POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Pipeline DevSecOps

Relatore:

Prof. Riccardo Sisto

Candidato:

Luca Antoniotti

ANNO ACCADEMICO 2020/2021

Sommario

DevSecOps è una nuova metodologia di sviluppo software in cui la sicurezza è parte integrante del ciclo di vita delle applicazioni, l'etimologia della parola nasce dall'unione di *Development*, *Security* e *IT Operations*. Con *Development* si intendono le operazioni di creazione del software, il termine *Security* comprende tutti i controlli di sicurezza che vengono eseguiti durante il ciclo di vita dell'applicativo, mentre *IT Operations* racchiude le attività di rilascio, monitoraggio e supporto.
L'obiettivo è rendere i processi più fluidi, rapidi ed efficienti. I vantaggi di questo approccio sono la maggior trasparenza e la possibilità di dare un feedback immediato allo sviluppatore prima che il prodotto venga rilasciato in produzione.
Questa metodologia si basa su due principi fondamentali:

- una rapida collaborazione tra i membri del team: non essendoci una divisione dei ruoli tutti i componenti lavorano allo stesso progetto e si ha una comunicazione più veloce.
- automazione: utilizzo delle tecnologie per ridurre le attività manuali, accelerare i processi ed espandere gli ambienti. Infatti, grazie alla possibilità di automatizzare una serie di processi ripetitivi, la qualità dei prodotti migliora notevolmente e si riduce il rischio di errore dovuto alle attività manuali.

Il lavoro di ricerca ha l'obiettivo di approfondire le tematiche DevSecOps facendo particolare riferimento all'esecuzione automatizzata di specifiche analisi di sicurezza sugli applicativi. Si vuole unire il mondo dello sviluppo software con quello della sicurezza, permettendo al team di coordinarsi in maniera agile ed efficiente.
Al giorno d'oggi gli specialisti di security eseguono manualmente gran parte dei controlli di sicurezza nella fase finale dello sviluppo. Tuttavia, si sta cercando di includere questi controlli anche nelle fasi precedenti.

Per questo lavoro di ricerca si utilizza GitLab, una piattaforma web che ospita il codice sorgente e automatizza una sequenza di script, chiamata pipeline, per eseguire la build, i test di sicurezza e il deploy. E' stato scelto GitLab perché viene già adottato dalle aziende quindi non è necessario installare strumenti esterni

aggiuntivi; inoltre è semplice, ha dei container registry, ovvero registri privati e sicuri che contengono le immagini Docker, ed è integrabile con Kubernetes.
Kubernetes è un software open-source per l'automazione del deployment in un ambiente di produzione e gestione di applicativi in containers. Per questo lavoro di ricerca è stato utilizzato minikube, una distribuzione più snella di Kubernetes che permette di eseguire il deploy di un applicativo all'interno di in un cluster locale.
La pipeline, per funzionare correttamente, richiede la presenza del file *.gitlab-ci.yml* dove sono definiti i vari passaggi e gli script da eseguire per ogni fase.
Per fare l'analisi statica si integra nella pipeline AppScan Source che permette di scansionare il codice sorgente di un'applicazione e al termine produce un report contenente le vulnerabilità trovate. Per funzionare correttamente AppScan Source è connesso ad AppScan Enterprise, un server installato in cloud sulla piattaforma Amazon Web Service (AWS). Quest'ultimo raccoglie tutte le vulnerabilità e il relativo grado di criticità mostrandoli all'utente, il quale può dare loro delle priorità e decidere quale correggere prima.
Ulteriori strumenti di sicurezza sono stati integrati nella pipeline: kube-bench, kube-hunter, kubeaudit e KubeSec. Questi applicativi scansionano tutto l'ecosistema Kubernetes utilizzato e verificano la presenza di eventuali vulnerabilità o pratiche insicure producendo dei report. Per poter caricare sul server le vulnerabilità trovate dalle analisi dell'ambiente Kubernetes, è necessario che i report generati abbiano un formato accettabile da AppScan Enterprise. Pertanto, è stato creato uno script Python che crea un file CSV conforme alle richieste e lo invia al server di monitoraggio.

Il risultato finale del lavoro di ricerca è una pipeline DevSecOps completamente automatizzata. Ad ogni modifica del codice sorgente sul repository, salvo diverse indicazioni, la pipeline inizia e viene eseguita la build, l'analisi statica del codice, il deploy dell'applicativo e i controlli sull'ambiente Kubernetes. Utilizzando un'applicazione vulnerabile di circa 35000 righe di codice sono state riportate più di 1200 vulnerabilità di cui 564 con severità elevata.
La pipeline viene eseguita in totale autonomia impiegando 7 minuti per completare tutto il processo e permettendo così al team di sviluppo di ricevere i feedback in tempi brevi senza doversi coordinare con altre persone.
Un altro vantaggio di questa soluzione è l'alta integrabilità: è possibile aggiungere alla pipeline numerosi componenti a seconda delle esigenze del cliente e delle politiche aziendali.

Indice

Elenco delle figure 6

1 Introduzione 8
- 1.1 Contesto . . . 8
- 1.2 Obiettivo della tesi . . . 9
- 1.3 Capitoli successivi . . . 9

2 Analisi del dominio di riferimento 11
- 2.1 DevOps . . . 11
- 2.2 DevSecOps . . . 12
  - 2.2.1 CI/CD . . . 12
  - 2.2.2 Pipeline DevSecOps . . . 14
  - 2.2.3 Fasi DevSecOps . . . 14
- 2.3 Tecnologie DevSecOps . . . 16
  - 2.3.1 Jenkins . . . 16
  - 2.3.2 GitHub . . . 17
  - 2.3.3 Gitlab . . . 17
- 2.4 Limitazioni dei principali ambienti . . . 17
  - 2.4.1 Jenkins . . . 17
  - 2.4.2 GitHub . . . 17
- 2.5 Una scelta vantaggiosa: GitLab . . . 18
- 2.6 Kubernetes . . . 19
- 2.7 Analisi Statica: HCL AppScan . . . 19

3 Metodologia per lo sviluppo DevSecOps 21
- 3.1 Architettura . . . 22
- 3.2 GitLab: scelte implementative . . . 23
  - 3.2.1 Certificati SSL . . . 24
  - 3.2.2 GitLab Runner . . . 24
  - 3.2.3 Minikube . . . 26
- 3.3 GitLab: fasi della pipeline . . . 27

3.3.1 Build 28
3.3.2 Test 29
3.3.3 Deploy 31

4 Risorse utilizzate 40
4.1 Specifiche tecniche delle macchine 40
4.2 Requisiti delle applicazioni 41
4.2.1 GitLab 41
4.2.2 HCL AppScan Source 42
4.2.3 HCL AppScan Enterprise 43
4.2.4 Minikube 44

5 Analisi dei risultati 46
5.1 Damn Vulnerable Web Application 46
5.2 Differenze con lo sviluppo tradizionale 47
5.3 Vulnerabilità trovate 49

6 Limitazioni e lavori futuri 53
6.1 Limitazioni 53
6.2 Lavori futuri 54

7 Conclusioni 56

A Manifest 58
A.1 Deployment.yml 58
A.2 Service.yml 59
A.3 Ingress.yml 60
A.4 kube-bench.yaml 60
A.5 kube-hunter.yaml 62
A.6 kubeaudit.yaml 63

Bibliografia 65

Elenco delle figure

2.1 CI/CD 13
2.2 Fasi DevSecOps 14

3.1 Architettura 22
3.2 gitlab.rb 23
3.3 Pipeline 27
3.4 Variabili e stages 28
3.5 Build 28
3.6 Test 29
3.7 Run_Assessment 30
3.8 Deploy 32
3.9 Test sull'infrastruttura Kubernetes 34
3.10 parser_and_uploader.py 38

4.1 Topologia AppScan 44

5.1 Environments 47
5.2 Homepage dvwa 47
5.3 Pipeline terminata di GitLab 48
5.4 Output di AppScan Source 49
5.5 AppScan Enterprise 50
5.6 Vulnerabilità con criticità elevata 50
5.7 Vista di una singola vulnerabilità 51
5.8 Vista Information 51

A.1 Deployment.yml 59
A.2 Service.yml 59
A.3 Ingress.yml 60
A.4 kube-bench.yaml 62
A.5 kube-hunter.yaml 63
A.6 kubeaudit.yaml 64

Capitolo 1

Introduzione

1.1 Contesto

Al giorno d'oggi molte aziende utilizzano delle metodologie inefficienti che si basano sui processi manuali per sviluppare nuovi applicativi. Le alte aspettative dei clienti finali e l'evoluzione delle tecnologie hanno aumentato la necessità di realizzare nuove tecniche di sviluppo software che abbiano la capacità di minimizzare i possibili danni derivati da errori di natura umana.
In questo scenario nasce DevOps [1], un nuovo approccio che mira ad aumentare la collaborazione tra la parte di *Development* e quella di *Operations* gestendo il monitoraggio, le prestazioni, l'infrastruttura e la distribuzione del software.
DevOps utilizza elementi tecnologici e infrastrutturali che possono essere facilmente controllabili tramite il codice. Inoltre l'automazione permette ai nuovi applicativi di essere rilasciati più frequentemente e, per ognuno di essi, si utilizzano sempre le stesse procedure.
Con il DevOps, i cicli di sviluppo software sono più brevi e veloci ma è necessario prendere in considerazione anche la sicurezza, ramo dell'informatica spesso sottovalutato dagli stessi elementi del team o lasciato in disparte perché considerato come passaggio saltuario all'interno di una catena di sviluppo. Infatti, la visione tradizionale, prevede che le vulnerabilità possano essere trovate una volta terminato lo sviluppo prima che il prodotto venga rilasciato al cliente.
In questo contesto nasce il DevSecOps [2] il cui obiettivo è quello di creare una mentalità secondo la quale ognuno è responsabile della sicurezza e deve essere inclusa in tutte le fasi dello sviluppo di un nuovo applicativo.
Il DevSecOps, basandosi sul concetto di automazione, si propone di integrare al suo interno una serie di controlli di sicurezza in modo da ottenere un prodotto più sicuro senza aumentare i tempi di produzione.

1.2 Obiettivo della tesi

All'interno dello scenario precedentemente descritto si colloca il lavoro di tesi svolto: l'obiettivo è quello di creare una pipeline DevSecOps che permetta di automatizzare e migliorare i controlli di sicurezza durante lo sviluppo di un applicativo, rilevando il prima possibile le vulnerabilità presenti nell'applicativo e nella configurazione dell'infrastruttura sulla quale viene rilasciato.
E' stato progettato e implementato un ambiente DevSecOps che sia in grado di fare build, check e deploy di un'applicazione web basata su Kubernetes. GitLab è stato usato sia come gestore del repository sia come CI/CD orchestrator; i controlli di sicurezza sono stati eseguiti da *HCL AppScan Source for Automation* e altri specifici tools per le applicazioni basate su Kubernetes come *kube-bench, kube-hunter, kubeaudit* e *KubeSec.*
Grazie a questa soluzione sono state rilevate 958 vulnerabilità che sono visualizzabili all'interno di un'unica *dashboard* centralizzata fornita da AppScan Enterprise.

1.3 Capitoli successivi

Dopo aver presentato il contesto in cui nasce il lavoro di tesi e i suoi obiettivi, i capitoli successivi sono strutturati nel seguente modo:

- **Capitolo 2**: presenta e approfondisce i concetti di DevOps e DevSecOps; inoltre, sono presentati i principali strumenti impiegati al giorno d'oggi dalle aziende e i rispettivi limiti. Infine si introducono i tool utilizzati all'interno del lavoro di tesi.

- **Capitolo 3**: descrive l'architettura realizzata e la metodologia seguita, presentando nel dettaglio la pipeline e le scelte implementative prese.

- **Capitolo 4**: contiene le specifiche tecniche delle macchine utilizzate e i requisiti hardware e software dei singoli programmi installati.

- **Capitolo 5**: descrive l'applicativo analizzato e presenta i risultati ottenuti in termini di tempo e di vulnerabilità trovate.

- **Capitolo 6**: contiene le limitazioni riscontrate durante la fase di realizzazione della soluzione e presenta alcuni esempi di possibili sviluppi futuri per migliorare il lavoro.

- **Capitolo 7**: riassume gli obiettivi del lavoro di tesi e i risultati ottenuti dalla soluzione implementata.

- **Appendice A**: racchiude i manifest utilizzati per eseguire il rilascio del codice all'interno dell'ambiente di produzione e per comunicare correttamente con l'istanza creata; inoltre sono presenti i file YAML che eseguono le analisi sull'ambiente Kubernetes.

Capitolo 2

Analisi del dominio di riferimento

2.1 DevOps

DevOps [3] è una metodologia di sviluppo software che ha l'obiettivo di diminuire il tempo necessario per realizzare una nuova versione di un applicativo.
DevOps nasce dall'unione di due parole ovvero *Development* e *IT Operations*. Con *Development* [4] si intendono le operazioni di creazione del software, mentre *IT Operations* [5] racchiude le attività di rilascio dell'applicativo, monitoraggio e supporto.

Questo approccio si basa sui concetti di automazione e collaborazione.
L'automazione consiste nell'utilizzare le tecnologie in modo da ridurre le attività manuali garantendo così un aumento dell'efficienza nell'esecuzione di processi automatizzabili. Infatti, grazie alla possibilità di automatizzare una serie di processi ripetitivi, la qualità dei prodotti migliora notevolmente e si riduce il rischio di errore dovuto alle attività manuali.
Un aspetto fondamentale della metodologia DevOps è l'aumento di comunicazione e collaborazione all'interno dell'azienda. L'utilizzo di strumenti DevOps e l'automazione dei processi di rilascio software consentono una maggiore collaborazione mediante l'unione dei flussi di lavoro. La comunicazione tra sviluppatori, produzione e altri team aziendali (come marketing e vendite) è più efficiente, in questo modo l'intera organizzazione può allinearsi maggiormente ai propri obiettivi [6].
DevOps [7] permette a ruoli in precedenza isolati, tra cui sviluppo, operazioni IT e controllo della qualità, di coordinarsi e collaborare per fornire prodotti migliori e più affidabili. Grazie all'adozione di una cultura DevOps, i team possono rispondere meglio alle esigenze dei clienti.

2.2 DevSecOps

Per sfruttare tutti i benefici di un approccio DevOps occorre tenere conto anche di un altro elemento molto importante per le applicazioni: i controlli di natura statica, come l'analisi del codice e delle dipendenze, o di natura dinamica, come l'analisi dell'ambiente di produzione.
In passato, i controlli di sicurezza venivano eseguiti manualmente e soltanto nella fase finale dello sviluppo da un team specifico. Quando i cicli di sviluppo duravano mesi o persino anni, la sicurezza non aveva la stessa importanza che ha oggi. Una metodologia DevOps efficace garantisce cicli di sviluppo rapidi e frequenti (di settimane o giorni), ma anche le iniziative DevOps più efficienti possono fallire se le pratiche di sicurezza adottate sono eseguite occasionalmente.
La maggior parte dei problemi che mettono a rischio le applicazioni ha origine nella progettazione iniziale, il che rende difficoltoso rilevarli e mitigarli nelle fasi successive.
Al giorno d'oggi la sicurezza ha assunto un ruolo centrale nello sviluppo software al punto da coniare una nuova parola: "*DevSecOps*" [8].
DevSecOps si propone di integrare la sicurezza degli applicativi e delle infrastrutture già dalle prime fasi dello sviluppo, includendo controlli automatici di natura statica e dinamica, in questo modo si ottengono dei processi più lineari ed efficienti risparmiando tempo e risorse. Se la sicurezza viene presa in considerazione mentre il processo di sviluppo è in corso non si subiscono rallentamenti, poiché si trae vantaggio dalle diverse soluzioni di monitoraggio e automazione. Inoltre, i team operativi e di sviluppo sono direttamente coinvolti nel processo; pertanto, si riduce il distacco con gli esperti di sicurezza evitando sin da subito eventuali falle. Ciò significa che le versioni software sicure e stabili vengono prodotte in tempi più brevi.
Come nel DevOps, anche nel DevSecOps il successo di questa metodologia e la sua efficienza dipendono da quanto i team sostengono questo nuovo sviluppo, per questo motivo è importante comunicare apertamente i vantaggi del nuovo sistema [9].
Sia il DevOps che il DevSecOps si basano sul concetto di CI/CD.

2.2.1 CI/CD

CI/CD è un metodo per la distribuzione frequente degli applicativi che integra al suo interno il concetto di automazione. L'acronimo CI/CD [10] indica *Continuous Integration* (CI), *Continuous Delivery and Deployment* (CD).

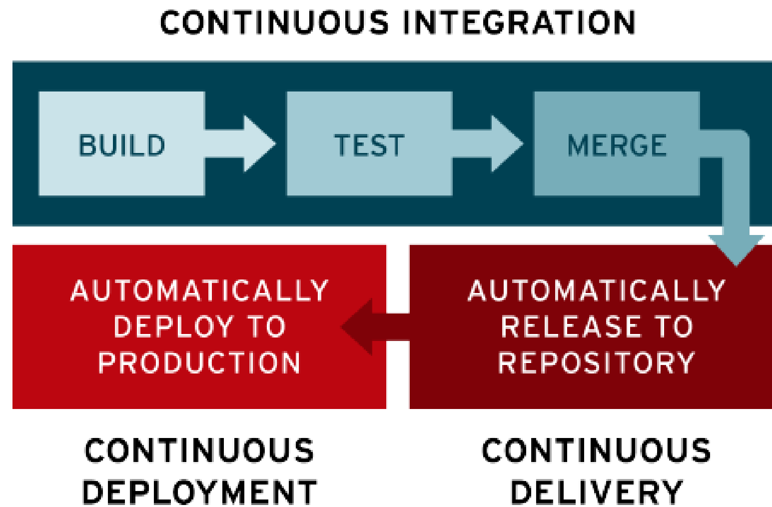

Figura 2.1: CI/CD

- **Continuous Integration**: tutti gli sviluppatori caricano le proprie modifiche del codice all'interno di un repository centrale condiviso. Ogni cambiamento effettuato innesca automaticamente una build (ovvero la compilazione del software che genera un file eseguibile) e una sequenza di test che verifica le funzionalità dell'applicativo.

- **Continuous Delivery**: è composto da diverse fasi che preparano e monitorano automaticamente un rilascio in produzione. Tuttavia, è necessario includere almeno una fase manuale di approvazione e avvio del deploy in un ambiente di produzione.

- **Continuous Deployment**: è l'evoluzione del Continuous Delivery, qui i cambiamenti effettuati al codice sono automaticamente rilasciati in un ambiente di produzione, senza eseguire attività manuali.

L'obiettivo del CI/CD è quello di rendere lo sviluppo software e il processo di rilascio più veloce e robusto. In questo modo le modifiche apportate al codice diventano attive nel minor tempo possibile.
Questa nuova prassi supera i problemi relativi all'integrazione di nuovo codice. In particolare, l'approccio CI/CD introduce l'automazione costante e il monitoraggio continuo durante tutto il ciclo di vita delle applicazioni, dalle fasi di integrazione e test a quelle di distribuzione e deployment [11].

2.2.2 Pipeline DevSecOps

Le pipeline DevSecOps (o pipeline CI/CD) sono una sequenza di script eseguiti in successione, al fine di realizzare una nuova versione del software. Si introduce il monitoraggio e l'automazione per migliorare lo sviluppo, il testing e il rilascio dell'applicativo. Ogni passaggio potrebbe essere eseguito manualmente, ma integrandolo all'interno di una pipeline si ottiene un processo più veloce ed efficiente. Inoltre automatizzando le attività le si rende più affidabili e il rischio di riscontrare errori diminuisce. In questo modo il deploy diventa ripetibile e più frequente evitando la fase di preparazione iniziale.
I passaggi che compongono una pipeline CI/CD [12] sono sottoinsieme di task raggruppate in quella che è nota come fase della pipeline. Le fasi tipiche della pipeline includono:

- **Build**: si compila l'applicazione e si testano le funzionalità
- **Test**: in questa fase si eseguono i test di sicurezza. Qui l'automazione permette di risparmiare molto tempo e risorse
- **Rilascio**: l'applicazione si rilascia al repository ed è pronta per essere mandata in produzione
- **Deploy**: in questa fase il codice viene mandato in produzione

Un altro vantaggio di questa soluzione, oltre all'automazione e al risparmio di tempo, è l'integrabilità; infatti, è possibile aggiungere alla pipeline numerosi componenti a seconda delle esigenze del cliente e delle politiche aziendali.

2.2.3 Fasi DevSecOps

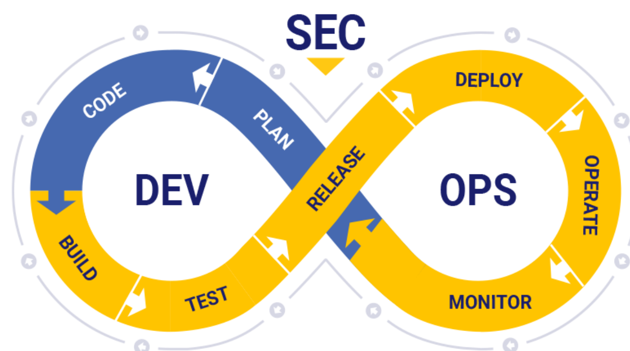

Figura 2.2: Fasi DevSecOps

Il processo DevSecOps è composto da 8 fasi [13] come illustrato nella figura 2.2:

- **Pianificazione (Plan)**: è la fase iniziale dove si raccolgono i requisiti da analizzare per poter scegliere la strategia migliore da applicare al rilascio della catena di DevSecOps.
- **Code**: include tutti gli aspetti della codifica, tra cui scrittura, test, revisione e integrazione del codice da parte dei membri del team.
- **Build**: una volta terminata la scrittura, lo sviluppatore carica il codice nel repository attraverso un'operazione chiamata *push*. Questo avvia la pipeline, viene eseguita la compilazione del codice e una serie di test per verificare le funzionalità del nuovo software.
- **Test**: qui si eseguono una serie di test automatici che hanno l'obiettivo di testare la sicurezza dell'applicativo.
- **Release**: è il punto in cui il risultato della build è pronto per essere distribuito in un ambiente di produzione.
- **Deploy**: il software viene rilasciato in un ambiente di produzione in modo coerente e affidabile
- **Operate**: arrivati a questo punto il cliente può utilizzare l'applicativo e fornire dei feedback sul prodotto.
- **Monitor**: la fase finale del ciclo DevSecOps è il monitoraggio. Attraverso i feedback del cliente e l'analisi del flusso di dati si può cercare di migliorare ulteriormente il prodotto o risolvere eventuali problemi. Pertanto il ciclo riparte dalla prima fase, ovvero la pianificazione.

All'interno di ogni fase del DevSecOps si possono effettuare ulteriori test di sicurezza tra cui:

- **Threat Modeling** [14]: ha l'obiettivo di identificare i requisiti di sicurezza da soddisfare, individuare eventuali minacce e le potenziali vulnerabilità e dare priorità ai metodi di riparazione. Questo processo consente una comprensione più profonda del sistema e delle implicazioni sulla sicurezza dovuta alle decisioni di progettazione. In generale, le aziende eseguono il *threat modeling* durante la fase di progettazione.
- **Analisi Statica**: ricerca le vulnerabilità presenti nel codice senza eseguirlo. I problemi che può identificare sono degli errori di programmazione, valori non definiti, errori di sintassi o best practice di scrittura del codice non rispettate.

- **Penetration Testing** [15]: prevede un'analisi attiva e passiva del sistema per individuare eventuali punti deboli, difetti tecnici e vulnerabilità. L'obiettivo è evitare che un attaccante, o una instabilità del sistema, possano impattare sulla confidenzialità, integrità e disponibilità delle risorse. Questa tipologia di analisi valuta la sicurezza di una applicazione web, di conseguenza i test riguardano tutto il sistema informatico di una organizzazione. Ad esempio si possono testare le funzionalità dell'applicazione, i meccanismi di autenticazione, l'interazione con i database, la configurazione del server e tutti gli elementi che lo circondano nella rete.

- **Security Audit**: sono dei controlli periodici che valutano il livello di rischio di un sistema o di un'applicazione. Ha l'obiettivo evitare potenziali attacchi, consentendo di individuare i sistemi più esposti ai pericoli in modo da permettere alle aziende di adottare le contromisure adeguate.

Qualora gli strumenti lo permettano, questi controlli possono essere integrati all'interno della pipeline ed eseguiti automaticamente.

2.3 Tecnologie DevSecOps

Per sfruttare al meglio la CI è fondamentale adottare un approccio disciplinato e processi efficienti. Ambienti sviluppati appositamente possono facilitare l'intero processo. I software per la CI offrono supporto nella creazione di repository, nel building e testing, ma anche nel controllo delle versioni.
Gli strumenti di CI, tuttavia, non si distinguono unicamente per le funzionalità che offrono, ma presentano notevoli differenze anche in termini di prezzo e licenza. Anche se molti dei software sono open source e dunque gratuiti, alcuni prodotti offrono strumenti a pagamento [16].

2.3.1 Jenkins

Jenkins [17] è un server open source di CI/CD, scritto interamente in Java, che ha l'obiettivo di automatizzare il deployment.
Jenkins ha un ampio set di funzionalità che permettono di automatizzare tutte le fasi del ciclo di vita del software, dalla build al deploy. Il comportamento base di Jenkins è estendibile grazie alla presenza di numerosi plugin, garantendo la compatibilità con tutti i tools necessari nella catena CI/CD [18]. Questo strumento è facilmente installabile e utilizzabile dal sito web ed è gratuito. Inoltre è compatibile con tutti i principali sistemi operativi macOS, Unix e Windows.

2.3.2 GitHub

GitHub [19] nasce come servizio di hosting per progetti software e controllo versione tramite Git [20].
E' possibile utilizzare GitHub come strumento di CI/CD integrandolo con GitHub Actions il quale facilita l'automazione del flusso di lavoro. Si possono eseguire una serie di script per compilare, testare e distribuire un applicativo presente nel repository GitHub. Ogni flusso di lavoro è formato da script eseguiti dopo uno specifico evento [21].

2.3.3 Gitlab

GitLab [22] è una piattaforma web che utilizza il controllo di versione Git e può essere usato sia come repository che come strumento per il CI/CD. Al suo interno sono presenti numerosi servizi utilizzabili per coprire l'intero ciclo di sviluppo del software, dall'analisi al rilascio.
Il modulo CI/CD integrato in GitLab ha avuto negli ultimi anni un rapido tasso di adozione grazie alla sua relativa semplicità d'uso ed all'alto grado di personalizzazione offerto.

2.4 Limitazioni dei principali ambienti

2.4.1 Jenkins

Jenkins utilizza diversi plugins per estendere le sue funzionalità lasciando all'utente finale la decisione di aggiungere o meno un determinato strumento. La limitazione più significativa è data dalla mancanza di un supporto nativo per le attività di base, come ad esempio Docker o GitHub, ma bisogna installare un plugin apposito. Jenkins è nato prima di Docker e dei microservizi, pertanto, non offre un buon supporto per sviluppare le applicazioni di nuova generazione.
Jenkins nasce come server CI, per realizzare anche il CD si richiede la presenza di altri strumenti che possano automatizzare il rilascio del software. Il problema, oltre a dover spendere del tempo all'inizio per configurare tutti gli strumenti necessari, è che questi plugins sono scritti da terze parti, in vari linguaggi e con caratteristiche diverse [23].

2.4.2 GitHub

Su GitHub non esiste un orchestratore di pipeline ovvero uno strumento che consente di richiamare al momento giusto le varie attività automatizzate; infatti, non è possibile eseguire contemporaneamente delle pipeline che hanno dipendenze

le une dalle altre; pertanto, il completamento richiede più tempo.
Inizialmente bisogna spendere del tempo per settare la pipeline perché non esistono delle preconfigurazioni.
Con GitHub non è possibile rieseguire un singolo job, ad esempio un test fallito, ma bisogna rilanciare tutta la pipeline, generando così una diminuzione di produttività e un aumento dei tempi di rilascio.
Anche con questo strumento Docker non è integrato nativamente ma bisogna installare un plugin sviluppato da terze parti; inoltre, anche per i controlli e gli scan di sicurezza sono richiesti software addizionali [24].

2.5 Una scelta vantaggiosa: GitLab

A differenza di altre soluzioni, GitLab si adatta facilmente a qualunque progetto, indipendentemente dalla sua tipologia, dall'architettura o dai linguaggi con cui è sviluppato.
GitLab automatizza le operazioni di CI e CD mediante la pipeline che viene eseguita ogni qualvolta uno sviluppatore esegue delle modifiche al codice. L'esecuzione di un job, ovvero uno script, nella pipeline avviene solo se quello precedente è terminato con successo. In caso contrario, il processo si interrompe e GitLab notifica allo sviluppatore che c'è stato un errore; inoltre è possibile eseguire più pipeline utilizzando un approccio speculativo.
La descrizione e la personalizzazione delle pipeline viene effettuata direttamente sul repository, creando un file YAML opportuno, chiamato *.gitlab-ci.yml* . L'aggiunta di un file con questo nome attiva automaticamente il modulo CI/CD di GitLab per quel repository [25]. GitLab offre la possibilità di abilitare l'*Auto DevOps* [26] che imposta autonomamente la pipeline, eliminando la fase di configurazione iniziale. Con questa funzionalità si rileva il linguaggio del codice, si esegue la fase di build e test, si effettua la misurazione della qualità del codice, lo si analizza cercando le vulnerabilità e infine l'applicazione viene rilasciata in un ambiente di produzione.
GitLab racchiude al suo interno tutti gli strumenti necessari per le operazioni di *Development* e *IT Operations*, semplificandone la complessità e velocizzandone i processi.
GitLab supporta nativamente dei *container registry*, ovvero registri privati e sicuri utilizzati per contenere le immagini Docker, e l'integrazione con Kubernetes, in questo modo è più facile lavorare con le applicazioni basate sui container e con lo sviluppo cloud [27].

2.6 Kubernetes

Kubernetes (K8s) [28] è un software open-source per l'automazione del deployment in un ambiente di produzione, scalabilità, e gestione di applicativi in containers [29]. Sempre più applicazioni utilizzano la tecnologia basata sui contenitori, quindi ad oggi Kubernetes risulta la scelta migliore del settore per l'orchestrazione su larga scala [7].
I container sono simili alle macchine virtuali, ma presentano un modello di isolamento più leggero, condividendo il sistema operativo (OS) tra le applicazioni. Le container image vengono prodotte al momento della compilazione dell'applicativo. La piattaforma vanta un grande ecosistema in rapida crescita [30]. Consente di gestire con semplicità cluster (insieme di nodi) all'interno dei quali vengono eseguite le applicazioni containerizzate. Kubernetes offre le possibilità di orchestrare e gestire la distribuzione dei container, in modo scalabile, al fine di coordinare i carichi di lavoro [29].
Kubernetes può esporre un container usando un nome DNS o il suo indirizzo IP. Se il traffico verso un container è alto, Kubernetes è in grado di distribuirlo su più container in modo che il servizio rimanga stabile [30].
Gli elementi principali sono [29] master, nodi e pod:

- **Master**: la macchina che controlla i nodi Kubernetes. È il punto di origine di tutte le attività assegnate, mantiene lo stato desiderato del cluster e gestisce l'esecuzione delle applicazioni.

- **Nodi**: queste macchine eseguono le attività assegnate richieste. Sono controllate dal nodo master.

- **Pod**: un insieme di uno o più contenitori distribuiti su un singolo nodo che condividono indirizzo IP, nome host ed altre risorse.

Kubernetes [31] utilizza delle API per descrivere lo stato desiderato del cluster, ovvero quali applicazioni eseguire, le immagini da utilizzare e il numero di repliche da istanziare. Le API Kubernetes sono richiamate dalla riga di comando, tramite il comando *kubectl*.

2.7 Analisi Statica: HCL AppScan

HCL AppScan [32] è un insieme di strumenti di monitoraggio e test per la sicurezza delle applicazioni. Ha lo scopo di testare gli applicativi web e quelli locali durante il processo di sviluppo.
Esistono diverse edizioni di questo prodotto:

- **AppScan on Cloud** [33]: contiene una raccolta di strumenti di test di sicurezza, rileva le vulnerabilità e ne facilita la correzione. Sono presenti una serie di funzionalità di gestione che consentono al team di monitorare lo stato di sicurezza della loro applicazione e mantenere la conformità con i requisiti.
- **AppScan Enterprise** [34]: offre un testing di sicurezza scalabile delle applicazioni e funzionalità per gestire i rischi. Sono presenti delle dashboard di gestione che aiutano a classificare e dare priorità alle vulnerabilità identificando quelle più critiche e ad alto rischio per l'azienda. L'interfaccia REST di AppScan Enterprise permette l'integrazione con vari strumenti di DevOps.
- **AppScan Standard** [35]: utilizza un motore di scansione per analizzare un applicativo e testarne le vulnerabilità. I risultati del test sono prioritari, in questo modo si permette, all'operatore, di risolvere velocemente le problematiche e analizzare a fondo le vulnerabilità critiche trovate.
- **AppScan Source** [36]: aiuta a sviluppare un software più sicuro ed evitare le vulnerabilità che emergono in una fase avanzata del ciclo di vita dell'applicativo. I test di sicurezza, se integrati nelle prime fasi del ciclo di sviluppo, riducono l'esposizione al rischio e i costi di riparazione. AppScan Source utilizza la tecnologia Intelligent Finding Analytics (IFA) basata sull'apprendimento automatico per aiutare a identificare rapidamente le vulnerabilità critiche.

Capitolo 3

Metodologia per lo sviluppo DevSecOps

L'obiettivo del lavoro di tesi è quello di integrare, all'interno di una pipeline GitLab, i test di sicurezza come l'analisi statica del codice e delle dipendenze o l'analisi dell'ambiente di produzione Kubernetes. Automatizzando questi controlli, oltre a proteggere l'applicativo dagli attacchi, si evita un rallentamento del flusso di lavoro DevSecOps. Per sviluppare questa architettura sono necessarie delle configurazioni realizzate su misura dato che GitLab non consente un'integrazione nativa con gli strumenti di sicurezza utilizzati, come ad esempio AppScan. Inoltre la mancanza di una documentazione ben dettagliata e l'utilizzo di protocolli proprietari hanno reso più complesso il raggiungimento dell'obiettivo.

3.1 Architettura

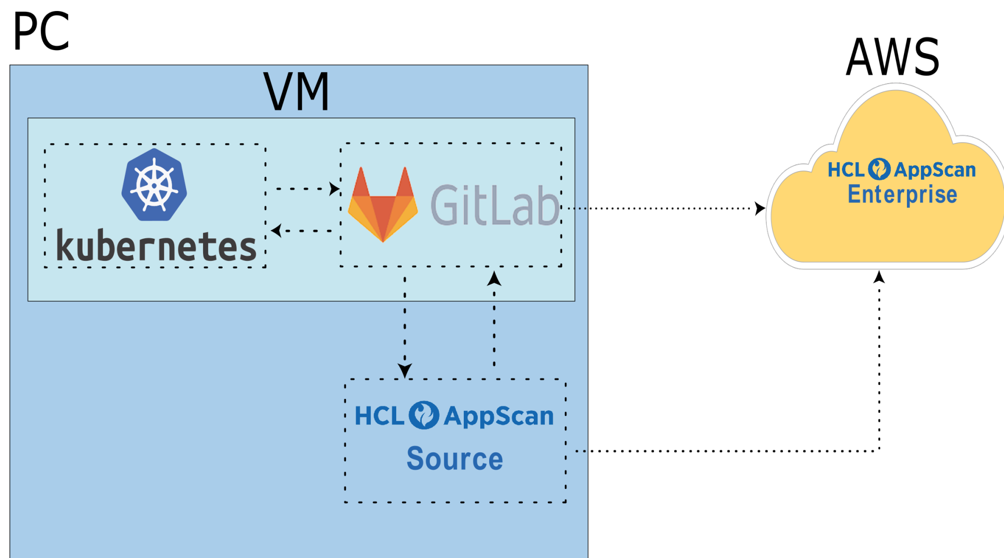

Figura 3.1: Architettura

In figura 3.1 è possibile visualizzare lo schema architetturale realizzato. Per questo lavoro di tesi sono state utilizzate tre macchine:

- Macchina Virtuale (VM) Linux Ubuntu 20.04 LTS
- Sistema host Windows 10 pro
- Server Amazon Web Server (AWS) in cloud

La prima macchina utilizzata è una macchina virtuale, in inglese *Virtual Machine* (VM), ed è un software che emula l'hardware di un computer. La VM utilizza le risorse hardware della macchina fisica, ma rimane isolata dal software installato sul computer host. Per funzionare correttamente, un componente chiamato hypervisor viene utilizzato per gestire le risorse hardware allocate alla VM. In questo modo il sistema operativo e i programmi possono essere eseguiti sulla *virtual machine* come se fossero installati sulla macchina fisica [37].

La VM e il sistema host comunicano tramite il protocollo **ssh** che permette di stabilire una sessione remota cifrata tramite interfaccia a riga di comando [38]. SSH è ampiamente utilizzato dagli amministratori di rete per la gestione remota di sistemi e applicazioni, consentendo loro di accedere a qualsiasi dispositivo supporti questo protocollo ed eseguire comandi o spostare file [39].
Il protocollo SSH supporta diversi metodi di autenticazione, tra cui l'autenticazione con chiave pubblica per connessioni automatizzate. L'autenticazione con chiave

pubblica è consigliata rispetto ad username e password per ragioni di sicurezza, infatti, l'autenticazione con chiave pubblica, fornisce una complessità crittografica che password lunghe non possono offrire [40].

La macchina virtuale e il servizio cloud comunicano attraverso le API REST, ovvero un'interfaccia di programmazione dell'applicazione, mentre si utilizzano dei protocolli proprietari tra AppScan Source e AppScan Enterprise.

3.2 GitLab: scelte implementative

L'installazione di GitLab [41] su un server non pone particolari difficoltà, ma è relativamente impegnativa in termini di tempo. Per gestire in maniera sicura il server è necessario, oltre all'installazione vera e propria, spendere del tempo per la configurazione iniziale.
Per questo lavoro di tesi si definisce l'indirizzo URL per raggiungere GitLab e i suoi registri; inoltre, si indica il percorso dei certificati SSL necessari per poter utilizzare HTTPS [42], ovvero un protocollo che crea un canale di comunicazione sicuro proteggendo l'integrità e la riservatezza dei dati scambiati.
Queste configurazioni si definiscono all'interno del file *gitlab.rb* situato nella cartella */etc/gitlab/* e sono rappresentate nel riquadro 3.2.

```

external_url 'https://gitlab.example.com'

registry_external_url 'https://gitlab.example.com:5050'

nginx['redirect_http_to_https'] = true

nginx['ssl_certificate'] = "/etc/gitlab/ssl/gitlab.example.com.crt"

nginx['ssl_certificate_key'] = "/etc/gitlab/ssl/gitlab.example.com.
    key"
```

Figura 3.2: gitlab.rb

Successivamente si crea una chiave SSH [43], ovvero delle credenziali di accesso diverse da username e password, che GitLab utilizza per comunicare in maniera sicura con Git.
Infine si configurano degli strumenti complementari necessari per la realizzazione della pipeline quali i container registry, GitLab Runner, ovvero un'applicazione dedicata all'esecuzione degli script della pipeline, e l'integrazione con Kubernetes.

Per evitare configurazioni complesse si può anche installare e utilizzare GitLab su un server cloud, in questo modo, il software può essere utilizzato subito. Tuttavia, il vantaggio di un'installazione manuale è la maggiore flessibilità in quanto gli utenti possono decidere autonomamente quali sono le risorse aggiuntive necessarie ed installare precisamente ciò di cui hanno bisogno per le proprie esigenze applicative.

3.2.1 Certificati SSL

SSL [44] è un protocollo di sicurezza che può essere aggiunto ad HTTP per incrementare l'affidabilità di un sito grazie all'autenticazione, l'integrità e la cifratura dei dati. Al giorno d'oggi, tutte le pagine web sono rilasciate via HTTPS.
Dopo aver installato GitLab bisogna creare dei certificati [45] per il nostro dominio e installarli nel progetto. Il motivo per cui i certificati sono così importanti è che cifrano la connessione tra il client e il server attraverso un sistema di autenticazione e convalidazione. In questo modo si proteggono i dati sensibili scambiati fra due sistemi, impedendo la lettura e l'intercettazione del traffico.
Uno dei primi passi che è stato fatto in questo progetto, è stato la creazione di una *certification authority* (CA) [46], ovvero un soggetto terzo di fiducia abilitato ad emettere e firmare una serie di certificati digitali tramite una procedura di certificazione che segue lo standard SSL. Il primo certificato serve per autenticare la CA mentre il secondo il server GitLab.

3.2.2 GitLab Runner

GitLab Runner [47] è un'applicazione che lavora con GitLab CI/CD per eseguire gli script della pipeline. Si può decidere di utilizzare i runner offerti da GitLab o installarli su un'infrastruttura che si possiede o si gestisce. Quest'ultima soluzione è quella adottata, ed è consigliata per ragioni di sicurezza, perché il codice sorgente e il flusso di dati non vengono trasmessi sulla rete ma le operazioni si eseguono in locale. Inoltre, utilizzando il runner su una macchina dedicata, si può garantire l'isolamento dei task e si evita un sovraccarico di lavoro da parte del server migliorando le performance. Installando GitLab tramite l'*omnibus package* è necessario integrare manualmente i runner ed è possibile modificarne la configurazione. Inoltre, avendo sulla stessa macchina GitLab e i Runners, si risparmia una quantità significativa di memoria.
I runner possono essere assegnati a qualunque progetto presente sul server tramite un'operazione chiamata registrazione. Tuttavia bisogna prestare attenzione alla visibilità concessa; infatti, gli *shared runner* possono accedere a tutti i repository di GitLab. In questo lavoro di tesi, sono stati utilizzati degli *specific runner* i quali possono essere assegnati soltanto ad un progetto.
Quando si registra un nuovo runner, bisogna scegliere l'esecutore (*executor*) che

determina l'ambiente in cui si eseguono gli script.
Esistono diversi tipi di executor [48]:

- **Shell**: è il più semplice da configurare. Tutte le dipendenze richieste per la build devono essere installate sulla stessa macchina in cui è installato GitLab Runner.
- **Virtual Machine**: questo tipo di esecutore permette di usare una macchina virtuale già creata, che viene clonata e usata per eseguire gli script della pipeline.
- **Docker**: esegue la build in un contenitore separato e isolato, in questo modo si crea un ambiente di costruzione pulito e con una gestione semplice delle dipendenze che possono essere messe all'interno dell'immagine Docker.
- **Docker Machine**: è una versione alternativa dell'esecutore precedente in cui si aggiunge il supporto per il ridimensionamento automatico, in questo modo si gestisce autonomamente la quantità di risorse da assegnare.
- **Kubernetes**: utilizza un cluster Kubernetes per le operazioni. L'esecutore, tramite le API, crea un nuovo Pod per ogni script che esegue.
- **SSH**: permette a GitLab Runner di connettersi a un server esterno ed eseguire lì tutte le operazioni.
- **Custom Executor**: è possibile specificare un ambiente di esecuzione personalizzato.

Soluzioni esplorate

Il primo executor adottato è stato *ssh* il quale richiede, durante la fase di registrazione, di specificare l'URL del server ssh, la porta, username e la password dell'utente che utilizza il servizio, infine il percorso del file contente la chiave ssh precedentemente creata. La configurazione risulta semplice e minimale, tuttavia, non è possibile utilizzare questa strategia per eseguire i test di sicurezza perché, in questi ultimi, è richiesta la connessione ad ulteriori macchine tramite il protocollo ssh, ma ciò non è possibile perché sarebbero necessari più terminali. Inoltre l'executor presenta un problema di sicurezza perché, per creare una connessione ssh, sono necessari uno username e una password che vengono salvate in chiaro all'interno del file di configurazione.
Un'altra soluzione testata è stata *Docker-in-Docker* ovvero un esecutore Docker che utilizza Docker stesso come immagine perché al suo interno ha tutti gli strumenti necessari per eseguire gli script della pipeline. Per poter utilizzare questa soluzione è necessario modificare il file *config.toml* presente nella cartella */etc/gitlab-runner*

e aumentare i privilegi grazie all'opzione *privileged = true*. I problemi di questa soluzione sono molteplici e sono legati alla raggiungibilità e alla sicurezza; infatti, è stato necessario definire nel file *config.toml* un URL alternativo formato dall'indirizzo IP del server GitLab perché il runner non riusciva a connettersi e a comunicare correttamente con quest'ultimo. Un'altra impostazione aggiunta è stata *environment = ["GIT_SSL_NO_VERIFY=true"]* in questo modo si disattiva il controllo sui certificati SSL che non venivano riconosciuti perché autofirmati. Infine anche l'integrazione con Kubernetes è risultata complessa perché i due servizi, Docker e Kubernetes, appartenevano a sotto-reti differenti, pertanto, non riuscivano a comunicare correttamente.

Soluzione adottata

Per questo lavoro di tesi, è stato scelto come esecutore di GitLab Runner quello shell perché è pronto all'uso. Se si vogliono apportare delle modifiche alla configurazione queste risultano minimali, l'unica possibilità è decidere il tipo di shell che eseguirà lo script: bash, powershell o sh. Inoltre, utilizzando l'esecutore shell, è possibile accedere a tutti i programmi installati sulla macchina Linux ed utilizzare degli script esterni salvati sul filesystem.
In generale, non è sicuro eseguire i test con lo shell executor. Le operazioni vengono eseguite con i permessi dell'utente di GitLab Runner e possono accedere al codice di altri progetti presenti sullo stesso server. Questa soluzione è adottabile solo per le build affidabili. Tuttavia, riducendo i permessi assegnati a GitLab Runner è possibile limitare questo problema.

3.2.3 Minikube

Come ambiente di produzione è stato scelto Kubernetes perché si sta affermando come soluzione completa, non solo perché gestisce applicazioni, servizi e risorse, ma organizza anche l'infrastruttura presente composta da macchine virtuali. Allo stesso tempo, Kubernetes offre processi di controllo che monitorano lo stato di salute dei nodi, pod e container disponibili nel cluster.
Minikube [49] è uno strumento che permette di eseguire il deploy di un applicativo all'interno di in un cluster locale, offre le stesse funzionalità di Kubernetes ma con capacità ridotte; infatti, è possibile utilizzare un solo nodo. E' una distribuzione più snella che ottimizza le risorse e non bisogna investire tempo nella configurazione, pertanto, è la soluzione per piccoli progetti basati su container.
Minikube non necessita di un'infrastruttura complessa e permette di avere un ambiente simile ad uno di produzione reale su un server dedicato.
Per comunicare in maniera corretta con GitLab, minikube deve contenere il dominio e l'indirizzo IP all'interno del file *hosts* collocato nella directory */etc/* del cluster

[50]. Il file hosts è un file di testo di piccole dimensioni ma molto utile, che memorizza i nomi host e gli indirizzi IP a cui sono associati. La sua funzione è quella di indirizzare correttamente la navigazione nella rete a partire dal nome di un sito web. Risolvendo i nomi in indirizzi IP numerici, il file hosts è uno strumento indispensabile per un protocollo di rete. Tutte le volte che si accede ad un sito web, il dispositivo verifica se il file hosts contiene una voce o una regola relativa al sito stesso. Per modificare il file è necessario prendere il controllo del cluster, tramite il protocollo SSH, e aggiungere la seguente riga:

```
1 192.168.1.86  gitlab.example.com
```

dove 192.168.1.86 è l'IP del server di Gitlab all'interno della rete locale.

3.3 GitLab: fasi della pipeline

Per questo lavoro di tesi è stata creata una pipeline composta da tre fasi, build, test e deploy, che automatizzano in maniera sicura il rilascio del codice.
L'obiettivo è eseguire automaticamente l'analisi statica dell'applicativo e dei controlli di sicurezza su tutta l'infrastruttura Kubernetes.
La pipeline che è stata costruita per questo lavoro di tesi ha la seguente struttura:

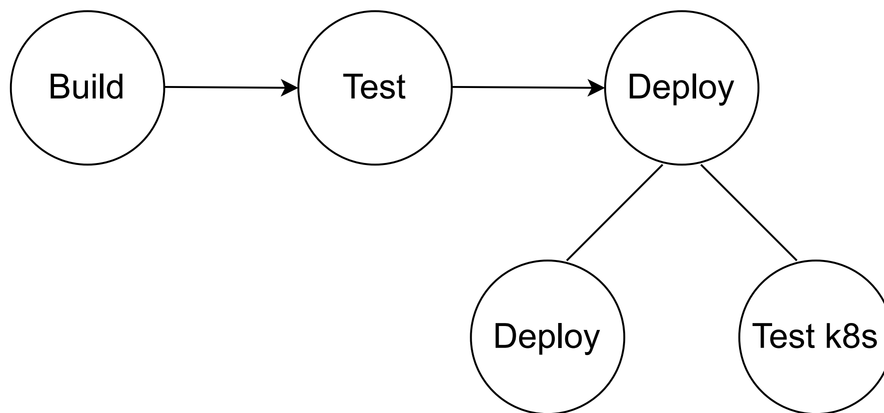

Figura 3.3: Pipeline

In figura 3.3 sono presenti i tre passaggi principali, build, test e deploy, quest'ultimo è composto da due sottofasi: il deploy in un ambiente di produzione e i test di natura dinamica sull'infrastruttura Kubernetes.
La pipeline [51] è definita all'interno di un file chiamato *.gitlab-ci.yml* che definisce le operazioni da eseguire. In generale, nel file sono presenti le seguenti componenti:

- **Variables**: contiene delle variabili comuni a tutti i processi.
- **Stages**: in cui si fissano le fasi costitutive della pipeline.
- **Before_script**: si definiscono delle operazioni preliminari da eseguire prima dello script.

In questo lavoro di tesi sono presenti solo le variabili e gli stages della pipeline:

```
variables:
  KUBE_NAMESPACE: dvwa-3-production

stages:
  - build
  - test
  - deploy
```

Figura 3.4: Variabili e stages

3.3.1 Build

Quando uno sviluppatore apporta delle modifiche al codice, tramite l'operazione di *push*, la pipeline si avvia. La prima fase è la build che ha l'obiettivo di creare un'immagine Docker del codice sorgente e di caricare il risultato all'interno del container registry di GitLab.

```
build app:
  stage: build
  script:
    - docker build -t $CI_REGISTRY/root/dvwa/image:latest .
    - docker login -u $CI_REGISTRY_USER -p $CI_REGISTRY_PASSWORD
      $CI_REGISTRY
    - docker push $CI_REGISTRY/root/dvwa/image:latest
```

Figura 3.5: Build

In figura 3.5 è possibile visualizzare lo script relativo alla fase di build.

Ogni fase della pipeline è formata da un nome (*build app*), lo stage a cui appartiene e lo script da eseguire.

La *docker build* costruisce un'immagine Docker a partire dal codice sorgente presente nel repository. L'opzione -t serve per assegnare un nome e un tag al risultato e il formato utilizzato è *nome:tag*. I tag sono importanti perché contengono informazioni sulla versione dell'immagine e la identificano univocamente.
Prima di poter caricare l'immagine nel container registry di GitLab bisogna autenticarsi con la *docker login*. Come credenziali si utilizzano le variabili *CI_REGISTRY_USER* e *CI_REGISTRY_PASSWORD* che sono create da GitLab per automatizzare il processo di build e deploy. L'autenticazione è valida solo per la build, infatti, ogni volta che si vuole accedere ai registri bisogna ripetere questa procedura.
Infine, grazie alla *docker push*, si carica il risultato della build, identificato dalla coppia *nome:tag*, nel container registry.

3.3.2 Test

Terminata con successo la build inizia la fase di test. AppScan Source esegue l'analisi statica del codice e le vulnerabilità trovate sono inviate ad AppScan Enterprise.

```
1  test:
2    stage: test
3    script:
4      - echo '
5          cd "C:\Users\Luca\Desktop" &&
6          (git clone git@gitlab.example.com:root/dvwa.git ||
7          (cd dvwa && git pull &&
8             del /f "AppScan Source Report.pdf")) &&
9          copy "C:\Users\Luca\Desktop\New folder\dvwa.ppf"
10              C:\Users\Luca\Desktop\dvwa &&
11         cd "C:\Program Files (x86)\IBM\AppScanSource\bin" &&
12         AppScanSrcCli.exe SCR Run_Assessments.txt &&
13         cd C:\Users\Luca\Desktop\dvwa &&
14         git add * &&
15         git commit -am "commited AppScan report" &&
16         git push -o ci.skip' |
17       sshpass -f $HOME/cwd/ssh_pass_file ssh Luca@192.168.1.240
```

Figura 3.6: Test

In figura 3.6 è possibile visualizzare la fase di test.

Questo script è strutturato su due livelli. Il livello esterno, ovvero *sshpass -f $Home/cwd/ssh_pass_file ssh Luca@192.168.1.240*

crea una connessione SSH con l'host 192.168.1.240, la macchina Windows. Il nome dell'utente che si vuole autenticare è posto prima della @ mentre la password è salvato nel file *ssh_pass_file*.
Il livello interno contiene uno script da eseguire sulla nuova macchina.
La prima istruzione è una *cd ("change directory")* che viene utilizzata per cambiare la cartella di lavoro corrente. Si clona il progetto GitLab da analizzare, e, se è già presente, si esegue l'operazione di git pull che sincronizza il contenuto della cartella con quello del repository. Successivamente si procede con il copiare, all'interno della cartella, un file di progetto AppScan Source da analizzare: dvwa.ppf.
Al termine di queste operazioni di configurazione, si esegue l'analisi statica del codice attraverso lo script *Run_Assessments.txt*.
Infine si produce un report che si carica sul repository GitLab.

Run_Assessments.txt

```
1  login_file https://15.236.2.108:9443/ase/ C:\Users\Luca\.ounce\
       ouncecli.token -acceptssl
2  LOG ON C:\myLogFile.log
3  cd dvwa
4  unregister
5  cd ..
6  del dvwa
7  new dvwa C:\Users\Luca\Desktop\dvwa
8  cd dvwa
9  im C:\Users\Luca\Desktop\dvwa\dvwa.ppf
10 refresh
11
12 scan C:\Users\Luca\Desktop\dvwa\dvwa\dvwa.ozasmt -name dvwa
13 register
14 publishassess
15 report "DISA Application Security and Development STIG V4R4" pdf-
       comprehensive "C:\Users\Luca\Desktop\dvwa\reports\AppScan Source
       Report.pdf"
16 publishassessase -aseapplication dvwa C:\Users\Luca\Desktop\dvwa\dvwa
       \dvwa.ozasmt
17 quit
```

Figura 3.7: Run_Assessment

In figura 3.7 è possibile visualizzare lo script realizzato per eseguire l'analisi statica.

Questo script autentica l'istanza di AppScan Source con AppScan Enterprise, il server di monitoraggio. I parametri utilizzati sono l'URL del server, ovvero il suo

indirizzo IP *https://15.236.2.108:9443/ase/*, e un token di accesso. Il flag *acceptssl* si usa per accettare automaticamente il certificato SSL. Inoltre, grazie all'operazione di *LOG ON*, si registrano tutti i passaggi e i relativi output all'interno di un file. Siccome questo file può essere eseguito più volte è necessario cancellare le applicazioni create in precedenza con l'operazione di *unregister* e *del*.
Successivamente, su AppScan Source, si crea una nuova applicazione chiamata *dvwa* e si importa il relativo file di progetto.
Dopo aver eseguito queste operazioni inizia la scansione del codice sorgente che ha l'obiettivo di trovare le vulnerabilità presenti.
Tramite il comando *report* si genera un documento contenente le vulnerabilità identificate e il loro grado di pericolosità. Gli stessi risultati sono pubblicati nel database di AppScan Source con il comando *pubblishassess* e sul server di monitoraggio con l'operazione di *pubblishassessase*.

La fase di test è stata quella più ardua perché AppScan Source non è compatibile con il sistema operativo della macchina utilizzata per ospitare GitLab, pertanto, si è optato per suddividere i flussi di lavoro su macchine diverse. Creare questa architettura è stato complesso perché al prodotto non è stata allegata una documentazione per configurare correttamente l'ambiente.

3.3.3 Deploy

La fase del deploy è composta da due parti: il rilascio del codice e i test sull'infrastruttura Kubernetes.

Rilascio

Il deploy è la fase di rilascio del codice in un ambiente, ne esistono di due tipi: *staging* e *production* [52].
L'ambiente di staging è configurato per ospitare la versione dell'applicativo che si vuole rilasciare in produzione. Il team di sviluppo può eseguire le correzioni e le modifiche finali prima di rendere il prodotto disponibile al cliente.
L'ambiente di production è configurato per ospitare la versione di rilascio finale di un prodotto destinato agli utenti finali.
La scelta dell'ambiente viene comunicata all'interno del file *.gitlab-ci.yml* nella parte successiva allo script.

```
deploy:
  stage: deploy
  script:
   - kubectl config set-cluster "minikube" --server="$KUBE_URL"
       --certificate-authority="$KUBE_CA_PEM_FILE"
   - kubectl config set-credentials "minikube" --token="$KUBE_TOKEN"
```

```
7     - kubectl config set-context "minikube" --cluster="minikube"
8         --user="minikube" --namespace="$KUBE_NAMESPACE"
9     - kubectl config use-context "minikube"
10        --namespace="$KUBE_NAMESPACE"
11    - kubectl delete secret gitlab.example
12        --namespace="$KUBE_NAMESPACE" --ignore-not-found
13    - kubectl create secret docker-registry gitlab.example
14        --docker-server="https://${CI_REGISTRY}"
15        --docker-username="$CI_REGISTRY_USER"
16        --docker-password="$CI_JOB_TOKEN"
17        --namespace="$KUBE_NAMESPACE"
18    - kubectl delete deployment dvwa-deployment --ignore-not-found
19    - sed -i "s~__CI_REGISTRY_IMAGE__~${CI_REGISTRY_IMAGE}~"
20        manifest/deployment.yml
21    - sed -i "s/__CI_ENVIRONMENT_SLUG__/${CI_ENVIRONMENT_SLUG}/"
22        manifest/deployment.yml manifest/service.yml
23        manifest/ingress.yml
24    - sed -i "s/__VERSION__/${CI_BUILD_REF_NAME}_${CI_BUILD_REF}/"
25        manifest/deployment.yml
26    - kubectl apply -f manifest/deployment.yml -n $KUBE_NAMESPACE
27    - kubectl apply -f manifest/service.yml -n $KUBE_NAMESPACE
28    - kubectl apply -f manifest/ingress.yml -n $KUBE_NAMESPACE
29    - kubectl rollout status -f manifest/deployment.yml
30        -n $KUBE_NAMESPACE
31    - kubectl get all,ing -l ref=${CI_ENVIRONMENT_SLUG}
32        -n $KUBE_NAMESPACE
33
34  environment:
35    name: production
36    url: http://dvwa.com:30002
37    kubernetes:
38      namespace: $KUBE_NAMESPACE
39  only:
40    - master
```

Figura 3.8: Deploy

In figura 3.8 è possibile visualizzare la fase di deploy.

Per questo lavoro di tesi è stato utilizzato l'ambiente di produzione per integrarlo con il cluster Kubernetes, inoltre si è scelto di eseguire il deploy indipendentemente dalle criticità delle vulnerabilità trovate nella fase di test perché l'obiettivo era quello di eseguire anche dei controlli sull'infrastruttura Kubernetes, pertanto, se la pipeline si fosse dovuta interrompere le analisi non sarebbero state eseguite.
L'URL *http://dvwa.com:30002* è l'indirizzo utilizzabile per raggiungere l'applicativo tramite l'interfaccia web.

Utilizzando Kubernetes, tramite la sintassi *namespace: $KUBE_NAMESPACE*, dichiariamo in quale cluster virtuale effettuare il rilascio. Il cluster virtuale è una partizione del cluster Kubernetes. KUBE_NAMESPACE è il nome di una variabile e contiene *dvwa-3-production* ovvero il nome del progetto, il suo id e l'ambiente selezionato.
L'unico ramo del repository di cui si esegue il deploy è il master.
Durante la fase iniziale del deploy si configurano le variabili d'ambiente di minikube, il cluster locale Kubernetes. Una delle operazioni eseguite è la creazione di un nuovo segreto, chiamato *gitlab.example*, con le credenziali di accesso a Docker, che verranno utilizzate da GitLab per autenticarsi al cluster.
Le tre istruzioni più importanti si trovano nella seconda parte dello script e sono le *apply*. La prima esegue il deployment dell'immagine creata nella fase di build e salvata nel container registry di GitLab. Il file *deployment.yml*, che si trova nella cartella *manifest* all'interno del repository, contiene tutte le informazioni necessarie per creare una distribuzione dell'applicazione. Nel file YAML è possibile decidere il numero di repliche da eseguire, per questo lavoro di tesi è uno.

```
1      spec:
2       replicas: 1
```

Per autenticarsi, il segreto precedentemente creato viene utilizzato con la seguente sintassi:

```
1      imagePullSecrets:
2          - name: gitlab.example
```

La seconda apply crea un *Service*, ovvero un metodo per esporre l'applicativo in esecuzione nel pod sulla rete. Kubernetes assegna al pod un indirizzo IP e un nome DNS. All'interno del manifest *service.yml* si specifica la porta del nodo del cluster, la 30002, e quella verso cui si vuole destinare il servizio ovvero la porta 80 di HTTP.
La terza apply crea un oggetto API che gestisce gli accessi esterni ai service nel cluster.
L'istruzione *kubectl rollout status -f manifest/deployment.yml -n $KUBE_NAMESPACE* attende il completamento del deployment per proseguire.
Infine, nell'ultima istruzione, si ha una stampa degli stati degli oggetti creati per avere un feedback visivo del risultato ottenuto.

Test sull'infrastruttura Kubernetes

L'ultima parte della pipeline è incaricata di eseguire i controlli sull'infrastruttura Kubernetes. Per questo lavoro di tesi sono stati utilizzati degli strumenti di

monitoraggio del cluster per verificarne l'integrità e le funzionalità di sicurezza.

```
post-deploy:
  stage: deploy
  script:
    - ssh -T git@gitlab.example.com
    - rm -R dvwa
    - cd /home/luca/cwd/
    - rm -R dvwa
    - git clone git@gitlab.example.com:root/dvwa.git
    - cd dvwa/reports
    - rm -f *.json
    - rm -f *.txt
    - kubectl apply -f ../manifest/kube-bench.yaml
    - kubectl wait --for=condition=complete
        -f ../manifest/kube-bench.yaml --timeout=60s
    - POD=$(kubectl get pods | grep kube-bench | awk '{print $1}')
    - kubectl logs $POD > kube-bench.json
    - kubectl delete -f ../manifest/kube-bench.yaml

    - kubectl apply -f ../manifest/kube-hunter.yaml
    - kubectl wait --for=condition=complete
        -f ../manifest/kube-hunter.yaml --timeout=60s
    - POD=$(kubectl get pods | grep kube-hunter | awk '{print $1}')
    - kubectl logs $POD > kube-hunter.txt
    - kubectl delete -f ../manifest/kube-hunter.yaml


    - (kubesec scan ../manifest/deployment.yml > kubesec.json)
      || true

    - kubectl apply -f ../manifest/kubeaudit.yaml
    - (kubeaudit all --format json > kubeaudit.json ) || true


    - git add kube-bench.json kube-hunter.txt kubesec.json
        kubeaudit.json
    - git commit -am "commited reports"
    - git push -o ci.skip
    - python3 /home/luca/cwd/parser_and_uploader.py $APP_SCAN
```

Figura 3.9: Test sull'infrastruttura Kubernetes

In figura 3.9 è possibile visualizzare la fase test sull'infrastruttura Kubernetes.

La prima operazione è la clonazione del repository sulla macchina Linux, in questo modo KubeSec può accedere al codice da analizzare. Soltanto questo strumento è installato direttamente sulla macchina, gli altri vengono eseguiti a partire da un file di configurazione YAML caricato sul repository.
Il primo strumento utilizzato è **kube-bench** [53] che verifica se Kubernetes è distribuito in maniera sicura eseguendo dei benchmark, ovvero dei test specifici per valutare le prestazioni di un dispositivo o l'efficacia di un processo tecnico. Questo tool può essere eseguito all'interno di una pod, pertanto, grazie alla *kubectl apply*, si fornisce al cluster il file YAML per l'installazione e l'esecuzione di kube-bench.
La *kubectl wait* attende la fine delle scansioni prima di poter leggere il report.
Utilizzando l'espressione *(kubectl get pods | grep kube-bench | awk 'print $1')* si recupera il nome della pod in cui è stato installato kube-bench.
La *kubectl logs* permette di leggere i logs, ovvero il registro delle attività, che contiene i risultati delle analisi. L'output del comando viene salvato nel file *kube-bench.yaml*.
Il secondo strumento utilizzato è **kube-hunter** [54] che rileva le vulnerabilità del cluster. Si sonda un dominio o un range di indirizzi per trovare le porte aperte dedicate a Kubernetes, si controlla l'esistenza delle risorse Kubernetes e se il cluster è attaccabile sfruttando le vulnerabilità note. Anche questo strumento è eseguito all'interno di una pod, pertanto, la sua installazione e l'esecuzione è analoga al caso precedente.
Si prosegue con le analisi di **KubeSec** [55] che convalida i file di configurazione e i manifesti usati per il deployment, verifica l'assenza di debolezze quali l'esecuzione di immagini come utente non amministrativo e altri rischi comuni. Infine per ogni vulnerabilità rilevata KubeSec fornisce un punteggio di pericolosità.
Questo software è installato all'interno della VM ed è eseguibile dal terminale con il comando *kubesec scan ../manifest/deployment.yml > kubesec.json*. L'opzione scan inizia le analisi del file YAML passato come argomento dello script e ritorna un vettore di oggetti JSON contenenti le vulnerabilità trovate. Il risultato viene salvato nel file *kubesec.json*.
L'ultimo test eseguito è **kubeaudit** [56] che controlla se i pod hanno impostazioni predefinite non sicure o se i container consentono la *privilege escalation*, ovvero lo sfruttamento di un errore di configurazione al fine di acquisire il controllo di risorse normalmente precluse [57]. Kubeaudit produce tre tipi di risultati: *Error*, *Warning* e *Info*, il primo è un problema di sicurezza o una configurazione invalida, il secondo è una best practice non rispettata, mentre il terzo è un messaggio di natura informativa che non richiede alcuna azione. Questo strumento viene installato all'interno di una pod e, con il comando *kubeaudit all –format json*, si iniziano le analisi e l'output prodotto è realizzato nel formato JSON.

Al termine delle scansioni tutti i report prodotti vengono caricati all'interno del repository. Siccome la *git push* aziona la pipeline, si utilizza il flag *-o ci.skip* che ne

blocca l'esecuzione.

L'ultima istruzione esegue lo script python *parser_and_uploader.py* che per essere eseguito necessita del parametro *$APP_SCAN* contenente la password per autenticarsi al server di monitoraggio ed è salvato all'interno di GitLab.
Per poter caricare sul server le vulnerabilità trovate dalle analisi dell'ambiente Kubernetes, è necessario che i report generati abbiano un formato accettabile da AppScan Enterprise, pertanto, lo script crea un file CSV conforme alle richieste e lo invia al server di monitoraggio.

parser_and_uploader.py

```

import pandas as pd
import json
import requests
import sys

cnt = 1
to_csv = {}

with open ('/home/luca/cwd/dvwa/reports/kube-bench.json') as f:
    data = json.load(f)

for d in data['Controls']:
    for t in d['tests']:
        for r in t['results']:
            str1 = ''
            str2 = 'Remediation: '
            description = str1.join(r['test_info']).replace('\n',' ')
            remediation = str2 + (r['remediation']).replace('\n',' ')
            tmp = {"Issue Type" : 'kube-bench',"Description" :
    description, "Location" : remediation}
            to_csv.update({cnt:tmp})
            cnt = cnt + 1

f.close()

with open ('/home/luca/cwd/test2.txt') as f:
    for line in f:
        data = json.loads(line)
        str1 = ''
        str2 = 'Message: '
        description = str1.join(data['AuditResultName']).replace('\n'
    ,' ')
        message = str2 + (data['msg']).replace('\n',' ')
```

```
33          tmp = {"Issue Type" : 'kubeaudit',"Description" : description
     , "Location" : message}
34          to_csv.update({cnt:tmp})
35          cnt = cnt + 1
36
37 f.close()
38
39 with open ('/home/luca/cwd/dvwa/reports/kubesec.json') as f:
40      text = f.read()
41      datas = json.loads(text)
42      data = datas[0]
43      advice = data['scoring']['advise']
44      for a in advice:
45          str1 = ''
46          str2 = 'Reason: '
47          description = str1.join(a['id']).replace('\n',' ')
48          reason = str2 + (a['reason']).replace('\n',' ')
49          tmp = {"Issue Type" : 'kubesec',"Description" : description,
     "Location" : reason}
50          to_csv.update({cnt:tmp})
51          cnt = cnt + 1
52
53 f.close()
54
55 with open ('/home/luca/cwd/dvwa/reports/kube-hunter.txt') as f:
56      text = f.read()
57      index = text.index('{"nodes"')
58      string = json.loads(text[index:])
59      for v in string['vulnerabilities']:
60          str1 = ''
61          str2 = 'Description: '
62          description = str1.join(v['vulnerability']).replace('\n',' ')
63          reason = str2 + (v['description']).replace('\n',' ')
64          tmp = {"Issue Type" : 'kube-hunter',"Description" :
     description, "Location" : reason}
65          to_csv.update({cnt:tmp})
66          cnt = cnt + 1
67
68 f.close()
69
70 pdObj = pd.read_json(json.dumps(to_csv), orient='index')
71 pdObj.to_csv('data.csv',index=None, mode='w+')
72
73 url = "https://15.236.2.108:9443/ase/api/"
74 user = "AWS\Administrator"
75 featureKey = "AppScanEnterpriseUser"
76 password = sys.argv[1]
77
78 data = {"userId":user, "password":password,"featureKey":featureKey}
```

```
79
80 response = requests.post(url+"login",json=data, verify=False)
81
82 payload=json.loads(response.text)
83 asc_xsrf_token=payload['sessionId']
84
85 header = {"asc_xsrf_token":asc_xsrf_token}
86 cookie = {"asc_session_id":asc_xsrf_token}
87
88 response = requests.get(url+"applications/?columns=name",headers=
       header, cookies=cookie, verify=False)
89 applications=json.loads(response.text)
90 appId = str(applications[0]['id'])
91
92 response = requests.get(url+"scanners/",headers=header, cookies=
       cookie,verify=False)
93 scanners = json.loads(response.text)
94
95 for s in scanners['scannerColl']:
96     if(s['name']=="Generic"):
97         scanId=str(s["id"])
98 f = open ('/home/luca/cwd/data.csv')
99
100 data = {"uploadedfile":f}
101 response = requests.post(url+"issueimport/"+appId+"/"+scanId, headers
       =header, cookies=cookie, files=data,data={"scanName":"Generic"} ,
       verify=False)
102 response = requests.get(url+"issueimport/currentstatus",headers=
       header,cookies=cookie,verify=False)
103
104 f.close()
```

Figura 3.10: parser_and_uploader.py

In figura 3.10 è possibile visualizzare lo script incaricato di pubblicare i dati su AppScan Enterprise.

Le prime quattro funzioni dello script analizzano i report prodotti dalle scansioni creano un file CSV con i seguenti campi:

- **Issue Type** che contiene il nome dello strumento che ha trovato la vulnerabilità
- **Description** ovvero una descrizione della vulnerabilità
- **Location** contiene una possibile soluzione al problema

Il nome del campo non è strettamente correlato con il suo contenuto, ma vengono utilizzati perché è il formato richiesto e accettato da AppScan Enterprise per poter pubblicare i risultati sul server.
Nella seconda parte dello script si utilizzano le API REST del server, la prima chiamata è la *login* ed è necessaria per autenticarsi tramite username e password, quest'ultima è passata come argomento dello script. La relativa risposta contiene un token che deve essere conservato e utilizzato nelle API successive per autenticare le chiamate.
Per poter pubblicare i risultati delle scansioni sono necessari due parametri, ottenibili tramite l'API *applications* e *scanners*, e sono l'id dell'applicazione a cui si riferiscono le vulnerabilità e l'id dello scanner che ha eseguito le analisi. Siccome non è supportata un'integrazione nativa con gli strumenti di Kubernetes, si è utilizzato uno scanner generico.
L'API incaricata di importare sul server le vulnerabilità trovate è la *issueimport/appId/scanId*. Se questa chiamata ritorna il codice 202 la richiesta è stata accettata e il caricamento dei dati è in corso. Per verificarne lo stato si utilizza l'ultima API *issueimport/currentstatus*, che ne verifica lo stato del caricamento.

Capitolo 4

Risorse utilizzate

4.1 Specifiche tecniche delle macchine

Per questo lavoro di tesi sono state utilizzate più macchine per suddividere i carichi lavoro, inoltre AppScan Source e AppScan Enterprise non sono disponibili per le distribuzioni Linux, pertanto, sono stati impiegati più sistemi operativi, ognuno con un ruolo ben preciso.
La prima è una macchina virtuale (VM) Linux, la versione del sistema operativo è Ubuntu 20.04 LTS a cui sono stati assegnati 8 GB di RAM e 2 core, inoltre l'immagine è salvata sul hard disk del sistema host. Al suo interno sono stati installati GitLab, Docker e minikube. Il ruolo di questa macchina è quello di eseguire la pipeline di GitLab e coordinare le comunicazioni con gli altri servizi. La connessione con la seconda macchina è gestita all'interno del file *.gitlab-ci.yml* utilizzando il protocollo ssh, mentre quella con il server AWS avviene grazie alle API REST contenute in uno script python.

La seconda macchina è il sistema host, ovvero il calcolatore che ospita la VM ed è accessibile al client tramite protocolli di rete. Il sistema operativo utilizzato è Windows 10 pro, ha 16 GB di RAM e il processore è Intel ®Core™i7-10700 CPU @ 2.90GHz.
Questa macchina è stata utilizzata per ospitare AppScan Source, eseguire l'analisi statica del codice presente nel repository GitLab e pubblicare i risultati sul server di monitoraggio, ovvero AppScan Enterprise.

Il terzo componente è un server AWS [58], in questo modo si utilizza il *cloud computing*, ovvero la distribuzione tramite la rete delle risorse IT utilizzando una tariffazione basata sul consumo, pertanto, si pagano solo le risorse necessarie. Piuttosto che acquistare, possedere e mantenere i server fisici, è possibile accedere a servizi tecnologici, quali capacità di calcolo, storage e database, sulla base delle proprie necessità.

Il servizio utilizzato è *Amazon Elastic Compute Cloud (EC2)* che fornisce capacità computazionali scalabili nel cloud. Il tipo di istanza è *t2.xlarge* che è uno standard che fornisce 16 GiB di RAM, 4 vCPU e una piattaforma 64-bit.
Il sistema operativo installato in questa istanza è Windows Server 2012 R2 ed ospita AppScan Enterprise, ovvero il server utilizzato per raccogliere tutte le vulnerabilità e svolgere le operazioni di monitoraggio.

4.2 Requisiti delle applicazioni

4.2.1 GitLab

La versione di GitLab utilizzata è la 13.11.2 Enterprise Edition. GitLab è stato progettato per essere eseguito su macchine Linux, le distribuzioni supportate sono [59]:

- Ubuntu (16.04/18.04/20.04)
- Debian (9/10)
- CentOS (7/8)
- openSUSE Leap (15.2)
- SUSE Linux Enterprise Server (12 SP2/12 SP5)
- Red Hat Enterprise Linux
- Scientific Linux
- Oracle Linux

Requisiti Software

L'unico requisito software richiesto è Redis 6.0 o una versione superiore. Per questo lavoro di tesi la versione utilizzata è la 6.0.12. Redis [60] è un database NoSQL basato su una strutture dati a dizionario, ovvero chiave-valore: ogni valore immagazzinato è abbinato ad una chiave univoca che ne permette il recupero. Redis memorizza tutte le sessioni degli utenti e la coda delle attività in background.

Requisiti Hardware

Lo spazio di archiviazione richiesto dipende dalla dimensione del repository che si vuole creare su GitLab, il pacchetto di installazione, invece, richiede 2.5 GB di spazio disponibile.

I requisiti della CPU e della memoria RAM dipendono dal numero di utenti e dal carico di lavoro atteso: per 500 utenti sono consigliati 4 core e 4 GB di RAM, mentre per 1000 utenti le risorse raddoppiano quindi 8 core e 8 GB di RAM.

Database

PostreSQL è l'unico database supportato ed è incluso nel pacchetto di installazione. Il supporto verso altri database è stato rimosso dalla versione 12.1 di GitLab; pertanto, è necessaria una migrazione verso questa banca di dati.
Il server che esegue PostreSQL deve avere almeno 5-10 GB di spazio di archiviazione disponibile, anche se questo dipende dal numero di utilizzatori. GitLab 13 richiede PostreSQL 11 o una versione superiore, pertanto, in questo lavoro di tesi si è utilizzato PostreSQL 12.6.

4.2.2 HCL AppScan Source

La versione di AppScan Source utilizzata è *AppScan Source for Automation 10.0.4.0*. Questo strumento è stato progettato per essere eseguito su macchine Windows o su server Linux, le distribuzioni supportate sono [61]:

- Windows 10 Education
- Windows 7 Enterprise o superiori
- Windows 7 Professional o superiori
- Windows 7 Ultimate
- Windows Server 2008 R2 Enterprise Edition
- Windows Server 2008 R2 Standard Edition o superiori
- Windows Server 2012 Datacenter Edition o superiori
- Windows Server 2012 Essentials Edition o superiori
- Windows Server 2012 Foundation Edition
- Red Hat Enterprise Linux (RHEL) Server 6 o 7
- Red Hat Enterprise Linux (RHEL) Workstation 6 o 7

Requisiti Software

AppScan Source richiede AppScan Enterprise versione 9.0.3 e Rational License Key Server 8.1.1 o successiva. In questo modo, ci si connette al server per eseguire scansioni e accedere ai dati condivisi.

Requisiti Hardware

Lo spazio di archiviazione richiesto è 3 GB, mentre ulteriori 4 GB sono necessari per l'installazione. Durante la scansione di applicazioni di grandi dimensioni potrebbe servire spazio su disco aggiuntivo per ospitare gli script di AppScan Source.
La RAM minima richiesta è 2 GB, per eseguire al meglio le scansioni 8 GB sono consigliati, mentre, Per quanto riguarda la CPU, 2 core sono sufficienti.
Siccome AppScan Source comunica con AppScan Enterprise, è necessaria una connessione alla rete pubblica.

4.2.3 HCL AppScan Enterprise

La versione di AppScan Enterprise utilizzata è *AppScan Enterprise 9.0.3.14*. Questo strumento è stato progettato per essere eseguito su un server Windows, le distribuzioni supportate sono [62]:

- 64-bit Windows Server 2008 R2 SP1 (64-bit)
- Windows 2012 Server x86-32, 64-bit tolerate
- Windows 2012 Server R2 (DataCenter) x86-32, 64-bit tolerate
- Windows 2012 Server R2 (Standard) x86-32, 64-bit tolerate
- Windows Server 2016 (Standard and Data-centre) x86-32, 64 bit tolerate

I seguenti componenti sono installati automaticamente durante l'installazione:

- .NET 4.6.2 framework
- IIS 7.5 e le sue dipendenze
- Rational®License Server

Si richiede una CPU quad-core, 16 GB di RAM e 200 GB di spazio su disco.
AppScan Enterprise deve comunicare con un database SQL per contenere le vulnerabilità trovate. Le versioni supportate sono:

- Microsoft SQL Server 2008 SP3
- Microsoft SQL Server 2008 R2 SP2
- Microsoft SQL Server 2012
- Microsoft SQL Server 2014

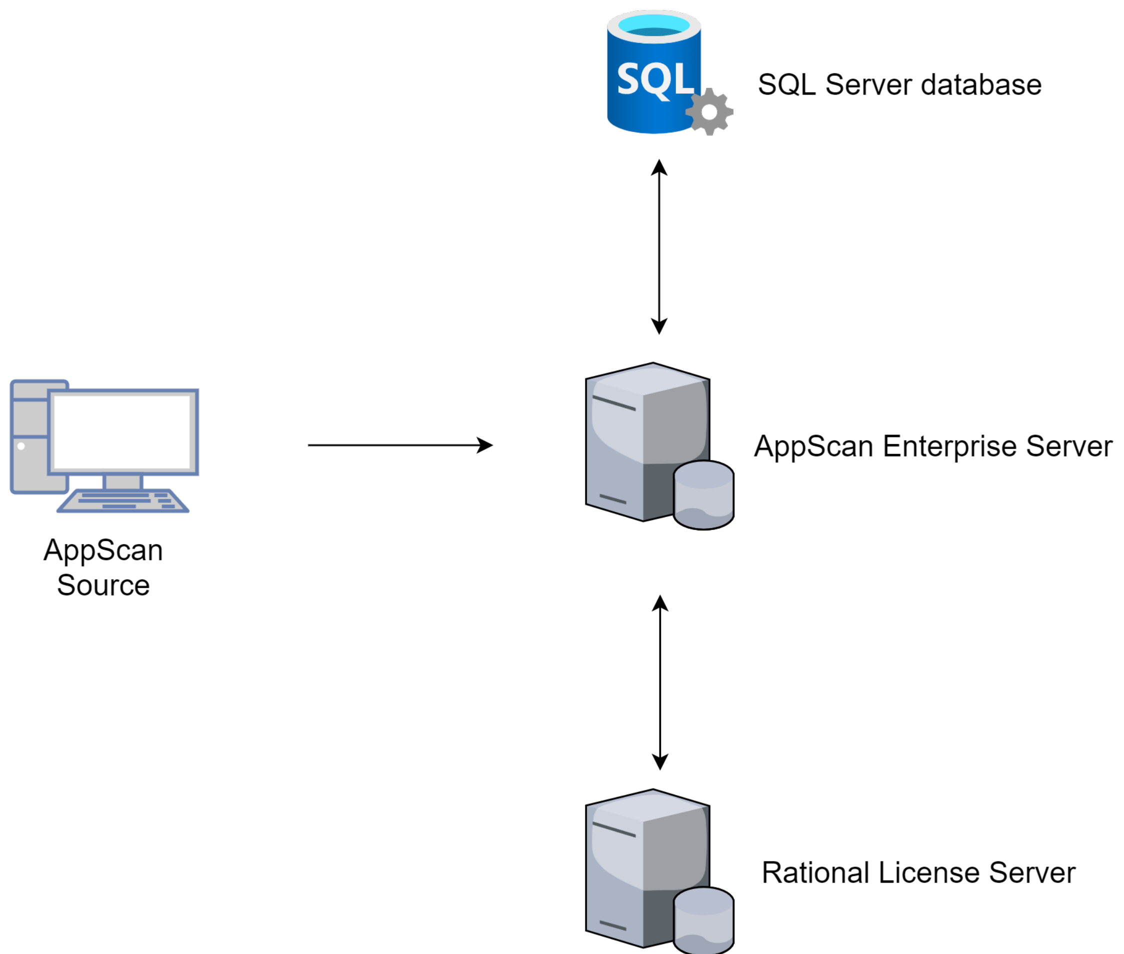

Figura 4.1: Topologia AppScan

In figura 4.1 è rappresentata una possibile topologia dell'architettura di AppScan.

4.2.4 Minikube

Minikube non richiede una specifica versione del sistema operativo, i requisiti necessari sono [63]:

- 2 CPU o più
- 2 GB di RAM
- 20 GB di spazio libero su disco
- una connessione a Internet

- un gestore dei container o una macchina virtuale, come ad esempio: Docker, Hyperkit, Hyper-V, KVM, Parallels, Podman, VirtualBox o VMWare

Capitolo 5

Analisi dei risultati

5.1 Damn Vulnerable Web Application

Per questo lavoro di tesi è stata analizzata una *Damn Vulnerable Web Application* (dvwa), ovvero un'applicazione web vulnerabile per costruzione scritta in PHP e MySQL. I suoi obiettivi sono quelli di aiutare i team di sicurezza a testare le proprie competenze e strumenti in un ambiente controllato e di permettere agli sviluppatori web di comprendere meglio i processi di sicurezza.
L'applicazione è stata scelta perché contiene le vulnerabilità più comuni e sono note ed enumerabili, in questo modo è possibile avere un riscontro dalle analisi di sicurezza automatizzate all'interno della pipeline; inoltre, è già presente il *Dockerfile*, un file di testo che contiene la lista delle dipendenze e dei comandi per costruire la relativa immagine Docker.
In questo lavoro di tesi è mancato il confronto delle tempistiche con altri applicativi perché per avere dei risultati comparabili sarebbe stato necessario ricercare un'altra applicazione open-source con gli stessi requisiti della dvwa, ovvero che contenesse delle vulnerabilità note. Non è stato neanche possibile confrontare i risultati con altre applicazioni dell'azienda Reply o dei suoi clienti perché sarebbero state inserite all'interno di una tesi pubblica le eventuali vulnerabilità presenti all'interno degli applicativi citati in precedenza, questo non è possibile per gli accordi stipulati tra il cliente e Reply.
Confrontare questa soluzione con un'applicazione più complessa avrebbe prodotto dei risultati non confrontabili perché ci sarebbero stati dei tempi di esecuzione diversi, totalmente dipendenti dalle capacità del *personal computer* messo a disposizione e non dalla metodologia applicata.

Durante la fase del deploy, presentata nella sezione 3.3.3, si è impostato l'URL a cui accedere all'applicativo, ovvero *http://dvwa.com:30002/*. Ogni volta che si crea un nuovo deployment GitLab offre al possibilità di tracciarlo e facilitare l'accesso

all'applicativo grazie alla finestra *Environments*.

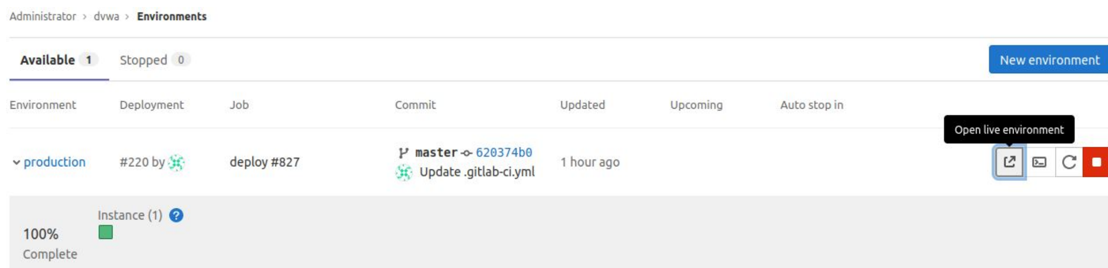

Figura 5.1: Environments

La figura 5.1 contiene tutti gli applicativi rilasciati in produzione ed è possibile verificarne l'integrità grazie alla funzione *Open live environment*. Cliccando sul seguente bottone si viene reindirizzati all'URL definito nella fase di deploy.
La schermata che appare è quella di *login*, le credenziali di accesso sono standard e sono *admin* e *password*, mentre l'homepage è presentata nell'immagine 5.2.

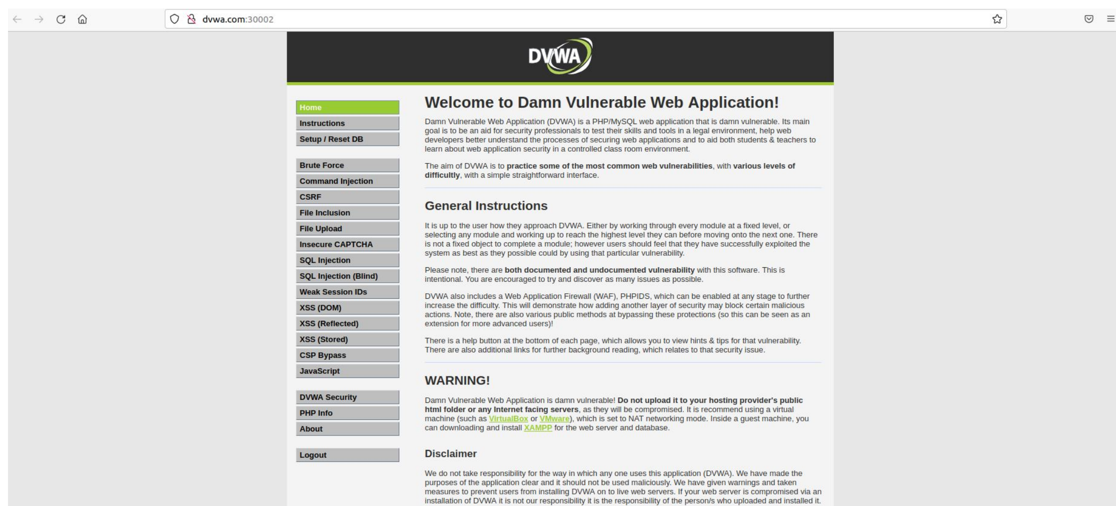

Figura 5.2: Homepage dvwa

5.2 Differenze con lo sviluppo tradizionale

La metodologia tradizionale di sviluppo software è strutturata in una sequenza lineare di fasi, in cui ognuna dipende dai risultati di quella precedente. La creazione di un nuovo prodotto può richiedere molto tempo; in generale, gli stadi da seguire e le relative tempistiche sono [64]:

- **Planning** da 2 a 4 settimane
- **Design** 2 settimane
- **Development** da 3 a 8 mesi
- **Implementazione** da 2 a 4 settimane
- **Testing e Manutenzione** da 3 a 6 settimane
- **Deployment** 3 settimane

Le tempistiche riportate non sono standard ma dipendono dal tipo di software che si sviluppa e dalla sua architettura. In questo lavoro di tesi è stata costruita una pipeline che automatizza una parte dell'implementazione, ovvero la build, la fase di test e il deployment.

Figura 5.3: Pipeline terminata di GitLab

In figura 5.3 è mostrato il risultato ottenuto.
Il repository di GitLab utilizzato contiene 357 file e 34821 righe di codice. Grazie agli strumenti utilizzati, la pipeline ci impiega 7 minuti e 02 secondi per eseguire la build, l'analisi statica, il deploy e i test sull'infrastruttura Kubernetes. Con questa soluzione automatizzata è possibile ripetere più volte l'esecuzione della pipeline senza doversi coordinare con altre persone e i feedback si ottengono prima, riducendo significativamente i tempi e la possibilità di commettere degli errori dovuti alle azioni manuali. In uno scenario reale, per rendere attive le modifiche è possibile impiegare una notevole quantità di tempo data dall'attività di coordinamento che non può essere misurata perché dipende dall'esperienza e dalle tecnologie messe a disposizione del team. Grazie alla ripetibilità e l'automazione della pipeline non è necessario coordinare la parte di sviluppo con quella operativa. Tuttavia, non è stato possibile stimare il risparmio di tempo in quanto non è stato condotto in parallelo uno sviluppo tradizionale da un team apposito, essendo l'unico a lavorare al progetto sono state fatte delle assunzioni, come quelle precedentemente citate, che potrebbero essere verificate in eventuali lavori futuri.
La configurazione della pipeline è valida indipendentemente dal tipo di progetto pertanto può essere riutilizzata con altri applicativi senza doverla ricreare tutte le volte da zero, però alcune modifiche sono necessarie perché bisogna considerare la configurazione dei tool e degli ambienti utilizzati, come ad esempio AppScan Source che dipende dal linguaggio dell'applicativo.

5.3 Vulnerabilità trovate

Analisi statica

```
Scan completed: Total files: 357 Total findings: 958 Total lines: 34821 vkloc: 27.311105 v-density: 3170.991643
Elapsed Time - 0 Hour(s) 3 Minute(s) 40 Second(s)
        ------------------
        Total Call Sites: 958
        Total Definitive Security Findings with High Severity: 103
        Total Definitive Security Findings with Medium Severity: 0
        Total Definitive Security Findings with Low Severity: 0
        Total Suspect Security Findings with High Severity: 16
        Total Suspect Security Findings with Medium Severity: 132
        Total Suspect Security Findings with Low Severity: 8
        Total Scan Coverage Findings with High Severity: 444
        Total Scan Coverage Findings with Medium Severity: 27
        Total Scan Coverage Findings with Low Severity: 221
        Total Lines: 34,821
        Max V-Density: 3,170.992
        Max V/kloc: 27.311
        V-Density: 3,170.992
        V/kloc: 27.311
```

Figura 5.4: Output di AppScan Source

L'immagine 5.4 è un estratto dell'output di AppScan Source. Si può notare che l'analisi statica è stata completata in 3 minuti e 40 secondi e sono state trovate 958 vulnerabilità con tre gradi di pericolosità diversi: *High*, *Medium* e *Low*. Un altro dato che riporta la figura 5.4 è il *vkloc*, ovvero il numero di vulnerabilità per 1000 linee di codice scansionato, mentre la *v-density* mette in relazione il numero e la criticità delle vulnerabilità trovate con la dimensione del progetto analizzato. Utilizzando un'applicazione vulnerabile *by design* questi numeri risultano essere elevati.
Le vulnerabilità trovate devono essere analizzate dagli esperti di sicurezza informatica al fine di filtrare eventuali falsi positivi e identificare con più accuratezza gli interventi da eseguire per la messa in sicurezza dell'applicativo e dell'ambiente sul quale è stato rilasciato.

AppScan Enterprise

AppScan Enterprise raccoglie, in un unico posto, le vulnerabilità trovate da tutti gli strumenti di sicurezza utilizzati, compresi quelli di Kubernetes. La schermata principale è mostrata in figura 5.5.

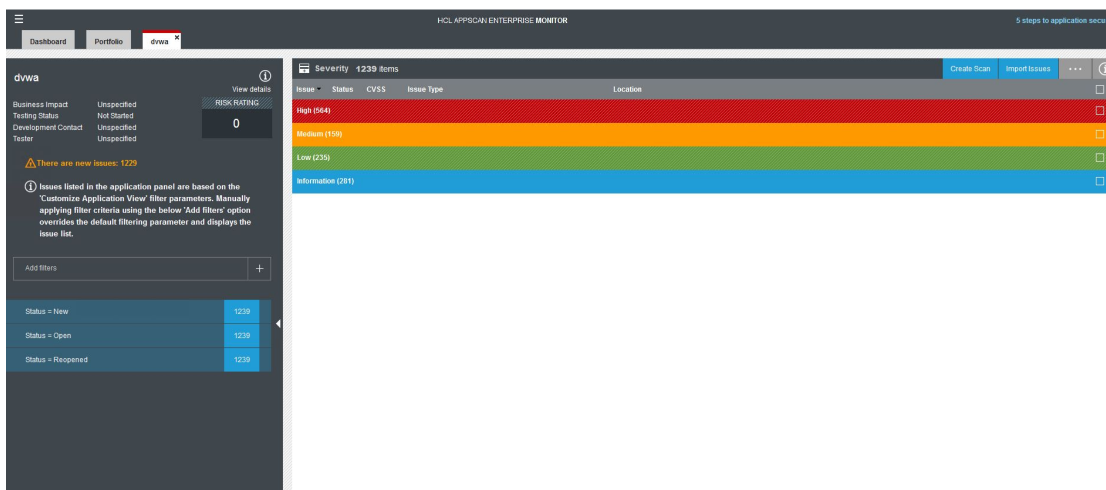

Figura 5.5: AppScan Enterprise

L'interfaccia presenta in modo chiaro la suddivisione delle vulnerabilità trovate, ogni riga contiene un punteggio CVSS [65], ovvero il sistema di valutazione delle vulnerabilità comuni utilizzato per comprendere la gravità del problema mediante un punteggio numerico, la categoria a cui appartiene la vulnerabilità e il file che la contiene.
Le righe possono essere espanse per vedere l'elenco delle singole criticità, come mostrato in figura 5.6.

Issue	Status	CVSS	Issue Type	Location
High (564)				
1915	New	9.7	Injection.SQL	C:\Users\Luca\Desktop\dvwa\dvwa\login.php(27)
1913	New	9.7	Injection.SQL	C:\Users\Luca\Desktop\dvwa\dvwa\dvwa\includes\DBMS\MySQL.php(36)
1912	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier.safe-includes.php(124)
1911	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier\LanguageFactory.php(156)
1910	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier.safe-includes.php(101)
1907	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\dvwa\includes\dvwaPage.inc.php(55)
1906	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier.safe-includes.php(106)
1905	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier.safe-includes.php(43)
1903	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier.safe-includes.php(147)
1902	New	0	Validation.Required	C:\Users\Luca\Desktop\dvwa\dvwa\vulnerabilities\brute\source \impossible.php(8)
1898	New	7.5	CrossSiteScripting	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier\Lexer\DOMLex.php(62)
1897	New	7.5	CrossSiteScripting	C:\Users\Luca\Desktop\dvwa\dvwa\dvwa\includes\dvwaPhpIds.inc.php(97)
1896	New	9.7	Injection.SQL	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\tests \IDS\MonitorTest.php(950)
1895	New	9.7	Injection.SQL	C:\Users\Luca\Desktop\dvwa\dvwa\login.php(34)
1894	New	9.7	Injection.SQL	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\tests \IDS\MonitorTest.php(960)
1893	New	6.4	Path Traversal	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\tests \IDS\InitTest.php(23)
1891	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\external\phpids\0.6\lib\IDS\vendors \htmlpurifier\HTMLPurifier.safe-includes.php(178)
1888	New	7.5	Injection	C:\Users\Luca\Desktop\dvwa\dvwa\vulnerabilities\brute\index.php(4)

Figura 5.6: Vulnerabilità con criticità elevata

È possibile ottenere una vista più dettagliata delle singole vulnerabilità cliccando sul contatore progressivo posto all'inizio di ogni riga. Le principali informazioni presenti sono relative alla causa che genera la vulnerabilità, il rischio associato, il file che la contiene e la funzione o chiamata critica, come mostrato in figura 5.7.

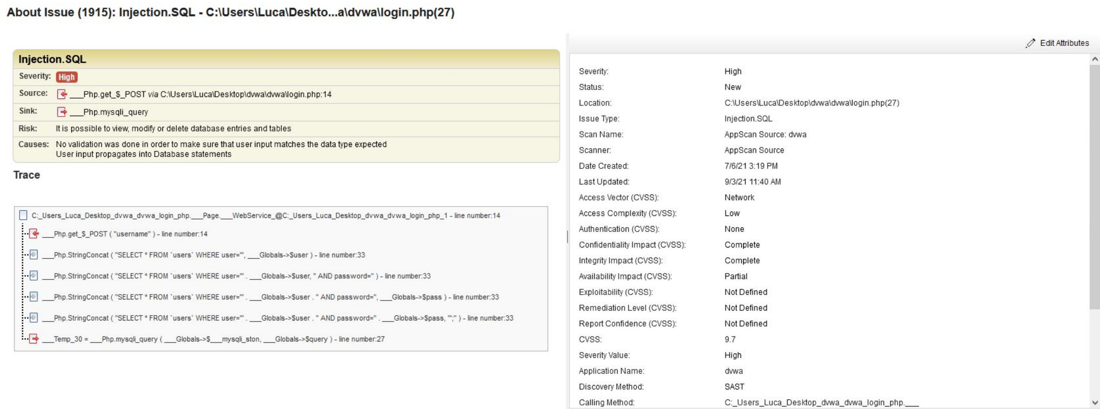

Figura 5.7: Vista di una singola vulnerabilità

Issue	Status	CVSS	Issue Type	Location
Information (281)				
2197	New		kube-bench	Remediation: Ensure that namespaces are created to allow for appropriate segregation of Kubernetes resources and that all new resources are created in
2196	New		kube-bench	Remediation: Follow the Kubernetes documentation and apply security contexts to your pods. For a suggested list of security contexts, you may refer to the CIS
2195	New		kube-bench	Remediation: Seccomp is an alpha feature currently. By default, all alpha features are disabled. So, you would need to enable alpha features in the
2194	New		kube-bench	Remediation: Follow the documentation and create namespaces for objects in your deployment as you need them.
2193	New		kube-bench	Remediation: Follow the Kubernetes documentation and setup image provenance.
2192	New		kube-bench	Remediation: Refer to the secrets management options offered by your cloud provider or a third-party secrets management solution.
2191	New		kube-bench	Remediation: if possible, rewrite application code to read secrets from mounted secret files, rather than from environment variables.
2190	New		kube-bench	Remediation: Follow the documentation and create NetworkPolicy objects as you need them.
2189	New		kube-bench	Remediation: If the CNI plugin in use does not support network policies, consideration should be given to making use of a different plugin, or finding an
2188	New		kube-bench	Remediation: Review the use of capabilites in applications running on your cluster. Where a namespace contains applicaions which do not require any
2187	New		kube-bench	Remediation: Ensure that allowedCapabilities is not present in PSPs for the cluster unless it is set to an empty array.
2186	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.requiredDropCapabilities is set to include either
2185	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.runAsUser.rule is set to either MustRunAsNonRoot or
2184	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.allowPrivilegeEscalation field is omitted or set to false.
2183	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.hostNetwork field is omitted or set to false.
2182	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.hostIPC field is omitted or set to false.
2181	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.hostPID field is omitted or set to false.
2180	New		kube-bench	Remediation: Create a PSP as described in the Kubernetes documentation, ensuring that the .spec.privileged field is omitted or set to false.

Figura 5.8: Vista Information

Come mostrato in figura 5.8, la severità dei risultati contenuti nei report di Kubernetes viene classificata come *Information*, ovvero come delle informazioni

che descrivono le tecnologie, le caratteristiche architetturali o i meccanismi di sicurezza utilizzati nel codice. Tuttavia, siccome gli strumenti di Kubernetes non sono nativamente compatibili con AppScan Enterprise, le vulnerabilità devono essere analizzate manualmente dal team di sicurezza.
In figura sono mostrate le vulnerabilità trovate dagli strumenti Kubernetes, ogni riga contiene il nome dello strumento di sicurezza utilizzato e un messaggio utile per comprendere meglio il problema.

Capitolo 6

Limitazioni e lavori futuri

6.1 Limitazioni

Il lavoro di tesi è stato sviluppato all'interno di un'architettura locale realizzata su un *personal computer* caratterizzato da un hardware limitato, questo rappresenta una limitazione in termini di performance perché si è inserito un server GitLab e un cluster Kubernetes dentro una macchina virtuale riducendo le performance; utilizzando un progetto più complesso si sarebbe generato un rallentamento della macchina.
Un'altra limitazione legata all'aver utilizzato un'architettura locale è stata la necessità di creare una *Certification Authority* e dei certificati autofirmati perché AppScan non li riconosce come attendibili.
All'interno dello script 3.10 le richieste fatte al server tramite le API REST utilizzano l'opzione *verify=false*, pertanto si accetta qualsiasi tipo di certificato presentato rendendo l'applicazione vulnerabile all'attacco *Man-in-the-middle*, ovvero l'intercettazione e la manipolazione del traffico di rete. Tuttavia, disattivare il controllo dei certificati è una pratica comune durante la fase di sviluppo locale di un applicativo perché si lavora all'interno di un ambiente protetto.
La stessa procedura è stata adottata anche all'interno dello script 3.7; infatti, nella prima istruzione di *login* si è utilizzata l'opzione *acceptssl* che accetta automaticamente qualsiasi certificato SSL fornito.

Un'ulteriore limitazione è rappresentata dal file di progetto *.ppf* di AppScan Source. Le applicazioni e i progetti di AppScan hanno file corrispondenti che conservano le informazioni di configurazione necessarie per la scansione. Questi file risiedono nella stessa *directory* del codice sorgente, perché le informazioni necessarie per compilare i progetti, come le dipendenze, sono simili a quelle richieste da AppScan Source per scansionarli correttamente. I file vengono generati manualmente quando si crea e si configura un'applicazione o un progetto perché non è possibile

automatizzare tramite lo script il processo di creazione di tali file.

Per adottare AppScan Enterprise come strumento di monitoraggio è necessario spendere del tempo per rendere compatibili i report delle analisi sull'ambiente Kubernetes con i formati di AppScan, questo rappresenta una limitazione in termini di automazione perché non è possibile eseguire sin da subito la pipeline; pertanto, è stato creato uno script che svolge tale funzione.

6.2 Lavori futuri

Il lavoro di tesi è stato un primo tentativo di automatizzare una serie di test di sicurezza, pertanto, quando l'analisi statica rileva delle vulnerabilità critiche, il processo non si interrompe ma si prosegue in modo da eseguire i controlli sull'infrastruttura Kubernetes per verificare il corretto funzionamento dei tool. La pipeline sviluppata è un ottimo punto di partenza per i molteplici lavori futuri.
Una possibile evolutiva che può essere integrata consiste nell'interrompere l'esecuzione del processo non appena vengono rilevate delle vulnerabilità con criticità elevata e notificare lo sviluppatore in modo che possa apportare le opportune modifiche al codice.
Un approccio simile potrebbe essere quello di utilizzare un doppio ambiente rilasciando il codice in quello di *staging* e se i controlli di sicurezza rilevano delle vulnerabilità critiche si interrompe il processo notificando lo sviluppatore, in caso contrario si prosegue con il deploy dell'applicativo in un ambiente di produzione.

In questo lavoro di tesi non è stato possibile misurare con precisione il risparmio di tempo perché non era presente un team di sviluppo che eseguisse manualmente tutte le attività della pipeline. Per validare le assunzioni fatte nel capitolo 5 è auspicabile ripetere questo esperimento utilizzando entrambi gli approcci e testandoli su diverse soluzioni, in questo modo è possibile stimare in maniera oggettiva il guadagno di tempo.

In un'ottica di business reale è possibile evolvere l'infrastruttura Kubernetes sostituendo il cluster locale minikube con un'infrastruttura in cloud in modo da poter gestire più nodi e ottenere un'affidabilità maggiore. In questo modo si risolverebbe il problema dei certificati autofirmati perché si utilizzerebbero quelli autentici che possono essere convalidati da AppScan, migliorando così il livello di sicurezza complessivo.

La pipeline ottenuta può essere migliorata integrando al suo interno altri strumenti di sicurezza quali:

- **Analisi delle dipendenze** [66]: è una pratica di sicurezza ampiamente utilizzata dalle aziende per rilevare le vulnerabilità note contenute nelle dipendenze di un progetto.

- **Dynamic Application Security Testing (DAST)**: è una tipologia di test *black-box*, ovvero senza accedere direttamente al codice, che simula degli attacchi cercando le vulnerabilità e controllando le interfacce dell'applicazione mentre questa è in esecuzione in un ambiente di test.
- **Interactive Application Security Testing (IAST)** [67]: queste tecniche testano l'applicazione in tempo reale mentre è in esecuzione in un ambiente di controllo qualità o test. IAST in genere viene implementato utilizzando dei sensori che osservano il funzionamento dell'applicazione e analizzano il flusso di traffico per identificare le vulnerabilità. IAST può essere facilmente integrato nella pipeline CI/CD, è altamente scalabile e può essere automatizzato.

Capitolo 7

Conclusioni

L'obiettivo della tesi era quello di integrare dei controlli di sicurezza, eseguiti con strumenti professionali adottati da numerose aziende, all'interno di una pipeline DevOps gestendo l'analisi statica del codice e dell'ambiente Kubernetes usato per il deploy.
Si è utilizzato GitLab come repository del codice e strumento di CI/CD. Sono stati testati diversi tipi di executor, in particolare ssh, Docker e shell.
Il primo presentava delle limitazioni legate alla impossibilità di aprire più connessioni ssh e di sicurezza perché username e password dell'utente venivano salvate in chiaro. Docker richiedeva una configurazione complessa perché, utilizzando dei certificati autofirmati, non si riusciva ad autenticare correttamente le istanze coinvolte. Inoltre il cluster Kubernetes appartiene a una sotto-rete differente rispetto a GitLab pertanto si sono verificati dei problemi di raggiungibilità.
L'executor adottato è stato quello shell perché oltre ad essere semplice da configurare è possibile accedere a tutti i programmi installati sulla macchina Linux ed utilizzare degli script esterni salvati sul filesystem.

Grazie al file *.gitlab-ci.yml* è stato definito il flusso di esecuzione della pipeline. La prima operazione è la compilazione del codice e la creazione di un'immagine Docker dell'applicativo che viene utilizzata nelle fasi successive.
Si prosegue con l'analisi statica del codice con AppScan Source per rilevare le vulnerabilità presenti e si effettua il rilascio in un ambiente di produzione.
L'ultima operazione effettuata sono dei controlli sull'infrastruttura Kubernetes con kube-bench, kube-hunter, kubeaudit e KubeSec per verificare la correttezza delle configurazioni e l'integrità dell'ambiente.
Grazie a questo lavoro di tesi è stato creato un metodo per poter includere tutte le vulnerabilità in un ambiente centralizzato, ovvero AppScan Enterprise, permettendo il monitoraggio in tempo reale.

La soluzione progettata può essere facilmente integrata all'interno di un processo DevSecOps aiutando gli sviluppatori a ricevere i feedback in un tempo minore

perché non è necessario coordinarsi con altri membri del team.

Appendice A

Manifest

In questa appendice vengono mostrati i manifest utilizzati per il deployment, la configurazione dell'ambiente Kubernetes e l'installazione di kube-bench, kube-hunter e kubeaudit.

A.1 Deployment.yml

Il codice mostrato nel riquadro A.1 riporta il file YAML necessario per eseguire il deployment dell'applicativo utilizzando l'immagine Docker costruita nella fase di build.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: dvwa-deployment
  annotations:
    app.gitlab.com/app: root-dvwa
    app.gitlab.com/env: production
spec:
  replicas: 1
  selector:
    matchLabels:
      app: dvwa
  template:
    metadata:
      labels:
        app: dvwa
      annotations:
        app.gitlab.com/app: root-dvwa
        app.gitlab.com/env: production
```

```yaml
    spec:
      containers:
      - name: dvwa
        image: gitlab.example.com:5050/root/dvwa/image:latest
        ports:
        - containerPort: 80
          name: hello-port
          protocol: TCP
      imagePullSecrets:
        - name: gitlab.example
```

Figura A.1: Deployment.yml

A.2 Service.yml

Il codice mostrato nel riquadro A.2 riporta il file YAML che crea un *Service* con il compito di esporre l'applicativo in esecuzione all'interno di una Pod come servizio di rete.

```yaml

apiVersion: v1
kind: Service
metadata:
  name: dvwa-service
  labels:
    app: dvwas
    ref: __CI_ENVIRONMENT_SLUG__
spec:
  type: NodePort
  selector:
    app: dvwa
  ports:
    - port: 80
      nodePort: 30002
      protocol: TCP
      targetPort: 80
```

Figura A.2: Service.yml

A.3 Ingress.yml

Il codice mostrato nel riquadro A.3 riporta il file YAML che crea un oggetto *Ingress* che ha il compito di gestire gli accessi al Service precedentemente creato tramite il protocollo HTTP.

```

apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: dvwa-ingress
  labels:
    app: dvwa
    ref: __CI_ENVIRONMENT_SLUG__
spec:
  rules:
  - host: gitlab.example.com
    http:
        paths:
        - path: /
          backend:
            serviceName: dvwa-service
            servicePort: 30002
```

Figura A.3: Ingress.yml

A.4 kube-bench.yaml

Il codice mostrato nel riquadro A.4 riporta il file YAML utilizzato per installare kube-bench all'interno di una pod. Alla riga 16 è presentato il comando utilizzato per lo svolgimento delle analisi ovvero:

kube-bench run control-plane master –json

Tramite questa istruzione si esegue la scansione del nodo master e del *Control Plane* di Kubernetes, ovvero l'insieme di componenti incaricati di gestire i nodi. Il flag *–json* indica il formato del report desiderato.

```

apiVersion: batch/v1
kind: Job
metadata:
  name: kube-bench
```

```
6  spec:
7    template:
8      metadata:
9        labels:
10         app: kube-bench
11     spec:
12       hostPID: true
13       containers:
14         - name: kube-bench
15           image: aquasec/kube-bench:latest
16           command: ["kube-bench","run", "control-plane,master", "--
   json"]
17           volumeMounts:
18             - name: var-lib-etcd
19               mountPath: /var/lib/etcd
20               readOnly: true
21             - name: var-lib-kubelet
22               mountPath: /var/lib/kubelet
23               readOnly: true
24             - name: var-lib-kube-scheduler
25               mountPath: /var/lib/kube-scheduler
26               readOnly: true
27             - name: var-lib-kube-controller-manager
28               mountPath: /var/lib/kube-controller-manager
29               readOnly: true
30             - name: etc-systemd
31               mountPath: /etc/systemd
32               readOnly: true
33             - name: lib-systemd
34               mountPath: /lib/systemd/
35               readOnly: true
36             - name: srv-kubernetes
37               mountPath: /srv/kubernetes/
38               readOnly: true
39             - name: etc-kubernetes
40               mountPath: /etc/kubernetes
41               readOnly: true
42             - name: usr-bin
43               mountPath: /usr/local/mount-from-host/bin
44               readOnly: true
45             - name: etc-cni-netd
46               mountPath: /etc/cni/net.d/
47               readOnly: true
48             - name: opt-cni-bin
49               mountPath: /opt/cni/bin/
50               readOnly: true
51       restartPolicy: Never
52       volumes:
53         - name: var-lib-etcd
```

```
54          hostPath:
55            path: "/var/lib/etcd"
56        - name: var-lib-kubelet
57          hostPath:
58            path: "/var/lib/kubelet"
59        - name: var-lib-kube-scheduler
60          hostPath:
61            path: "/var/lib/kube-scheduler"
62        - name: var-lib-kube-controller-manager
63          hostPath:
64            path: "/var/lib/kube-controller-manager"
65        - name: etc-systemd
66          hostPath:
67            path: "/etc/systemd"
68        - name: lib-systemd
69          hostPath:
70            path: "/lib/systemd"
71        - name: srv-kubernetes
72          hostPath:
73            path: "/srv/kubernetes"
74        - name: etc-kubernetes
75          hostPath:
76            path: "/etc/kubernetes"
77        - name: usr-bin
78          hostPath:
79            path: "/usr/bin"
80        - name: etc-cni-netd
81          hostPath:
82            path: "/etc/cni/net.d/"
83        - name: opt-cni-bin
84          hostPath:
85            path: "/opt/cni/bin/"
```

Figura A.4: kube-bench.yaml

A.5 kube-hunter.yaml

Il codice mostrato nel riquadro A.5 riporta il file YAML utilizzato per installare kube-hunter all'interno di una pod. Alle righe 12 e 13 è presentato il comando utilizzato per lo svolgimento delle analisi e i relativi argomenti ovvero:

kube-hunter –pod –report json

Quando kube-hunter viene eseguito con il flag *–pod* utilizza il token contenuto all'interno della pod per autenticarsi ai Service analizzati durante la scansione. L'argomento *report –json* indica il formato del report desiderato.

```
1
2 apiVersion: batch/v1
3 kind: Job
4 metadata:
5   name: kube-hunter
6 spec:
7   template:
8     spec:
9       containers:
10        - name: kube-hunter
11          image: aquasec/kube-hunter
12          command: ["kube-hunter"]
13          args: ["--pod", "--report","json"]
14      restartPolicy: Never
15  backoffLimit: 4
```

Figura A.5: kube-hunter.yaml

A.6 kubeaudit.yaml

Il codice mostrato nel riquadro A.6 riporta il file YAML utilizzato per installare kubeaudit all'interno di una pod.

```
1
2 apiVersion: v1
3 kind: ServiceAccount
4 metadata:
5   name: kubeaudit
6   namespace: default
7
8 ---
9
10 apiVersion: batch/v1
11 kind: Job
12 metadata:
13   name: kubeaudit
14   namespace: default
15 spec:
16   template:
17     metadata:
18       annotations:
19         container.apparmor.security.beta.kubernetes.io/kubeaudit:
     runtime/default
```

```
20          seccomp.security.alpha.kubernetes.io/pod: runtime/default
21      spec:
22        serviceAccountName: kubeaudit
23        restartPolicy: OnFailure
24        containers:
25          - name: kubeaudit
26            image: shopify/kubeaudit:v0.11
27            args: ["all", "--exitcode", "0"]
28            securityContext:
29              allowPrivilegeEscalation: false
30              capabilities:
31                drop: ["all"]
32              privileged: false
33              readOnlyRootFilesystem: true
34              runAsNonRoot: true
```

Figura A.6: kubeaudit.yaml

Bibliografia

[1] RAVI TEJA YARLAGADDA. *How Public Sectors Can Adopt the DevOps Practices to Enhance the System.* URL: https://poseidon01.ssrn.com/delivery.php?ID=907117081025014118101023003066127072026032046009065078108120101067127126070084007025032052096126039015001118017020089006099119042057064008052097084088010119004087093046043005069082066064020073083094115001076096022028087007100116073116022064081086074118&EXT=pdf&INDEX=TRUE (visitato il 09/2018) (cit. a p. 8).

[2] Shannon Lietz. *What is DevSecOps?* URL: https://www.devsecops.org/blog/2015/2/15/what-is-devsecops (visitato il 01/06/2015) (cit. a p. 8).

[3] Laura Zanotti. *Che cosa significa DevOps e perché oggi è un fondamentale della programmazione.* URL: https://www.zerounoweb.it/techtarget/searchdatacenter/programmazione-software-e-l-ora-degli-it-shop-devops/ (visitato il 11/12/2020) (cit. a p. 11).

[4] *Software development.* URL: https://en.wikipedia.org/wiki/Software_development (cit. a p. 11).

[5] *DevOps and Security Glossary Terms.* URL: https://www.sumologic.com/glossary/it-operations/ (cit. a p. 11).

[6] *Cos'è DevOps?* URL: https://aws.amazon.com/it/devops/what-is-devops/ (cit. a p. 11).

[7] *Che cos'è DevOps?* URL: https://azure.microsoft.com/it-it/overview/what-is-devops/#devops-overview (cit. alle pp. 11, 19).

[8] RedHat. *I vantaggi della metodologia DevSecOps.* URL: https://www.redhat.com/it/topics/devops/what-is-devsecops (cit. a p. 12).

[9] *Che cos'è DevSecOps e a cosa serve?* URL: https://www.ionos.it/digitalguide/server/sicurezza/che-cose-devsecops/ (visitato il 11/08/2020) (cit. a p. 12).

[10] Marko Anastasov. *What's the Difference Between Continuous Integration, Continuous Deployment and Continuous Delivery?* URL: https://semaphoreci.com/blog/2017/07/27/what-is-the-difference-between-continuous-integration-continuous-deployment-and-continuous-delivery.html (visitato il 27/07/2017) (cit. a p. 12).

[11] RedHat. *Cosa si intende con CI/CD?* URL: https://www.redhat.com/it/topics/devops/what-is-ci-cd (cit. a p. 13).

[12] RedHat. *What is a CI/CD pipeline?* URL: https://www.redhat.com/en/topics/devops/what-cicd-pipeline (cit. a p. 14).

[13] JakobTheDev. *The Eight Phases of a DevOps Pipeline.* URL: https://medium.com/taptuit/the-eight-phases-of-a-devops-pipeline-fda53ec9bba (visitato il 18/07/2019) (cit. a p. 15).

[14] *Threat Modeling.* URL: https://www.synopsys.com/glossary/what-is-threat-modeling.html (cit. a p. 15).

[15] Filiberto Santoro. *Penetration test, cos'è, come funziona e a che serve.* URL: https://www.cybersecurity360.it/soluzioni-aziendali/penetration-test-cose-come-funziona-e-a-che-serve/#Cose_il_penetration_test_o_pen_test (visitato il 22/05/2018) (cit. a p. 16).

[16] Programmazione del sito web. *Panoramica dei migliori strumenti di Continuous Integration continua (CI).* URL: https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/strumenti-di-continuous-integration/ (visitato il 19/03/2020) (cit. a p. 16).

[17] Jenkins. *Jenkins.* URL: https://www.jenkins.io/ (cit. a p. 16).

[18] *Cos'è Jenkins.* URL: https://www.geekandjob.com/wiki/jenkins (cit. a p. 16).

[19] GitHub. *Where the world builds software.* URL: https://github.com/ (cit. a p. 17).

[20] *GitHub.* URL: https://en.wikipedia.org/wiki/GitHub (cit. a p. 17).

[21] *Che cos'è la funzionalità GitHub Actions per Azure?* URL: https://docs.microsoft.com/it-it/azure/developer/github/github-actions (visitato il 30/10/2020) (cit. a p. 17).

[22] GitLab. *GitLab.* URL: https://about.gitlab.com/ (cit. a p. 17).

[23] Micha Hernandez van Leuffen. *The Many Problems with Jenkins and Continuous Delivery.* URL: https://thenewstack.io/many-problems-jenkins-continuous-delivery (visitato il 05/06/2017) (cit. a p. 17).

[24] *GitLab vs GitHub Continuous Integration Comparison*. URL: https://about.gitlab.com/devops-tools/github-vs-gitlab/ci-missing-github-capabilities/ (cit. a p. 18).

[25] Sergio Monteleone. *CI/CD con GitLab*. URL: https://www.html.it/articoli/ci-cd-con-gitlab/ (visitato il 03/12/2018) (cit. a p. 18).

[26] *What is Auto DevOps?* URL: https://about.gitlab.com/stages-devops-lifecycle/auto-devops/ (cit. a p. 18).

[27] *Kubernetes + GitLab Everything you need to build, test, deploy, and run your app at scale*. URL: https://about.gitlab.com/solutions/kubernetes/ (cit. a p. 18).

[28] *Orchestrazione di Container in produzione*. URL: https://kubernetes.io/it/ (cit. a p. 19).

[29] RedHat. *CONTAINER I vantaggi di Kubernetes*. URL: https://www.redhat.com/it/topics/containers/what-is-kubernetes (cit. a p. 19).

[30] *Cos'è Kubernetes?* URL: https://kubernetes.io/it/docs/concepts/overview/what-is-kubernetes/ (cit. a p. 19).

[31] *Concetti*. URL: https://kubernetes.io/it/docs/concepts/ (cit. a p. 19).

[32] *Security AppScan*. URL: https://en.wikipedia.org/wiki/Security_AppScan (cit. a p. 19).

[33] *HCL AppScan on Cloud*. URL: https://www.hcltechsw.com/it/products/appscan/offerings/asoc (cit. a p. 20).

[34] *HCL AppScan Enterprise*. URL: https://www.hcltechsw.com/it/products/appscan/offerings/enterprise (cit. a p. 20).

[35] *HCL AppScan Standard*. URL: https://www.hcltechsw.com/it/products/appscan/offerings/standard (cit. a p. 20).

[36] *HCL AppScan Source*. URL: https://www.hcltechsw.com/it/products/appscan/offerings/source (cit. a p. 20).

[37] *Macchina virtuale (VM)*. URL: https://www.geekandjob.com/wiki/macchina-virtuale-vm (cit. a p. 22).

[38] *Secure Shell*. URL: https://it.wikipedia.org/wiki/Secure_Shell (cit. a p. 22).

[39] Laura Zanotti. *SSH – Secure Socket Shell: definizione e a che cosa serve*. URL: https://www.zerounoweb.it/techtarget/searchsecurity/ssh-definizione-e-a-che-cosa-serve/ (visitato il 20/11/2020) (cit. a p. 22).

[40] Giovanni Sacheli. *Chiavi SSH Guida Completa*. URL: https://www.evemilano.com/guida-chiavi-ssh/#3 (visitato il 11/01/2020) (cit. a p. 23).

[41] *Tutorial di GitLab: installazione e primi passi.* URL: https://www.ionos.it/digitalguide/siti-web/programmazione-del-sito-web/tutorial-di-gitlab/ (visitato il 21/10/2020) (cit. a p. 23).

[42] *Proteggere il sito con il protocollo HTTPS.* URL: https://developers.google.com/search/docs/advanced/security/https?hl=it (visitato il 12/07/2021) (cit. a p. 23).

[43] Giovanni Sacheli. *Chiavi SSH Guida Completa.* URL: https://www.evemilano.com/guida-chiavi-ssh/#2 (visitato il 11/01/2020) (cit. a p. 23).

[44] André Miranda. *Tutorial: Securing your GitLab Pages with TLS and Let's Encrypt.* URL: https://about.gitlab.com/blog/2016/04/11/tutorial-securing-your-gitlab-pages-with-tls-and-letsencrypt/ (visitato il 04/11/2016) (cit. a p. 24).

[45] *SSL/TLS certificates.* URL: https://docs.gitlab.com/ee/user/project/pages/custom_domains_ssl_tls_certification/ssl_tls_concepts.html (cit. a p. 24).

[46] *Certificate authority.* URL: https://it.wikipedia.org/wiki/Certificate_authority (cit. a p. 24).

[47] *GitLab Runner.* URL: https://docs.gitlab.com/runner/ (cit. a p. 24).

[48] *Executors.* URL: https://docs.gitlab.com/runner/executors/ (cit. a p. 25).

[49] *Minikube: Kubernetes alla massima portata con il minimo sforzo.* URL: https://www.ionos.it/digitalguide/server/tools-o-strumenti/kubernetes-minikube/ (visitato il 20/10/2020) (cit. a p. 26).

[50] *Che cos'è il file hosts e come lo si modifica?* URL: https://www.ionos.it/digitalguide/server/configurazione/file-hosts/ (visitato il 12/09/2019) (cit. a p. 27).

[51] Lorenzo Testa. *Conosciamo GitLab pipeline come strumento di Continuous Integration.* URL: https://www.intre.it/2020/07/03/gitlab-pipeline-strumento-continuous-integration/ (visitato il 03/07/2020) (cit. a p. 27).

[52] Shadi Namrouti. *Staging environment vs Production environment.* URL: https://softwareengineering.stackexchange.com/questions/117945/staging-environment-vs-production-environment/385029#385029 (visitato il 06/01/2019) (cit. a p. 31).

[53] *kube-bench.* URL: https://github.com/aquasecurity/kube-bench (cit. a p. 35).

[54] *kube-hunter.* URL: https://kube-hunter.aquasec.com/ (cit. a p. 35).

[55] *kubesec.* URL: https://github.com/controlplaneio/kubesec (cit. a p. 35).

[56] *kubeaudit*. URL: https://github.com/Shopify/kubeaudit (cit. a p. 35).

[57] *Privilege escalation*. URL: https://it.wikipedia.org/wiki/Privilege_escalation (cit. a p. 35).

[58] *Amazon Web Services: cos'è e come funziona?* URL: https://www.softwave-soltec.it/amazon-web-service-cose-e-come-funziona/ (visitato il 05/11/2020) (cit. a p. 40).

[59] *Installation requirements*. URL: https://docs.gitlab.com/ee/install/requirements.html (cit. a p. 41).

[60] Giuseppe Maggi. *Redis: un DBMS NoSQL a dizionario*. URL: https://www.html.it/articoli/redis/ (visitato il 19/12/2014) (cit. a p. 41).

[61] *Windows system requirements*. URL: https://help.hcltechsw.com/appscan/Source/9.0.3/topics/system_requirements_win.html (cit. a p. 42).

[62] *Hardware and software requirements*. URL: https://help.hcltechsw.com/appscan/Enterprise/9.0.3/topics/r_sysreqs_control_center.html (cit. a p. 43).

[63] *minikube start*. URL: https://minikube.sigs.k8s.io/docs/start/ (cit. a p. 44).

[64] DJ Wardynski. *How Long Does it Take to Build Custom Software for a Business?* URL: https://www.brainspire.com/blog/how-long-does-it-take-to-build-custom-software-for-a-business (visitato il 14/09/2017) (cit. a p. 47).

[65] Bernhard Zwickler. *Sistema di valutazione delle vulnerabilità comuni (CVSS): cos'è, come funziona, gli sviluppi futuri*. URL: https://www.cybersecurity360.it/soluzioni-aziendali/sistema-di-valutazione-delle-vulnerabilita-comuni-cvss-cose-come-funziona-gli-sviluppi-futuri/ (visitato il 04/03/2020) (cit. a p. 50).

[66] *OWASP Dependency-Check*. URL: https://owasp.org/www-project-dependency-check/ (cit. a p. 54).

[67] Julie Peterson. *All About IAST – Interactive Application Security Testing*. URL: https://www.whitesourcesoftware.com/resources/blog/iast-interactive-application-security-testing/#What_Is_IAST (visitato il 16/07/2020) (cit. a p. 55).