cvss) and its applicability to federal agency systems. Technical Report NISTIR 7435, NIST.

[Merkel, 2014] Merkel, D. (2014). Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2.

[Microsoft, 1996] Microsoft (1996). Fat32 file system specification.

[Miller et al., 2016] Miller, J. S., Veiga, F., Shakib, A., and Nelson, N. (2016). *Hybrid Cloud for Dummies*. John Wiley & Sons.

[Mirkovic and Reiher, 2004] Mirkovic, J. and Reiher, P. (2004). A taxonomy of ddos attack and ddos defense mechanisms. *ACM SIGCOMM*, 34(2):39–53.

[Mohan and Singh, 2018] Mohan, V. and Singh, P. (2018). Building secure ci/cd pipelines. In *Proceedings of the 10th International Conference on Security and its Applications*. ACM.

[MoonCloud, 2024] MoonCloud (2024). Mooncloud: Continuous compliance assessment and assurance platform.

[Myers et al., 2011] Myers, G. J., Sandler, C., and Badgett, T. (2011). *The Art of Software Testing*. John Wiley & Sons.

[Nakamoto, 2008] Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system.

[Newman, 2015] Newman, S. (2015). *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media.

[NIST, 2013] NIST (2013). Nist special publication 800-53, security and privacy controls for federal information systems and organizations. Technical report.

[NIST, 2021] NIST (2021). National institute of standards and technology (nist).

[Open Web Application Security Project, 2010] Open Web Application Security Project (2010). Owasp secure coding practices - quick reference guide.

[Open Web Application Security Project, 2017a] Open Web Application Security Project (2017a). Cross-site scripting (xss).

[Open Web Application Security Project, 2017b] Open Web Application Security Project (2017b). Sql injection.

[Pressman, 2005] Pressman, R. S. (2005). *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math.

[Rahman, 2016] Rahman, A. (2016). Integrating security into the devops culture. *IEEE Security & Privacy*, 14(3):35–45.

[Raymond, 2001] Raymond, E. S. (2001). *The Cathedral the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly Media.

[RedHat, 2024] RedHat (2024). Metodologia devsecops: Sviluppo, sicurezza, operazioni. *RedHat*.

[Sabt et al., 2015] Sabt, M., Achemlal, M., and Bouabdallah, A. (2015). Trusted execution environment: What it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64.

[Sadowski et al., 2018] Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., and Jaspan, C. (2018). Lessons from building static analysis tools at google. *Communications of the ACM*, 61(4):58–66.

[Satyanarayanan, 2017] Satyanarayanan, M. (2017). The emergence of edge computing. *Computer*, 50(1):30–39.

[Scarfone and Mell, 2007] Scarfone, K. and Mell, P. (2007). Guide to intrusion detection and prevention systems (idps).

[Seacord, 2005] Seacord, R. C. (2005). *Secure Coding in C and C++*. Addison-Wesley Professional.

[Shostack, 2014] Shostack, A. (2014). *Threat Modeling: Designing for Security*. John Wiley Sons.

[Skoudis and Zeltser, 2004] Skoudis, E. and Zeltser, L. (2004). *Malware: Fighting Malicious Code*. Prentice Hall Professional.

[Somerville, 2011] Somerville, I. (2011). *Software Engineering*. Addison-Wesley.

[Spinellis, 2006] Spinellis, D. (2006). *Code Quality: The Open Source Perspective*. Addison-Wesley.

[Srivastav et al., 2023] Srivastav, S., Allam, K., and Mustyala, A. (2023). Software automation enhancement through the implementation of devops. *International Journal of Research*.

[Stallings, 2015] Stallings, W. (2015). *Network Security Essentials: Applications and Standards*. Pearson.

[Stol and Fitzgerald, 2022] Stol, K.-J. and Fitzgerald, B. (2022). Devops and the integration of security: A systematic literature review. *Journal of Systems and Software*, 185:111209.

[Sutton et al., 2007] Sutton, M., Greene, A., and Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional.

[Tanenbaum and Bos, 2007] Tanenbaum, A. S. and Bos, H. (2007). *Modern Operating Systems*. Prentice Hall.

[The MITRE Corporation, 2021a] The MITRE Corporation (2021a). Cve - common vulnerabilities and exposures.

[The MITRE Corporation, 2021b] The MITRE Corporation (2021b). Cwe - common weakness enumeration.

[Turnbull, 2014] Turnbull, J. (2014). *The Docker Book: Containerization is the New Virtualization*. James Turnbull.

[Upton and Halfacree, 2014] Upton, E. and Halfacree, G. (2014). *Raspberry Pi User Guide*. John Wiley & Sons.

[Voigt and Von dem Bussche, 2017] Voigt, P. and Von dem Bussche, A. (2017). *The EU General Data Protection Regulation (GDPR): A Practical Guide*. Springer.

[Ylonen, 1996] Ylonen, T. (1996). The secure shell (ssh) protocol.

Progetto sviluppato presso il SEcure Service-oriented Architectures Research (SESAR) Lab
http://sesar.di.unimi.it