



UNIVERSITA' DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

Elaborato per l'esame di Software Security

*Sicurezza nel Ciclo di Vita del
Software: un caso pratico con
Microsoft SDL in ambiente
cloud-native*

Anno Accademico 2024/2025

Docente
Roberto Natella

Autori
Francesco Scognamiglio M63001364
Felice Micillo M63001377

Indice

1	Introduzione	1
1.0.1	Obiettivo	2
2	Define Security Requirements	3
2.1	Modellazione tramite Abuse Case	4
2.2	Abuse Case Mapping	4
2.3	Implementazioni effettive delle contromisure	6
3	Define Metrics and Compliance Reporting	8
3.1	Vulnerabilità Rilevate e Risoluzione	9
3.2	Tracciamento e Reporting	10
4	Perform Threat Modeling (Privacy included)	12
4.1	Data Flow Diagram (DFD) dell'architettura	13
4.2	STRIDE Threat Modeling	14
4.3	Privacy Threat Modeling (LINDDUN)	15
5	Define And Use Cryptography Standards	17

5.1	Crittografia e gestione delle chiavi in Keycloak	18
5.2	Crittografia delle comunicazioni HTTPS	20
5.3	Firma di token personalizzati nel backend con <code>jsonwebtoken</code>	21
6	Manage Security Risk Of Third-Party Components	23
6.1	Analisi delle dipendenze Node.js	23
6.2	Analisi delle immagini container	24
6.3	Controllo dei chart Helm	26
7	Use Approved Tools	27
7.1	Ambienti di Sviluppo e IDE	28
7.2	Framework, Runtime e Librerie	29
7.3	Immagini Docker verificate e firmate	30
7.4	Conservazione e tracciabilità delle firme	32
8	Perform SAST e DAST (con GitLab)	34
8.1	Static Application Security Testing (SAST)	35
8.2	Dynamic Application Security Testing (DAST)	37
8.3	Raccolta dei report	38

Capitolo 1

Introduzione

In un contesto in cui la sicurezza del software non può più essere concepita come un'aggiunta a posteriori, ma come un elemento integrante dell'intero processo di sviluppo, il presente lavoro si propone di estendere un'infrastruttura già esistente — sicura, modulare e basata su tecnologie *cloud-native* — con una prospettiva metodologica e sistematica derivante dal modello *Microsoft Security Development Lifecycle (SDL)*.

Il progetto di base, già implementato, si fonda su un backend containerizzato orchestrato tramite *Kubernetes*, con una pipeline CI/CD automatizzata e una serie di misure di sicurezza già operative sia a livello infrastrutturale che applicativo. Tuttavia, ciò che si intende realizzare in questa fase è un percorso di formalizzazione e consolidamento dei controlli di sicurezza esistenti, e di eventuale integrazione di pratiche mancanti, secondo i principi del *Security Life Cycle (SLC)*.

1.0.1 Obiettivo

L'obiettivo principale è quello di applicare selettivamente e in maniera critica le attività del Microsoft SDL al progetto esistente, in modo da:

- Valutare in modo strutturato i rischi di sicurezza già mitigati e quelli ancora presenti;
- Mappare ogni attività tecnica a uno o più punti dello SDL;
- Individuare strumenti automatizzabili per estendere i controlli o formalizzare quelli esistenti;
- Offrire una visione completa e giustificata del ciclo di vita della sicurezza del software adottato.

Il risultato finale sarà un documento tecnico accompagnato da una presentazione che descrive il processo seguito, le scelte effettuate, gli strumenti impiegati e le evidenze raccolte per ciascuna attività SDL selezionata.

Capitolo 2

Define Security Requirements

Una delle prime attività previste nel *Security Development Lifecycle (SDL)* consiste nell'identificare in maniera formale i requisiti di sicurezza a partire dai requisiti funzionali dell'applicazione. Questa operazione permette di adottare un approccio proattivo alla sicurezza, prevenendo scenari di attacco sin dalle fasi iniziali e codificando le contromisure nel design e nello sviluppo.

Nel contesto del nostro progetto, in cui l'autenticazione è gestita tramite *Keycloak* e l'accesso alle API avviene mediante *token JWT*, abbiamo scelto di modellare i requisiti di sicurezza usando il metodo degli abuse case. Tale approccio permette di ragionare a partire dai casi d'uso esistenti per individuare potenziali abusi e attori malevoli, e progettare le opportune contromisure.

A differenza dell'approccio UML classico, abbiamo adottato una rappresentazione tabellare più immediata e leggibile, che mantiene l'efficacia del modello senza appesantirne la formalizzazione.

2.1 Modellazione tramite Abuse Case

Ogni funzionalità dell'applicazione è stata analizzata con una doppia prospettiva:

- Qual è l'intento legittimo dell'uso previsto?
- Come potrebbe essere abusata da un attore malevolo, interno o esterno?

Per ogni abuse case individuato, abbiamo valutato:

- L'attore coinvolto (es. bot, utente malintenzionato, attaccante esterno)
- Il livello di impatto stimato (basso, medio, alto)
- Le contromisure già presenti nel sistema o da implementare

L'obiettivo è mappare i rischi connessi ai flussi applicativi reali, non solo da un punto di vista teorico, ma anche in relazione all'implementazione attuale del progetto.

2.2 Abuse Case Mapping

USE CASE	ABUSE CASE	DETTAGLI
Login	Tentativo di brute force tramite credenziali ripetute	Attore: Attaccante esterno Impatto: Alto Contromisure: Rate limiting, CAPTCHA, blocco temporaneo account
Signup	Registrazioni automatizzate da bot	Attore: Bot/script Impatto: Medio Contromisure: CAPTCHA, verifica email obbligatoria
Reset Password	Furto del link/token di reset	Attore: Intercettazione token (attaccante esterno) Impatto: Alto Contromisure: Token one-time, scadenza breve, trasmissione HTTPS
Accesso area GraphQL (create/update/delete esperienze)	Accesso non autorizzato a dati privati o non propri	Attore: Utente autenticato malevolo Impatto: Alto Contromisure: Verifica proprietà risorse lato server (es. autore == user.id)
Gestione OTP	Attivazione OTP non valida o spoofing codice	Attore: Utente o attaccante che intercetta flusso Impatto: Alto Contromisure: Validazione sicura codice OTP, timeout configurabile, logging
Modifica email/username	Cambiamento non autorizzato dell'identità	Attore: Utente autenticato Impatto: Alto Contromisure: Verifica ownership, verifica email post-modifica

Tabella 2.1: Mappatura Use Case vs Abuse Case

2.3 Implementazioni effettive delle contromisure

Le contromisure individuate nella fase di modellazione sono state in buona parte realizzate e integrate direttamente nel progetto, sfruttando le funzionalità offerte da *Keycloak*, il backend *Express* e i meccanismi di verifica interna dell'applicazione. Di seguito riportiamo una descrizione puntuale delle implementazioni:

- **Login – Rate Limiting**

La protezione contro brute force è gestita nativamente da Keycloak, che consente di configurare i limiti di tentativi di accesso falliti direttamente dal pannello amministrativo del realm. CAPTCHA e blocco account non sono attualmente implementati, in quanto il rate limiting è ritenuto sufficiente per il profilo di rischio atteso.

- **Signup – Verifica email**

Dopo la registrazione, gli utenti non possono accedere finché non hanno verificato l'indirizzo email. Questo è forzato a livello di realm Keycloak, e supportato da una API custom che genera un link contenente un token firmato inviato via email. Solo cliccando su tale link, l'account viene attivato.

- **Reset Password – Token email one-time**

Il reset della password è protetto da un meccanismo di token

monouso con scadenza temporale, trasmesso via email. La pagina di reset può essere raggiunta solo tramite un link contenente questo token. La validità e unicità del token vengono verificate prima di permettere la modifica.

- **Accesso area GraphQL – Controlli di ownership**

Tutti i resolver GraphQL sono protetti da un middleware (requireToken) che verifica la presenza di un access token valido. Inoltre, per operazioni su risorse personali (es. esperienze), viene verificata esplicitamente la proprietà (`authorId === user.id`) per impedire modifiche non autorizzate.

- **Gestione OTP – Integrazione nativa Keycloak**

L'autenticazione a due fattori è gestita interamente da Keycloak, che fornisce funzionalità per la configurazione e validazione del codice OTP, basato su TOTP ciclico. Il flusso di setup è assistito da API che espongono il QR code e attivano l'OTP previa verifica.

- **Modifica Email/Username – Accesso autenticato**

Le operazioni di modifica del profilo utente sono permesse solo agli utenti autenticati. È inoltre prevista una verifica post-modifica: in particolare, la modifica dell'email impone una nuova verifica, mantenendo l'account in stato “non verificato” finché il nuovo indirizzo non viene confermato.

Capitolo 3

Define Metrics and Compliance Reporting

La sicurezza del software non può essere migliorata senza una misurazione sistematica dei problemi emersi e delle attività correttive intraprese. Il terzo punto del Microsoft SDL propone di definire metriche di sicurezza e strumenti di reporting per tenere traccia dei rischi noti, delle vulnerabilità rilevate e dello stato delle contromisure adottate.

Nel contesto del nostro progetto, l'approccio è stato pragmatico e focalizzato sulla registrazione e risoluzione di vulnerabilità individuate tramite tool automatici. In particolare, è stato utilizzato *NodeJsScan* nella pipeline CI/CD, che ha prodotto risultati significativi.

3.1 Vulnerabilità Rilevate e Risoluzione

Tra le varie segnalazioni, la più rilevante ha riguardato l'assenza di HTTP security headers, fondamentali per mitigare attacchi comuni come *XSS*, *clickjacking* o *MIME sniffing*. Gli header inizialmente mancanti erano:

- Content-Security-Policy
- X-Frame-Options
- Strict-Transport-Security
- X-Content-Type-Options
- X-XSS-Protection
- X-Download-Options
- X-Powered-By
- Public-Key-Pins (HPKP) (ignorato in quanto deprecato)

Per correggere questa criticità è stato introdotto un middleware personalizzato in *Express* che aggiunge in modo sistematico tutti gli header consigliati, portando la severità delle vulnerabilità da "Medium/High" a "Resolved", ad eccezione di HPKP.

```
// Security headers middleware
app.use((req, res, next) => {
  res.setHeader('Content-Security-Policy', "default-src 'none'; frame-ancestors 'none'");
  res.setHeader('X-Frame-Options', 'DENY');
  res.setHeader('Strict-Transport-Security', 'max-age=63072000; includeSubDomains');
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader('X-XSS-Protection', '1; mode=block');
  res.setHeader('X-Download-Options', 'noopen');
  res.removeHeader('X-Powered-By');
  next();
});
```

Figura 3.1: Middleware in *server.js*.

3.2 Tracciamento e Reporting

Per garantire visibilità, tracciabilità e responsabilità nella gestione delle vulnerabilità, è stato introdotto un meccanismo di reporting semplice ma efficace: la creazione e il mantenimento di un file denominato *SECURITY_METRICS.md* nel repository del progetto.

Tale file contiene una tabella versionata che documenta le vulnerabilità rilevate tramite tool automatici (es. NodeJsScan), assegnando a ciascuna un punteggio di rischio, uno stato di risoluzione e, se disponibile, la contromisura implementata.

La struttura adottata è la seguente:

VULNERABILITÀ	GRAV.	PROB.	RISCHIO	STATO
Content-Security-Policy assente	5	4	20	Risolta
X-Frame-Options mancante	4	3	12	Risolta
Strict-Transport-Security assente	4	2	8	Risolta
X-Content-Type-Options mancante	3	3	9	Risolta
X-XSS-Protection non configurato	3	2	6	Risolta
X-Download-Options mancante	2	2	4	Risolta
Public-Key-Pins (HPKP) mancante	1	1	1	Ignorata

Tabella 3.1: Valutazione delle vulnerabilità di sicurezza HTTP Header

Il file di report ha un duplice scopo:

- Tecnico, perché permette al team di sicurezza o agli sviluppatori di avere un riferimento aggiornato sui problemi rilevati e trattati.
- Organizzativo, perché può essere usato come punto di partenza per definire KPI interni.

Capitolo 4

Perform Threat Modeling (Privacy included)

Nel contesto dello sviluppo di software sicuro, il *Threat Modeling* rappresenta una delle attività più critiche e ad alto impatto. Lo scopo di questa fase è identificare, classificare e valutare in anticipo le potenziali minacce alla sicurezza e alla privacy del sistema, ancora prima che queste possano manifestarsi sotto forma di vulnerabilità reali o attacchi.

Nel nostro progetto, il Threat Modeling è stato strutturato seguendo le linee guida del *Microsoft Security Development Lifecycle (SDL)* e ha incluso due prospettive complementari:

- **STRIDE**: un modello classico orientato alla sicurezza, utile per identificare minacce tecniche nei flussi di dati, nei componenti software e nei punti di interazione del sistema.

- **LINDDUN:** un modello focalizzato sulla privacy, ideato per rilevare rischi legati al trattamento dei dati personali, come tracciabilità, identificabilità e divulgazione non autorizzata.

4.1 Data Flow Diagram (DFD) dell'architettura

Per effettuare una corretta analisi delle minacce, è stato realizzato un *Data Flow Diagram (DFD)* dell'architettura utilizzando *Microsoft Threat Modeling Tool*. Lo schema rappresenta l'intero flusso dei dati tra i componenti principali del nostro sistema, basato su Kubernetes e diviso in due namespace principali: *ingress-nginx* e *app*.

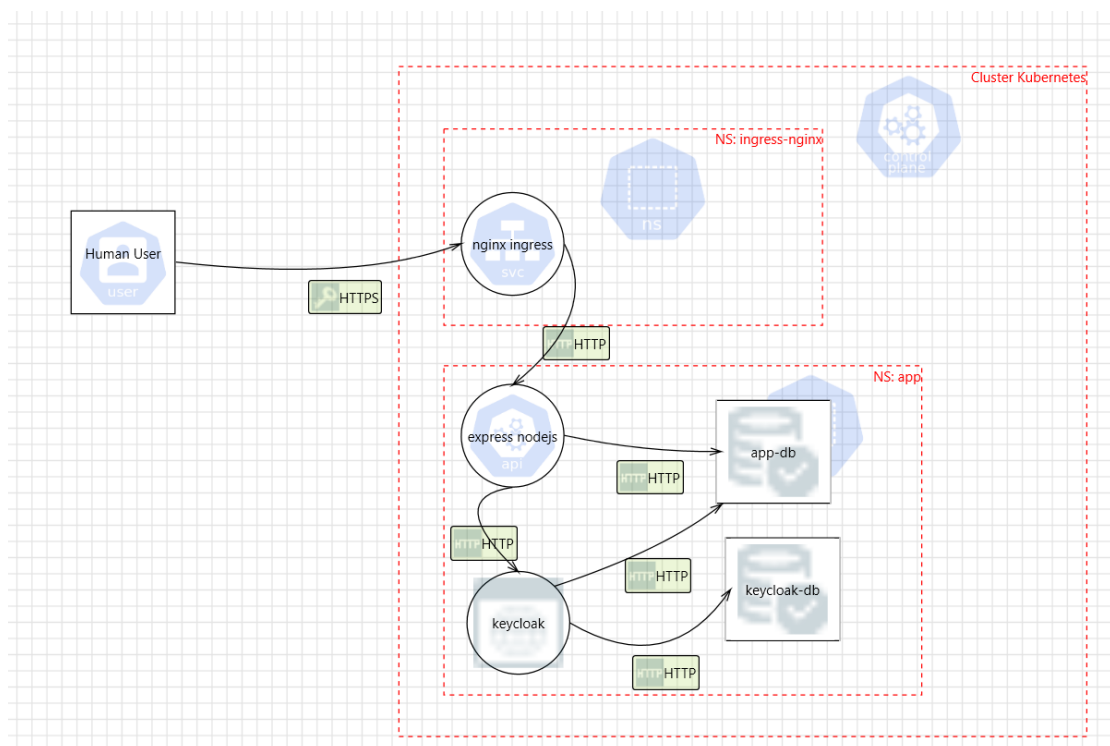


Figura 4.1: DFD in Microsoft Threat Modeling Tool.

Nel DFD sono evidenziati i seguenti componenti:

- Human User (esterno)
- nginx ingress controller, che gestisce le richieste HTTPS esterne
- express nodejs API, che funge da backend applicativo
- Keycloak, come sistema di autenticazione e gestione identità
- Database applicativo (app-db) e Keycloak DB (keycloak-db)

I flussi di dati (data flows) includono:

- HTTPS dal client a nginx ingress
- HTTP interni tra i servizi (giustificato successivamente)
- Connessioni da express verso i DB e da Keycloak ai suoi sottosistemi

Alcune comunicazioni interne avvengono tramite HTTP non cifrato. Tuttavia, l'intero cluster è isolato su una rete privata Kubernetes, senza esposizione pubblica dei servizi interni, e ciò riduce drasticamente la superficie di attacco.

4.2 STRIDE Threat Modeling

L'analisi è stata condotta usando il *Microsoft Threat Modeling Tool*, che ha generato automaticamente le minacce potenziali secondo il modello *STRIDE*. Per ogni elemento (processi, datastore, flussi), il tool

verifica se si possono applicare i sei tipi di minacce. I risultati sono stati esportati e classificati nel file *STRIDE.csv*.

Abbiamo poi valutato ogni threat per:

- Impatto
- Likelihood (probabilità)
- Risk Score (prodotto dei due)
- Stato di mitigazione

Di seguito una tabella riassuntiva con esempi rappresentativi:

Unnamed: 0	Elemento	Threat Type	Descrizione	Impatto	Likelihood	Risk Score	Stato	Mitigazione Applicata
0	HTTP	Spoofing of Destination Data Store app-db	app-db may be spoofed by an attacker and this may lead to data being written to the attacker's target instead of app-db. Consider using a standard authentication mechanism to identify the destination data store.	Alto	Medio	4	Mitigated	Mitigazione generica tramite rete privata e access control
1	HTTP	Potential SQL Injection Vulnerability for app-db	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker.	Alto	Medio	4	Mitigated	Uso di ORM Prisma che parametrizza le query
2	HTTP	Potential Excessive Resource Consumption for express nodejs or app-db	Does express nodejs or app-db take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.	Alto	Medio	4	Mitigated	Rate limiting e monitoraggio accessi nel cluster interno
3	HTTP	Potential Excessive Resource Consumption for keycloak or app-db	Does keycloak or app-db take explicit steps to control resource consumption? Resource consumption attacks can be hard to deal with, and there are times that it makes sense to let the OS do the job. Be careful that your resource requests don't deadlock, and that they do timeout.	Alto	Medio	4	Mitigated	Rate limiting e monitoraggio accessi nel cluster interno
4	HTTP	Potential SQL Injection Vulnerability for app-db	SQL injection is an attack in which malicious code is inserted into strings that are later passed to an instance of SQL Server for parsing and execution. Any procedure that constructs SQL statements should be reviewed for injection vulnerabilities because SQL Server will execute all syntactically valid queries that it receives. Even parameterized data can be manipulated by a skilled and determined attacker.	Alto	Medio	4	Mitigated	Uso di ORM Prisma che parametrizza le query

Figura 4.2: STRIDE

Il report completo è situato nella repository del progetto.

4.3 Privacy Threat Modeling (LINDDUN)

Per valutare i rischi legati alla privacy, abbiamo applicato il framework *LINDDUN*, che analizza i potenziali problemi in sette categorie:

- Linkability

CAPITOLO 4. PERFORM THREAT MODELING (PRIVACY INCLUDED)

- Identifiability
- Non-repudiation
- Detectability
- Data Disclosure
- Unawareness
- Non-compliance

L'analisi si è basata sullo stesso DFD del sistema, focalizzandosi su processi e datastore che trattano dati personali (es. Keycloak, app-db).

Componente	Categoria	Descrizione	Impatto	Likelihood	Risk Score	Mitigazione Applicata
Ingress	Detectability	Un attaccante esterno può dedurre pattern di utilizzo (es. orari di login)	3	2	6	Rate limiting, logging minimizzato
Express API	Linkability	Cross-request correlation tramite cookie o token persistenti	3	3	9	Uso di cookie "SameSite", token brevi e rotazione periodica
Express API	Unawareness	Gli utenti non sanno quali dati vengono raccolti durante le chiamate API	3	3	9	Privacy policy esplicita, banner di consenso
Express API	Non-compliance	Raccolta di dati non strettamente necessari per il servizio	4	2	8	Data minimization in input validation, revisione periodica dei campi raccolti
Keycloak	Identifiability	Possibilità di riconoscere univocamente un utente tramite metadata (es. indirizzo IP)	5	3	15	Anonimizzazione IP nei log, TTL corto per UUID, storage cifrato
Keycloak	Data Disclosure	Esposizione accidentale di credenziali o token	5	2	10	Hashing + salting, access control
Keycloak DB	Non-repudiation	Impossibilità di verificare la paternità di operazioni sul DB	4	2	8	Log firmati e timestamped, audit trail immutabile
app-db	Data Disclosure	Accesso diretto da pod non autorizzati che potrebbe esporre dati utente	4	2	8	NetworkPolicy Kubernetes, least privilege sui ServiceAccount
app-db	Linkability	Possibilità di correlare record tra tabelle e ricostruire profili utente	3	2	6	Pseudonimizzazione campi sensibili, separazione fisica o schema dei dati
Tutti i Log	Unawareness	Gli utenti non sanno che le loro azioni vengono loggate e per quanto tempo conservate	3	4	12	Log retention limitata, comunicazioni trasparenti nella privacy policy
Tutti i DB	Non-compliance	Conservazione dati oltre i termini previsti dalla normativa (es. GDPR)	4	2	8	Implementazione di processi automatici di data purging

Figura 4.3: LINDDUN

Capitolo 5

Define And Use

Cryptography Standards

In questa fase del ciclo di vita sicuro del software (SDL), abbiamo analizzato in dettaglio l'impiego della crittografia all'interno della nostra applicazione web, coprendo sia la parte di autenticazione e gestione identità (tramite Keycloak), sia le comunicazioni HTTPS (protette con Cert Manager e Let's Encrypt), sia la firma di token personalizzati attraverso la libreria `jsonwebtoken` nel backend Node.js.

L'obiettivo è stato quello di garantire:

- l'utilizzo di algoritmi considerati sicuri a livello industriale (es. RS256, AES, HS256, TLS 1.3),
- dimensioni delle chiavi adeguate (minimo 2048 bit per RSA, 256 bit per chiavi simmetriche),

- l'uso di chiavi generate in modo sicuro e mai hardcoded nel codice,
- un monitoraggio attivo di eventuali vulnerabilità nelle librerie crittografiche usate.

5.1 Crittografia e gestione delle chiavi in Keycloak

Keycloak rappresenta il cuore del sistema di autenticazione e autorizzazione dell'applicazione. Uno degli aspetti cruciali per la sua sicurezza è la corretta gestione dei meccanismi crittografici, utilizzati per la firma e (in alcuni casi) la cifratura dei token emessi. Il sistema supporta diversi algoritmi e tipologie di chiavi, visibili e gestibili attraverso la sezione “Keys” del Realm amministrativo.

Algorithm	Type	Kid	Use	Provider	Valid to	Public keys
HS512	OCT	40708818-9f21-41f2-bf98-7aea9df4ac1f	SIG	hmac-generated-hs512	-	
AES	OCT	edc0e604-9d61-4c44-bae6-101627187df	ENC	aes-generated	-	
RSA-OAEP	RSA	u3f5aWwMozErObAlaxPO-HVvx4XMpcbCrdPeO2RVA	ENC	rsa-enc-generated	6/3/2035, 9:51:00 PM	Public key Certificate
RS256	RSA	IPPeZlBmmTPSY3LZlR5KHVRyUlcWwU9DaOcyUq4	SIG	rsa-generated	6/3/2035, 9:51:01 PM	Public key Certificate

Figura 5.1: Admin Console di Keycloak.

Algoritmi utilizzati

Dalla console amministrativa di Keycloak, è stato verificato che sono attivi i seguenti algoritmi:

- **RS256 (RSA Signature)**: usato per la firma dei token JWT, garantisce autenticità e integrità. Le chiavi RSA sono di lunghezza pari a 2048 bit.
- **RSA-OAEP**: utilizzato per la cifratura asimmetrica, ad esempio in contesti SAML o trasmissione di dati sensibili.
- **AES (Advanced Encryption Standard)**: utilizzato per crittografia simmetrica interna, con chiavi a 256 bit.
- **HS512 (HMAC SHA-512)**: disponibile come metodo alternativo per la firma, utilizzando una chiave segreta simmetrica (OCTET).

Sicurezza e gestione

Tutte le chiavi sono state generate tramite provider sicuri di Keycloak, e non vengono esposte o esportate. I token firmati (ID token, Access token, Refresh token) sono verificabili da qualsiasi consumer tramite la chiave pubblica esposta sull'endpoint:

`/.well-known/openid-configuration`.

Validità e rotazione

Le chiavi principali risultano attive e valide fino al 2035, come da configurazione visualizzata nella console. In Keycloak è anche possibile attivare meccanismi di rotazione automatica delle chiavi e fallback su

chiavi precedenti, per garantire continuità di servizio anche in caso di aggiornamento.

5.2 Crittografia delle comunicazioni HTTPS

Per proteggere il traffico tra l'utente e l'applicazione, è stato configurato un sistema di cifratura end-to-end basato su TLS, utilizzando Let's Encrypt come Autorità di Certificazione (CA) e Cert Manager come strumento di gestione automatica dei certificati all'interno del cluster Kubernetes.

```
subject=CN=felixengineer.com
issuer=C=US, O=Google Trust Services, CN=WE1
---
No client certificate CA names sent
Peer signing digest: SHA256
Peer signature type: ECDSA
Server Temp Key: X25519, 253 bits
---
SSL handshake has read 2845 bytes and written 409 bytes
Verification error: unable to get local issuer certificate
---
New, TLSv1.3, Cipher is TLS_AES_256_GCM_SHA384
Server public key is 256 bit
This TLS version forbids renegotiation.
Compression: NONE
Expansion: NONE
No ALPN negotiated
Early data was not sent
Verify return code: 20 (unable to get local issuer certificate)
```

Figura 5.2: OpenSSL Certificate inspect.

Infrastruttura adottata

L'architettura prevede che tutte le richieste esterne passino attraverso un controller *nginx ingress*, che espone il servizio in HTTPS. La configurazione di Cert Manager consente la richiesta, l'emissione e il rinnovo automatico dei certificati X.509 da Let's Encrypt tramite il challenge ACME.

Dettagli del certificato TLS

Dall'ispezione del certificato in uso (tramite `openssl`), sono emersi i seguenti parametri:

- **Algoritmo di firma:** ECDSA con SHA-256
- **Chiave pubblica:** curva X25519 (lunghezza effettiva 253 bit)
- **Protocollo TLS:** versione 1.3
- **Cipher Suite:** TLS_AES_256_GCM_SHA384

5.3 Firma di token personalizzati nel backend con **jsonwebtoken**

Nel backend Node.js dell'applicazione, in aggiunta ai token OIDC generati da Keycloak, vengono utilizzati token JWT personalizzati per rap-

presentare informazioni temporanee, come conferme email o sessioni leggere. Per tale scopo viene impiegata la libreria `jsonwebtoken`.

Modalità d'uso

La funzione utilizzata nel codice è la seguente:

```
jwt.sign({ email }, secret, { expiresIn: '15m' });
```

Questa firma un payload contenente l'indirizzo email dell'utente, con un tempo di validità di 1 ora. Il token è firmato con un algoritmo simmetrico (default: HS256), che utilizza una chiave segreta precondivisa.

Sicurezza della chiave segreta

Per evitare vulnerabilità comuni (es. chiavi hardcoded o esposte in file), la chiave segreta utilizzata nella firma viene generata con alta entropia (512 caratteri) e gestita tramite HashiCorp Vault, integrato nel cluster Kubernetes tramite *Vault CSI driver*.

- La secret è montata dinamicamente come file nel container tramite CSI.
- Nessun valore sensibile è presente nel codice o nei manifesti Kubernetes.
- L'accesso è autorizzato tramite identity-based access control definito in Vault.

Capitolo 6

Manage Security Risk Of Third-Party Components

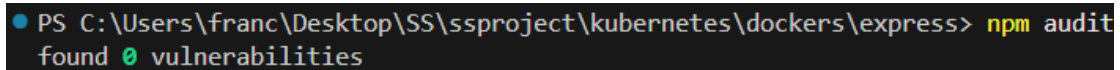
Durante lo sviluppo dell'applicazione è stato inevitabile l'uso di componenti di terze parti, siano essi librerie, container, Helm chart o interi servizi. Tuttavia, l'integrazione di questi elementi introduce un rischio significativo in termini di sicurezza, soprattutto se non monitorati nel tempo. L'obiettivo di questo step del ciclo SDL è identificare, analizzare e mitigare i rischi associati a tali componenti, attraverso l'uso di strumenti automatizzati e gestione delle dipendenze.

6.1 Analisi delle dipendenze Node.js

Il backend dell'applicazione è sviluppato in Node.js e fa ampio uso di pacchetti npm. Per verificarne la sicurezza, è stato eseguito un

controllo tramite il comando:

```
npm audit --production
```



```
PS C:\Users\franc\Desktop\SS\ssproject\kubernetes\docker\express> npm audit
found 0 vulnerabilities
```

Figura 6.1: npm audit su nodejs.

Il report non ha evidenziato alcuna vulnerabilità conosciuta nelle dipendenze runtime del progetto. La verifica è integrata anche nel processo CI/CD, garantendo che eventuali vulnerabilità future vengano rilevate tempestivamente.

6.2 Analisi delle immagini container

Sono state inoltre analizzate le immagini Docker utilizzate per i principali servizi in Kubernetes:

- **Keycloak** (immagine ufficiale: `quay.io/keycloak/keycloak:22`)
- **Node.js** (immagine base per il backend: `node:18`)
- **PostgreSQL** (database app e Keycloak: `postgres:15`)

Per l'analisi delle vulnerabilità è stato utilizzato lo strumento `Trivy`, che ha eseguito una scansione approfondita delle immagini. I risultati non hanno rilevato vulnerabilità applicative critiche o direttamente impattanti.

CAPITOLO 6. MANAGE SECURITY RISK OF THIRD-PARTY COMPONENTS

Target	Type	Vulnerabilities	Secrets
keycloak-trivy:v1 (redhat 9.6)	redhat	32	-
opt/keycloak/bin/client/keycloak-admin-cli-26.2.5.jar	jar	0	-
opt/keycloak/bin/client/lib/bcprov-jdk18on-1.80.jar	jar	0	-
opt/keycloak/bin/client/lib/keycloak-crypto-default-26.2.5.jar	jar	0	-
opt/keycloak/bin/client/lib/keycloak-crypto-fips1402-26.2.5.jar	jar	0	-
opt/keycloak/lib/app/keycloak.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.github.crac.org-crac-0.1.3.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.quarkus.quarkus-bootstrap-runner-3.20.1.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.quarkus.quarkus-classloader-commons-3.20.1.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.quarkus.quarkus-development-mode-spi-3.20.1.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.quarkus.quarkus-vertx-latebound-mdc-provider-3.20.1.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.smallrye.common.smallrye-common-constraint-2.12.0.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.smallrye.common.smallrye-common-cpu-2.12.0.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.smallrye.common.smallrye-common-expression-2.12.0.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.smallrye.common.smallrye-common-function-2.12.0.jar	jar	0	-
opt/keycloak/lib/lib/boot/io.smallrye.common.smallrye-common-io-2.12.0.jar	jar	0	-

Figura 6.2: npm audit su nodejs.

Alcune vulnerabilità sono state segnalate nei layer sottostanti (Debian/RedHat base), ma queste non risultano attualmente rilevanti in quanto:

- i container sono usati solo all'interno del cluster Kubernetes, non esposti direttamente,
- l'accesso è regolato da policy di rete e ruolo,
- le immagini sono ufficiali, mantenute attivamente e aggiornate di frequente.

6.3 Controllo dei chart Helm

L'applicazione utilizza Helm chart per il deployment di componenti come Cert-Manager, Vault, Vault-CSI, grafana, ecc. È stato verificato che:

- i chart utilizzati provengono da repository ufficiali (bitnami, ingress-nginx, etc.),
- le versioni delle immagini specificate nei `values.yaml` sono aggiornate e prive di vulnerabilità note,
- non sono stati rilevati valori predefiniti insicuri o configurazioni esposte.

Capitolo 7

Use Approved Tools

Questo capitolo documenta le azioni intraprese per assicurarsi che tutti gli strumenti critici, dagli ambienti di sviluppo ai container runtime, siano approvati, tracciabili e sottoposti a controlli di sicurezza.

L'obiettivo di questa fase è duplice: limitare i rischi legati a tool compromessi o malevoli e garantire la trasparenza e la riproducibilità del processo di sviluppo. Per ciascun componente, è stata registrata l'origine (sito ufficiale, registro pubblico, ecc.) e verificata la *firma digitale* o il *checksum*, ove disponibile.

Inoltre, per garantire un controllo continuo, alcuni artefatti critici (come immagini Docker e file di lock delle dipendenze) sono stati firmati digitalmente con *Sigstore Cosign* e i metadati delle firme sono conservati nel repository in una directory tracciata. Sebbene queste firme siano oggi generate manualmente, rappresentano un primo passo concreto verso un processo di validazione più automatizzato secondo le

linee guida *SLSA* (*Supply-chain Levels for Software Artifacts*).

7.1 Ambienti di Sviluppo e IDE

L'ambiente di sviluppo locale rappresenta il primo anello della supply chain software. Per il nostro progetto abbiamo scelto *Visual Studio Code*, installato esclusivamente dal sito ufficiale *Microsoft*. Scaricare il software dal sito ufficiale garantisce che il pacchetto scaricato non sia stato alterato da terze parti, evitando rischi dovuti a versioni distribuite tramite canali non ufficiali.

In aggiunta, sono state eseguite verifiche sulla firma digitale del pacchetto di installazione e dell'eseguibile post-installazione direttamente dal sistema operativo (*Windows*), assicurando che entrambi fossero firmati da Microsoft e corrispondenti ai certificati attesi.

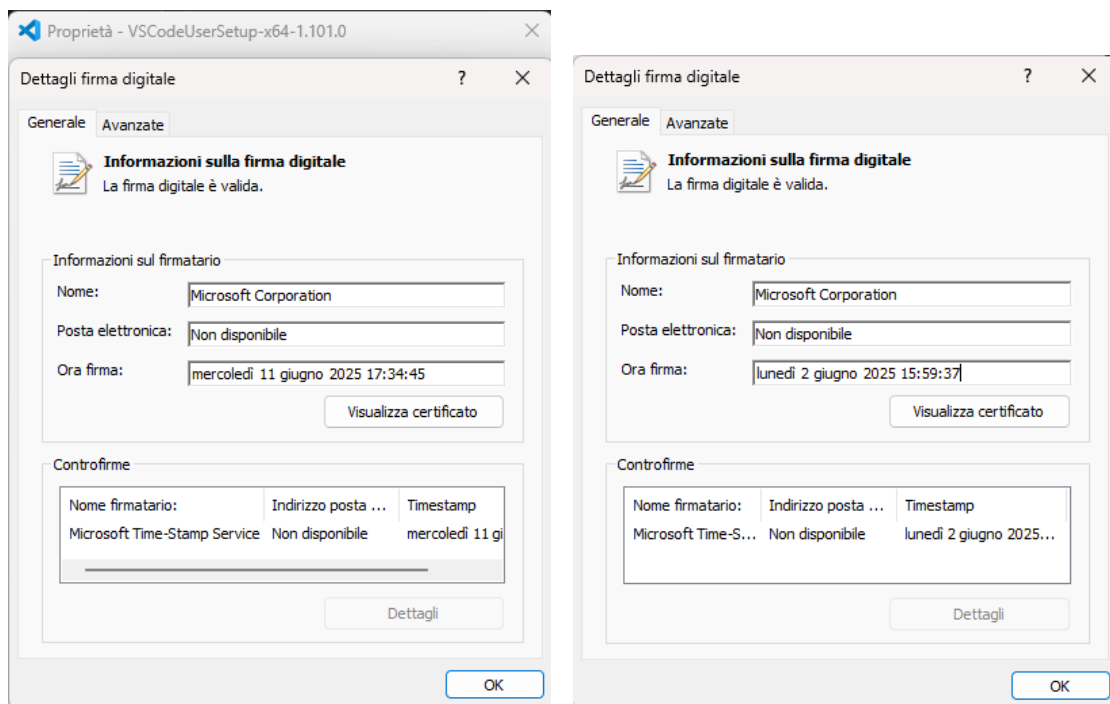


Figura 7.1: Verifica delle firme VSCode.

7.2 Framework, Runtime e Librerie

Il runtime adottato per lo sviluppo del backend è *Node.js*, con una base solida garantita dall'utilizzo dell'immagine ufficiale *node:lts-alpine* pubblicata su *Docker Hub*. Questa scelta consente di partire da un'immagine mantenuta dalla community ufficiale, minimizzando il rischio di vulnerabilità note o artefatti malevoli.

Tutti i framework utilizzati, tra cui *Express.js* per la gestione delle rotte, *Prisma ORM* per l'accesso ai dati, *GraphQL Yoga* per l'esposizione di API moderne e *jsonwebtoken* per la gestione dei token, sono installati tramite *npm*, il registro ufficiale dei pacchetti *Node.js*. Ciò garantisce la provenienza autentica dei componenti e facilita l'applicazione di controlli automatici su eventuali *CVE* (*Common Vulnerabili-*

ties and Exposures).

Per rafforzare ulteriormente la sicurezza della supply chain, è stato firmato il file *package-lock.json* associato al backend. Il file, estratto direttamente dal container in esecuzione nel cluster *Kubernetes*, rappresenta uno snapshot preciso e immutabile dello stato delle dipendenze installate. La firma è stata eseguita manualmente tramite Cosign, uno strumento open-source per la firma degli artefatti, e consente di verificare in ogni momento che il contenuto del lockfile non sia stato alterato.

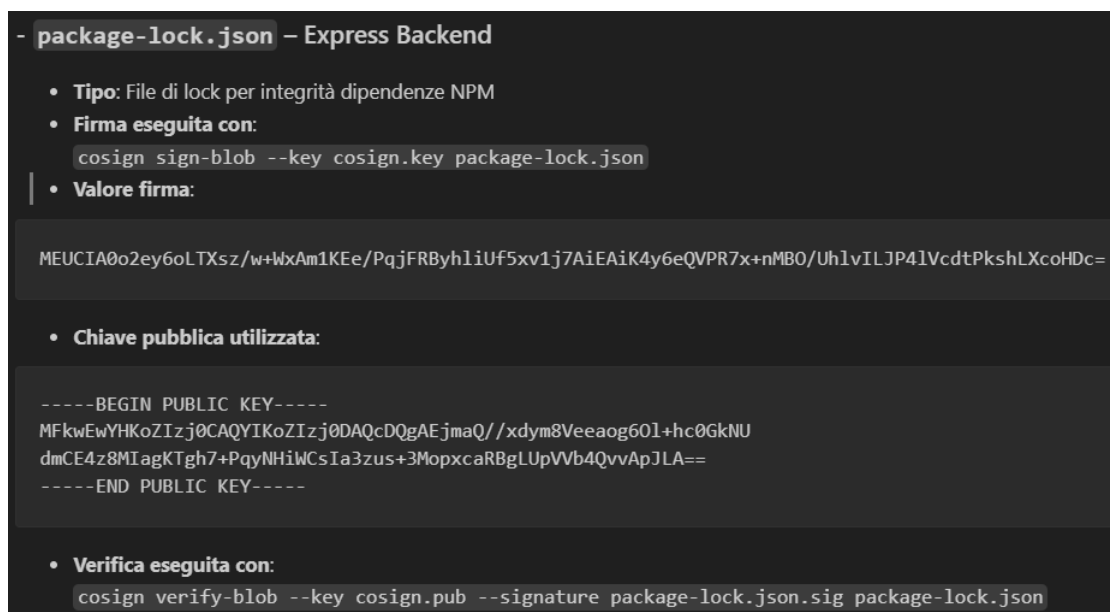


Figura 7.2: Cosign applicato a *package-lock.json*.

7.3 Immagini Docker verificate e firmate

Tutte le immagini *Docker* utilizzate nel progetto, sia quelle custom (come per l'API Express e il servizio Keycloak) sia quelle di terze parti

utilizzate nei *Helm chart*, sono state sottoposte a una verifica accurata della loro provenienza e integrità.

Le immagini personalizzate sono costruite a partire da immagini base ufficiali, come *node:lts-alpine* per il backend, e successivamente estese con il codice sorgente dell'applicazione. Al termine del processo di build, le immagini vengono caricate nel *GitLab Container Registry* del progetto. In questa fase, ogni immagine è firmata manualmente con *Cosign*, in modo da garantirne la paternità e la non alterazione.

La verifica della firma è stata eseguita tramite il comando *cosign verify*, il cui output è stato salvato insieme al digest (sha256) dell'immagine e alla relativa chiave pubblica. Queste informazioni sono conservate nel repository per garantire la tracciabilità nel tempo e permettere la riproducibilità e la validazione futura degli artefatti.

Anche per le immagini di terze parti, come quelle utilizzate da Helm chart ufficiali per *NGINX Ingress Controller*, *Cert Manager* o *PostgreSQL*, è stato verificato l'hash (digest) e la provenienza, assicurandosi che i riferimenti nei manifest siano puntati a immagini stabili e pubblicate da maintainer ufficiali.

```
[
  {
    "critical": {
      "identity": {
        "docker-reference": "registry.gitlab.com/felicemicillo13/ssproject/express"
      },
      "image": {
        "docker-manifest-digest": "sha256:a929d8ba459f5c40bd7e119bf1d4cf78c553671fe5c9e5e63207c8488a22d3e1"
      },
      "type": "cosign container image signature"
    },
    "optional": null
  }
]
```

Figura 7.3: Verifica Cosign sull'immagine Express.

7.4 Conservazione e tracciabilità delle firme

E' stato predisposto un sistema di conservazione strutturata delle firme digitali e dei relativi metadati, in modo da garantire trasparenza e verificabilità nel tempo.

Tutti gli artefatti critici firmati (come le immagini Docker e il file `package-lock.json`) sono accompagnati dai relativi file di firma (`.sig`), dalle chiavi pubbliche utilizzate per la verifica, e dagli output dei comandi *cosign verify*. Tutte le informazioni sono raccolte all'interno di un documento *MANIFEST.md*, redatto manualmente, che funge da inventario della sicurezza della supply chain.

Il file *MANIFEST.md* descrive in modo strutturato:

- le immagini firmate, con digest e comandi di verifica eseguiti,
- i file critici firmati manualmente (es. `package-lock.json`),
- la corrispondente chiave pubblica associata,

- la posizione dei file .sig e degli output di verifica,
- eventuali commenti aggiuntivi legati alla provenienza degli artefatti.

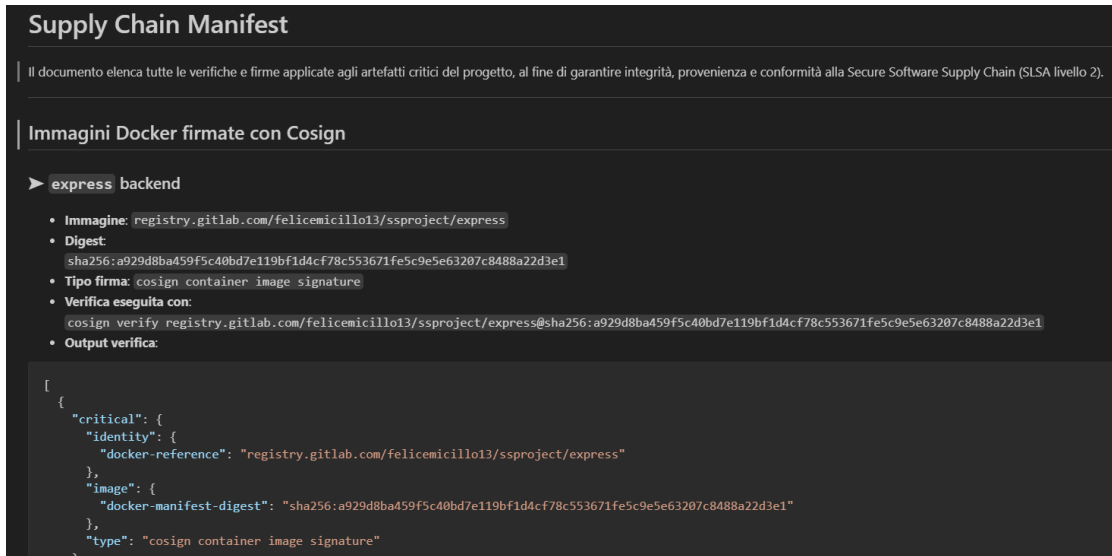


Figura 7.4: Manifest per l'auditing.

Capitolo 8

Perform SAST e DAST (con GitLab)

Abbiamo integrato due tecniche complementari all'interno della nostra CI/CD su GitLab:

- **SAST (Static Application Security Testing)**: analizza il codice sorgente lato backend prima della build per individuare pattern insicuri, errori logici e vulnerabilità comuni come injection, hardcoded secrets o uso scorretto della crittografia.
- **DAST (Dynamic Application Security Testing)**: analizza il comportamento dell'applicazione già in esecuzione, simulando attacchi reali da parte di un utente esterno (black box testing). Ciò permette di individuare problemi di sicurezza che

emergono solo in fase di runtime, come configurazioni errate, assenza di header, o endpoint esposti senza controllo.

L'implementazione di questi controlli automatizzati è stata effettuata sfruttando gli strumenti open-source *ESLint*, *NodeJSSCAN* e *OWASP ZAP*, orchestrati tramite una pipeline definita in *GitLab CI/CD*. Ogni fase produce report dettagliati archiviati come artefatti della build e consultabili per analisi, fixing e tracciabilità.

8.1 Static Application Security Testing (SAST)

Nel nostro progetto, la fase *SAST* è eseguita automaticamente all'interno della pipeline GitLab subito dopo il commit, nella fase iniziale del ciclo DevSecOps. Questa fase ha lo scopo di individuare vulnerabilità direttamente nel codice sorgente dell'API Express, senza necessità di esecuzione dell'applicazione.

Sono stati utilizzati i seguenti strumenti:

- **ESLint**: utilizzato per verificare la qualità del codice e rilevare eventuali violazioni delle best practice JavaScript, comprese regole di sicurezza e vulnerabilità comuni.
- **NodeJSSCAN**: un motore statico specializzato nell'analisi di codice Node.js. Esamina moduli, middleware e routing, identificando rischi come:

- Uso scorretto di eval o exec
- Inclusione di librerie non sicure
- Assenza di input validation
- Weak crypto (es. uso debole di JWT o hash senza salt)
- Hardcoded secrets

```
sast_express:
  stage: sast
  image: python:3.11-alpine
  before_script:
    - apk add --no-cache git nodejs npm
    - pip install nodejsscan
    - npm install -g eslint
  script:
    - cd kubernetes/docker/express
    - eslint . -f json -o "$CI_PROJECT_DIR/eslint_report.json"
    - nodejsscan -d . -o "$CI_PROJECT_DIR/nodejsscan_report.json"
    - |
      python3 -c "
      import json, sys
      from pprint import pprint
      with open('$CI_PROJECT_DIR/nodejsscan_report.json') as f:
        report = json.load(f)
        issues = report.get('sec_issues', {})
        mis_headers = report.get('missing_sec_header', {}).get('Web Security', [])
        relevant_mis_headers = [
          h for h in mis_headers
          if 'Public-Key-Pins' not in h.get('title', '')
        ]
        if issues:
          print('=== Security Issues Detected ===')
          for category, problems in issues.items():
            print(f'- {category}:')
            for item in problems:
              print(f"    • {item.get('title', 'Untitled')} [{item.get('filename', '?')}:{item.get('line', '?')}]")

          if relevant_mis_headers:
            print('=== Missing Security Headers (excluding HPKP) ===')
            for h in relevant_mis_headers:
              print(f"    • {h.get('title')}")
          if issues or relevant_mis_headers:
            print('Security issues found (excluding HPKP): failing job.')
            sys.exit(1)
      "
  artifacts:
    when: always
    paths:
      - eslint_report.json
      - nodejsscan_report.json
  only:
    - web
```

Figura 8.1: Codice Gitlab CI/CD del SAST.

8.2 Dynamic Application Security Testing (DAST)

Oltre all'analisi statica, il progetto prevede una fase *DAST* automatizzata che consente di testare l'applicazione in esecuzione simulando attacchi reali dal punto di vista dell'utente esterno. Per seguire questo test è stato utilizzato il tool *OWASP ZAP*.

Nella nostra pipeline GitLab, il job *dast_scan* utilizza OWASP ZAP (Zed Attack Proxy) nella sua versione containerizzata (ghcr.io/zaproxy/zaproxy:stable). L'esecuzione segue un piano di scansione personalizzato (*zap_config.yaml*) che include:

- Import automatico dell'OpenAPI (swagger) per la mappatura completa delle rotte.
- Script di autenticazione (*auth_script.js*) per accedere a endpoint protetti.
- Simulazioni di attacchi comuni (XSS, SQL Injection, ecc.)
- Verifica delle configurazioni HTTP e delle intestazioni di sicurezza.

Il test esegue le seguenti fasi:

- Avvio del proxy in modalità daemon
- Import dell'OpenAPI definition (*openapi.json*)

- Esecuzione del piano di scansione personalizzato
- Attesa del completamento via polling
- Generazione del report HTML (zap_report.html)

```

dast_scan:
  stage: dast_zap
  image: ghcr.io/zaproxy/zaproxy:stable
  allow_failure: false
  script:
    - mkdir -p /zap/wrk
    - cp zap_config.yaml /zap/wrk/
    - cp auth_script.js /zap/wrk/
    - cp openapi.json /zap/wrk/
    - zap.sh -daemon -port 8090 -host 0.0.0.0 -config api.disablekey=true -config api.addr.name=.* -config api.addr.regex=true -addoninstall authhelper -addoninstall graaljs &
    - sleep 30
    - curl "http://localhost:8090/JSON/openapi/action/importFile?file=/zap/wrk/openapi.json"
    - sleep 10
    - curl -X POST "http://localhost:8090/JSON/automation/action/runPlan/" -d "filePath=/zap/wrk/zap_config.yaml"
    - |
      while true; do
        finished=$(curl -s "http://localhost:8090/JSON/automation/view/planProgress/?planId=0" | jq -r '.finished')
        if [ -n "$finished" ]; then
          echo "Scan completed at: $finished"
          break
        fi
        sleep 2
      done
    - curl "http://localhost:8090/OTHER/core/other/htmlreport/" -o zap_report.html
  artifacts:
    when: always
    paths:
      - zap_report.html
  needs:
    - job: kubescape_scan
  only:
    - web

```

Figura 8.2: Codice Gitlab CI/CD del DAST.

8.3 Raccolta dei report

Al termine delle fasi di *SAST* e *DAST*, la pipeline GitLab esegue una fase di raccolta centralizzata dei report tramite il job *collect_reports*. Il job aggrega automaticamente tutti i file generati durante i test di sicurezza in un unico workspace, facilitando l'analisi e la consultazione dei risultati.

Vengono raccolti i seguenti report:

- *eslint_report.json*: risultati dell'analisi statica sullo stile e sulle best practice del codice.

- `nodejsscan_report.json`: rilevamento di vulnerabilità note e pattern insicuri nella codebase.
- `trivy_report_*.json`: scansioni dei container per vulnerabilità note (HIGH e CRITICAL).
- `kubescape_report.html`: verifica della sicurezza a livello di cluster Kubernetes (NSA framework).
- `zap_report.html`: report HTML generato da OWASP ZAP con i risultati della scansione dinamica.

Un piccolo script Python (*generate_combined_report.py*) combina automaticamente i report più rilevanti (JSON e HTML) in un unico documento finale chiamato *combined_report.html*, facilitando la condivisione e la consultazione da parte del team.

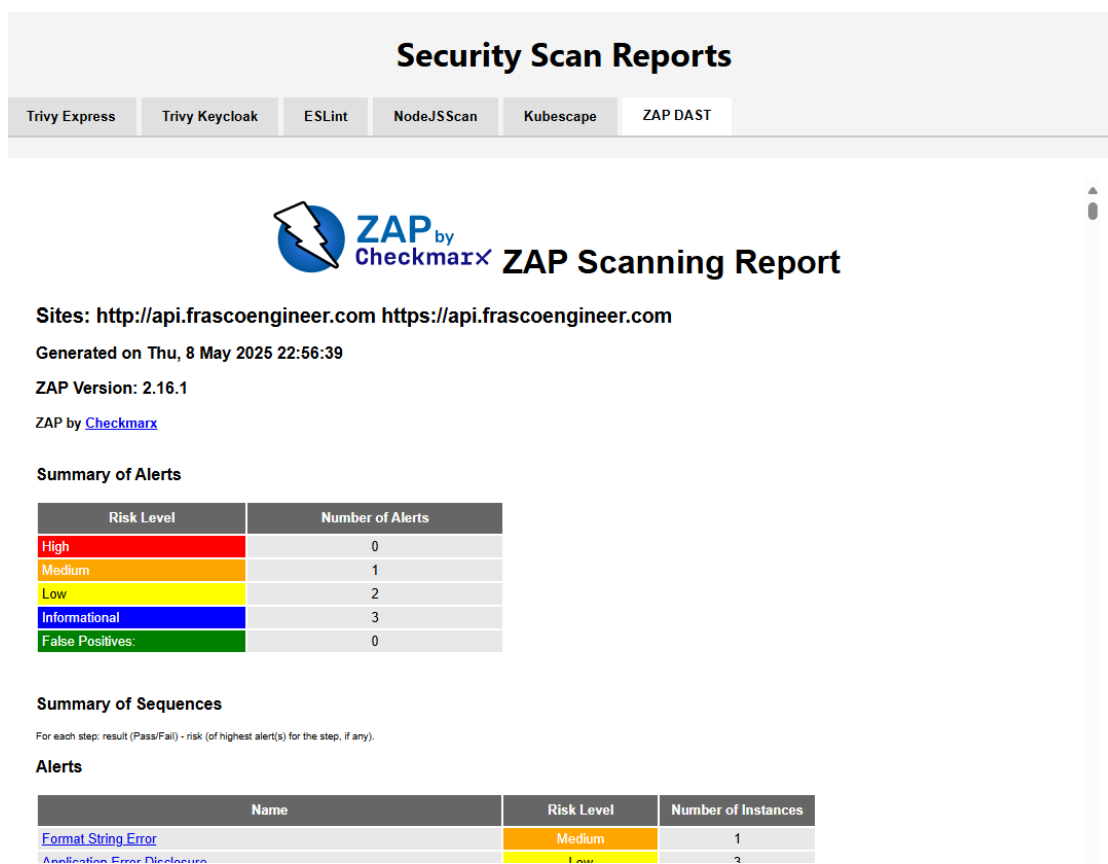


Figura 8.3: Report finale (combined_report.html).