UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Dipartimento di Ingegneria Informatica e dell'Automazione

Corso di Laurea in Ingegneria Informatica e dell'Automazione

Sfruttamento di una vulnerabilità di ArgoCD per la raccolta di informazioni sensibili

Exploiting a vulnerability in ArgoCD to collect sensitive information

Relatore:

Prof. Spalazzi Luca

Tesi a cura di:

Sbattella Mattia

A.A. 2022/23

Indice

1 Introduzione

1.1 Ambito e motivazioni

> *"La cyber security (anche detta sicurezza informatica) consiste nell'insieme di tecnologie, processi e misure di protezione progettate per ridurre il rischio di attacchi informatici."* [7]

La cybersecurity, nel contesto odierno, riveste un ruolo cruciale nella protezione delle infrastrutture digitali. In particolare, il **penetration testing** emerge come uno strumento fondamentale per valutare la sicurezza di un sistema identificandone le vulnerabilità.

Questa pratica consiste nell'esecuzione di simulazioni di attacchi informatici al fine di identificare, replicare ed in fine correggere le vulnerabilità presenti nei sistemi interessati. In questo modo, attraverso l'applicazione di tecniche di ethical hacking[1], il penetration testing si propone di valutare l'efficacia delle misure di sicurezza esistenti e di migliorare la resistenza di un sistema alle minacce informatiche.

Nel panorama delle infrastrutture cloud e nell'ambito dell'utilizzo sempre più diffuso di ambienti containerizzati, l'esigenza di garantire una robusta difesa contro minacce informatiche diviene ancora più importante.

Durante il mio percorso universitario ho avuto l'occasione di approfondire questa tematica. Grazie all'iniziativa del *Cybersecurity National Lab* ho avuto la possibilità di affrontare un percorso di addestramento denominato **CyberChallenge.IT**. Il percorso formativo mira a fornire le basi metodologiche e pratiche richieste per analizzare vulnerabilità e possibili attacchi, identificando le soluzioni più idonee a prevenirli, in ambiti diversi della cybersecurity.

In particolare, è organizzato nelle seguenti aree tematiche:

- Ethical Hacking
- Crittografia
- Sicurezza Web
- Sicurezza Software
- Sicurezza Hardware
- Sicurezza delle Reti
- Analisi Malware

[1] Hacking finalizzato a migliorare infrastrutture e non recare danno ad esse o ad i suoi utenti.

- Attacco/Difesa

Con le competenze acquisite e volendo continuare il percorso di tirocinio intrapreso, ho deciso di provare a sfruttare vulnerabilità note e di vitale importanza nel mondo del lavoro di tutti i giorni.

1.2 Obiettivo della tesi

L'obiettivo principale della tesi è esplorare e riuscire a sfruttare vulnerabilità recenti al fine di valutare la loro potenziale utilità in attacchi informatici. Attraverso l'analisi di scenari reali e l'applicazione di tecniche di penetration testing, si mira a comprendere la dinamica della vulnerabilità ed eventualmente suggerire una correzione.

1.3 Struttura della tesi

La tesi seguirà la seguente struttura logica ed organizzativa:

- Nel capitolo 2 verrà approfondita la vulnerabilità analizzata, motivandone la sua scelta e cercando di spiegare il contesto in cui si opera.
- Nel capitolo 3 verranno analizzate dettagliatamente tutte le risorse, gli strumenti e le metodologie utilizzate per replicare la vulnerabilità.
- Il capitolo 4 spiega, passaggio dopo passaggio, le istruzioni da compiere per replicare la vulnerabilità.
- Nel penultimo capitolo (5) verranno espresse considerazioni personali e dato un giudizio critico su quanto svolto durante la tesi.
- Infine, nell'ultimo capitolo (6), verranno esposte le conclusioni e i ringraziamenti.

2 Vulnerabilità

In questo capitolo verrà trattato ed approfondito lo scopo della tesi. Si introdurrà il **NIST** e la vulnerabilità selezionata, argomentando la motivazione della scelta.

2.1 NIST

Il NIST (National Institute of Standards and Technology) è un'agenzia governativa degli Stati Uniti d'America (logo in figura 1). Il NIST è responsabile di sviluppare e promuovere standard e linee guida per migliorare l'efficienza e la competitività delle organizzazioni, nonché di garantire la sicurezza e la privacy delle informazioni.

Figura 1: NIST (figura ripresa dal sito ufficiale [12])

Un settore di particolare rilevanza è quello della sicurezza informatica, dove il NIST è noto per lo sviluppo del Framework per la gestione della Cybersecurity (NIST Cybersecurity Framework).

Inoltre il NIST pubblica costantemente le vulnerabilità più rilevanti nel proprio Database **NVD** (National Vulnerability Database).

La definizione di vulnerabilità, come indicato nel sito del NIST, è la seguente:

> *"Weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat source."* [12]

ovvero una debolezza nelle procedure di sicurezza di un sistema informativo o nella sua implementazione, che potrebbe essere sfruttata da una fonte di minaccia.

Altre funzioni dell'NVD includono:

- **Raccolta di Informazioni sulle Vulnerabilità:** l'NVD raccoglie e cataloga informazioni dettagliate su vulnerabilità software da fonti pubbliche e private.

- **Assegnazione di Identificatori Unici:** assegna identificatori unici, come il numero di identificazione comune delle vulnerabilità (**CVE**), a ciascuna vulnerabilità. Questi identificatori sono utilizzati a livello globale per garantire un riferimento univoco.
- **Assegnazione di Metadati:** fornisce metadati dettagliati sulle vulnerabilità, comprese descrizioni, soluzioni e altre informazioni pertinenti.
- **Classificazione del Rischio:** classifica il rischio associato a ciascuna vulnerabilità per aiutare le organizzazioni a valutare l'importanza di applicare correzioni o misure di mitigazione.

2.2 Vulnerabilità ArgoCD

Tra le numerose vulnerabilità presenti, si è scelto di approfondire la **CVE-2023-40029**, la quale riguarda ArgoCD, un'estensione di Kubernetes dedicata all'implementazione della Continuous Deployment (CD). Secondo quanto riportato nella descrizione del NIST:

"Argo CD Cluster secrets might be managed declaratively using Argo CD and kubectl apply. As a result, the full secret body is stored in kubectl.kubernetes.io/last-applied-configuration annotation."

La valutazione di criticità assegnata è **HIGH**, con un punteggio di 9,6 su 10 (come visibile in figura 2). Data l'elevata importanza della vulnerabilità e la

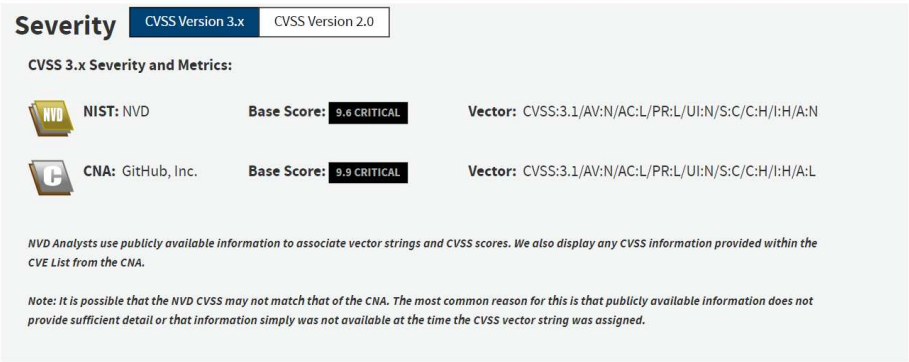

Figura 2: Valutazione NIST

rilevanza degli strumenti coinvolti, come Kubernetes, e considerando il recente rilascio della vulnerabilità (datato 7 settembre 2023), si è scelto di approfondire ulteriormente questo caso. La vulnerabilità rientra della categoria **CWE-532**: *Insertion of Sensitive Information into Log File*. Le informazioni scritte nei file di log possono essere di natura riservata e fornire indicazioni preziose ad un utente malintenzionato o esporre informazioni sensibili dell'utente. Le informazioni necessarie per replicare la vulnerabilità sono state riprese anche dal

repository Git [13].

Il seguente esempio aiuta a capire il concetto:

```
locationClient = new LocationClient(this, this, this);
locationClient.connect();
currentUser.setLocation(locationClient.getLastLocation());

...

catch (Exception e) {
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setMessage("Sorry, something went wrong");
AlertDialog alert = builder.create();
alert.show();
Log.e("Caught exception:"+ e + "While on User:"+User.toString());
}
```

L'esempio è stato ripreso direttamente da sito del CWE (common weakness enumeration), che cataloga tutti i tipi di vulnerabilità [3]. Quando l'applicazione rileva un'eccezione, memorizza l'oggetto utente nel file log. Poiché l'oggetto utente contiene dettagli sulla posizione, vengono inclusi anche questi. In questo modo, dati che potrebbero essere sensibili potrebbero venir esposti.

Volendo comparare questo tipo di vulnerabilità al tipo selezionato, è possibile evidenziare una similitudine tra i file di log e le *annotation* di Argo. Infatti, una volta effettuato il comando l'apply di un file di tipo *Cluster secret*, il suo contenuto verrà salvato nelle annotation stesse (salvando di conseguenza anche dati sensibili).

Nei prossimi capitoli verrà impiegato del tempo per spiegare molte delle risorse e strumenti sopra citati, come Kubernetes o il comando "*Kubectl apply*".

3 Metodologie e Strumenti

Attraverso questa sezione, si propone di esplorare le metodologie e gli strumenti chiave che hanno permesso di replicare la vulnerabilità. In particolar modo verranno approfonditi gli aspetti riguardanti **continuous integration** e **continuous delivery**, ovvero la pipeline per mettere in esecuzione un'applicazione. Per ottimizzare questo processo si utilizzano strumenti fondamentali come:

- **Docker**: ha un ruolo cruciale nella creazione di ambienti containerizzati.
- **Kubernetes**: offre un ambiente robusto e scalabile per la gestione di container.
- **Kind**: un potente strumento che semplifica la creazione di cluster Kubernetes all'interno di Docker, facilitando il processo di sviluppo e test.
- **ArgoCD**: offre una soluzione ottimizzata per la distribuzione delle applicazioni, migliorando notevolmente la sezione di continuous delivery.

insieme ad altri strumenti e metodologie che verranno introdotte successivamente nel capitolo.

3.1 CI/CD

Nel contesto dell'industria IT tradizionale, i dipendenti erano divisi in due team distinti: lo sviluppo, noto come il team **Dev**, e le operazioni IT, conosciute come il team **Ops**. Storicamente, questi due gruppi operavano in modo isolato, con scarsa comunicazione tra loro. Tuttavia, l'approccio organizzativo di **DevOps** ha rivoluzionato questa mentalità, promuovendo una stretta collaborazione tra Dev e Ops.

L'obiettivo principale di DevOps è realizzare la **continuous integration** (integrazione continua) da parte del team Dev e la **continuous delivery** (distribuzione continua) da parte del team Ops. Questo significa che non vi è più una netta distinzione tra lo sviluppo e l'implementazione di un software. Ogni piccola modifica al codice viene programmata, testata e distribuita senza interruzioni nell'esercizio del sistema. In altre parole, con DevOps c'è un flusso continuo di sviluppo, test e rilascio del software, senza la necessità di interruzioni per aggiornare il sistema (figura 3). Questo approccio consente un aggiornamento costante e graduale del software, garantendo una maggiore efficienza e agilità nello sviluppo e nell'implementazione delle applicazioni.

3.1.1 Continuous Integration

Durante lo sviluppo di applicazioni, uno degli obiettivi principali è permettere a diversi sviluppatori di collaborare contemporaneamente su varie funzionalità dell'applicazione. Tuttavia, quando diverse diramazioni del codice sorgente vengono unificate, le attività risultanti possono diventare noiose, manuali e richiedere molto tempo. Questo scenario si verifica quando gli sviluppatori apportano

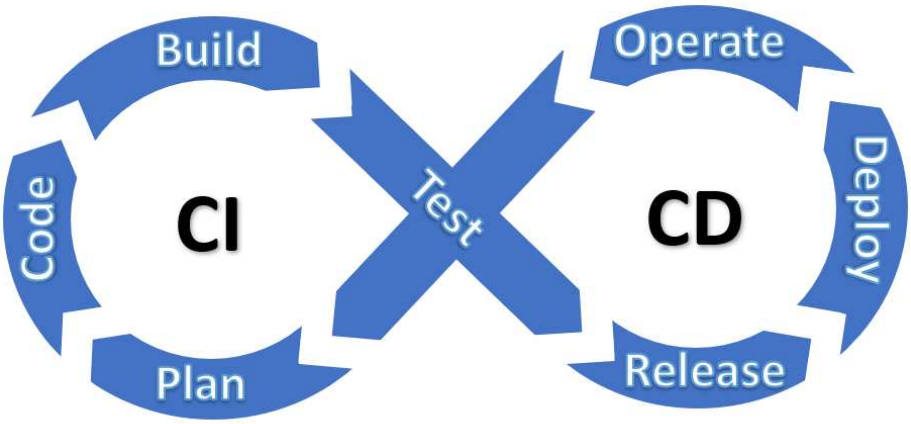

Figura 3: CI-CD

modifiche indipendenti all'applicazione che possono essere divergenti da quelle effettuate contemporaneamente da altri membri del team.

L'integrazione continua consente agli sviluppatori di incorporare le modifiche nel codice con maggiore frequenza all'interno di un'unica diramazione condivisa. Dopo l'unificazione, le modifiche vengono validate attraverso la compilazione automatica dell'applicazione e l'esecuzione di vari test automatici, come i test di unità e integrazione, per assicurare che le modifiche non abbiano causato problemi. Nel caso in cui sorgano conflitti tra il nuovo codice e quello esistente, l'integrazione continua agevola la risoluzione di tali conflitti.

3.1.2 Continuous Delivery

Dopo aver automatizzato le fasi di build e testing come parte dell'integrazione continua (CI), la distribuzione continua (CD) entra in gioco per gestire il rilascio automatizzato del codice convalidato in un repository. Per questo, è fondamentale che l'integrazione continua sia già parte integrante della pipeline di sviluppo per garantire l'efficacia del processo di distribuzione continua. L'obiettivo principale della distribuzione continua è mantenere il codice costantemente pronto per essere distribuito in un ambiente di produzione.

Ogni fase, che va dalla fusione delle modifiche al codice al rilascio di una build pronta per la produzione, coinvolge l'automazione dei test e del rilascio del codice. Alla fine di questo processo, il team operativo è in grado di eseguire il deployment dell'applicazione in produzione in modo rapido e senza complicazioni, grazie alla certezza che il codice sia stato testato e convalidato in ogni fase del processo di sviluppo.

3.2 Docker

Docker (logo in figura 4) è una piattaforma open-source che facilita la creazione, la distribuzione e l'esecuzione di applicazioni all'interno di container. I container sono un'unità di software leggera e portatile che includono tutto il necessario per eseguire un'applicazione, come il codice, le librerie e le dipendenze. Seppur di vi-

Figura 4: Docker (immagine ripresa da [17])

tale importanza, esula dalle specifiche della tesi entrare nel dettaglio di Docker, ma verranno trattati solamente alcuni componenti principali e come funzionano.

L'evoluzione delle modalità di distribuzione delle applicazioni (figura 5) ha segnato un importante passaggio dalle infrastrutture basate su server fisici a soluzioni virtualizzate. La virtualizzazione ha introdotto la possibilità di eseguire diverse macchine virtuali su una singola CPU fisica, portando notevoli miglioramenti in termini di efficienza delle risorse e flessibilità. Tuttavia, i container, caratterizzati da un isolamento più leggero, hanno ulteriormente rivoluzionato il panorama della distribuzione delle applicazioni. Grazie alla condivisione del sistema operativo tra le applicazioni, i container offrono una maggiore portabilità, scalabilità e vantaggi gestionali rispetto alla virtualizzazione tradizionale. Questo approccio supera le limitazioni della virtualizzazione classica, consentendo agli sviluppatori di distribuire e gestire le applicazioni in modo più efficiente e dinamico.

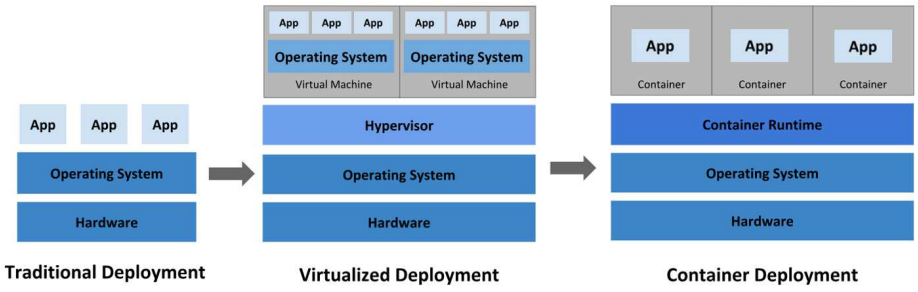

Figura 5: Evoluzione del Deployment di applicazioni.

- **Docker Engine:** è il motore principale di Docker ed è responsabile della

gestione dei container. Include il daemon di sistema[2] (Dockerd), un API che comunica con il Docker CLI e un registro per la memorizzazione delle Docker Images.

- **Docker CLI:** è l'interfaccia a riga di comando che consente agli utenti di interagire con Docker. Gli utenti possono eseguire comandi come la creazione di nuovi contenitori, la gestione delle immagini e il monitoraggio dello stato del sistema.
- **Docker Images:** le immagini Docker sono un modello di base per la creazione di contenitori. Un'immagine è uno snapshot leggero e autonomo che include il codice dell'applicazione, le librerie e tutte le dipendenze necessarie per eseguire l'applicazione.
- **Docker Containers:** i container Docker sono istanze in esecuzione di un'immagine. Mentre un'immagine è uno stato immutabile, un contenitore rappresenta uno stato in esecuzione di quella immagine, con il proprio filesystem scrivibile e la propria area di memoria.
- **Dockerfile:** è un file di testo che contiene le istruzioni per la costruzione di un'immagine Docker. Le istruzioni possono includere la base dell'immagine, le dipendenze, le variabili d'ambiente e i comandi per l'esecuzione di specifiche azioni durante la costruzione.
- **Docker Compose:** è uno strumento per la definizione e l'esecuzione di applicazioni Docker multicontainer. Con Docker Compose, è possibile definire un'applicazione composta da più servizi, ognuno dei quali è definito in un file docker-compose.yml.

Il flusso di lavoro di Docker è suddiviso principalmente in:

1. **Creazione di un'immagine:** gli sviluppatori definiscono un'immagine Docker usando un file Dockerfile. Questo file contiene tutte le istruzioni necessarie per creare un'immagine, inclusi i comandi per installare dipendenze, configurare l'ambiente e copiare il codice dell'applicazione.
2. **Costruzione dell'immagine:** utilizzando il comando *docker build*, l'immagine viene costruita secondo le istruzioni del Dockerfile. L'immagine risultante viene archiviata localmente o può essere inviata a un registro Docker per la condivisione.
3. **Esecuzione di un contenitore:** con il comando *docker run*, un'istanza di un'immagine viene avviata come un contenitore. Questo contenitore è completamente isolato dagli altri contenitori e dal sistema host, ma può comunque comunicare attraverso le interfacce di rete definite.

[2] È un tipo di processo informatico che opera in background su un sistema operativo, eseguono diverse attività senza l'interazione diretta dell'utente.

4. **Gestione dei contenitori:** gli sviluppatori possono interagire con i contenitori usando il Docker CLI. Possono visualizzare lo stato dei contenitori, arrestarli, eliminarli o eseguire comandi specifici all'interno di un contenitore in esecuzione.

Le immagini possono essere caricate su un registro Docker, che funge da deposito centrale. Da lì, le immagini possono essere scaricate e eseguite su qualsiasi host Docker. Questa facilità di distribuzione consente di scalare facilmente applicazioni distribuite su più ambienti.

La precedente sezione è stata scritta utilizzando l'ausilio della documentazione ufficiale di Docker [4].

3.3 Kubernetes

Con i container, sorge l'immediata necessità di un meccanismo per la gestione di carichi di lavoro e servizi containerizzati: **Kubernetes** (denominato brevemente K8s). Nato dal progetto interno di Google Borg (poi regalato alla comunità in modalità Open Source), Kubernetes è diventato lo standard di riferimento per quanto riguarda l'orchestrazione di container.

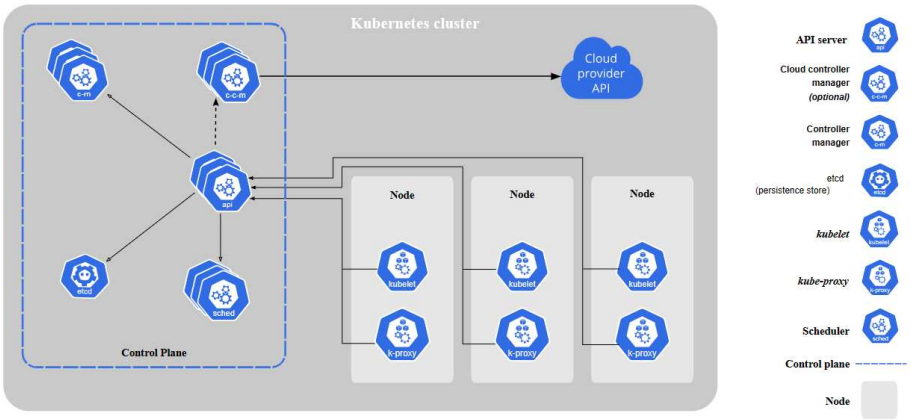

Figura 6: Cluster (Immagine ripresa dal sito ufficiale [9])

3.3.1 Architettura di un Cluster

Un cluster è un insieme di nodi (o macchine) che lavorano insieme per eseguire applicazioni e carichi di lavoro. Un cluster Kubernetes, rappresentato in Figura 6, è composto da due tipi principali di nodi: i nodi Master e i nodi Worker.

- **Master node**: contengono funzionalità essenziali per il funzionamento del cluster stesso.

- **Worker node**: mettono a disposizione le risorse necessarie per le applicazioni che gireranno all'interno del cluster.

L'insieme di funzionalità offerte dai Master node è detta **Control Plane**. Di seguito verranno illustrati in dettaglio i componenti di un cluster Kubernetes. La seguente sezione è stata scritta con l'ausilio del video guida riportato in bibliografia [11].

Control Plane

- **Controller Manager:** La sua funzione principale è garantire che lo stato attuale del cluster corrisponda allo stato desiderato definito tramite la configurazione. In particolare, il Controller Manager si occupa di monitorare continuamente il sistema, rilevando eventuali deviazioni dallo stato desiderato e prendendo le azioni necessarie per ristabilire l'equilibrio. Il controller manager contiene diversi controller, ognuno dei quali è specializzato in un compito specifico. Ad esempio il *replica controller* per garantire che il numero desiderato di repliche di un'applicazione sia in esecuzione, il *node controller* per gestire i nodi del cluster e il *service controller* per garantire la corrispondenza tra i servizi e i pod corrispondenti. La lista sopra non è esaustiva.

- **API server:** L'API Server di Kubernetes fornisce un'interfaccia centralizzata come punto di accesso per interagire con il sistema. Questa interfaccia può assumere diverse forme, tra cui un'interfaccia utente (UI) per l'interazione grafica, API per l'automazione tramite script, o una riga di comando come **Kubectl** per operazioni dirette e immediate.

- **Etcd:** Database persistente chiave-valore utilizzato come archivio di backup per tutti i dati del cluster Kubernetes.

- **Scheduler:** Lo scheduler ha il compito di avviare l'esecuzione dei Pod, i quali contengono i container che verranno eseguiti. La sua funzione principale è garantire che questa esecuzione avvenga nei nodi worker, rispettando rigorosamente i vincoli hardware e computazionali definiti dagli amministratori del cluster.

- **Cloud Control Manager:** Gestisce e dialoga con la piattaforma cloud.

Worker node

- **Kubelet:** è un demone operante su ogni singolo nodo, responsabile della supervisione del suo stato di salute e assicurando che sia in comunicazione con l'API server.

- **Kube-proxy:** Kube-proxy è un proxy di rete che viene eseguito su ciascun nodo worker nel cluster, implementando parte del Service Kubernetes (che spiegheremo dettagliatamente in seguito).

Il Kube-proxy mantiene le regole di rete sui nodi. Queste regole consentono la comunicazione di rete con i pod dalle sessioni di rete all'interno o all'esterno del cluster.

I nodi Worker, sebbene meno critici, ospitano numerosi container e svolgono un ruolo fondamentale nell'esecuzione delle applicazioni. Essi gestiscono il carico computazionale effettivo ed eseguono i container necessari. Dato il loro ruolo più operativo, possono risultare più pesanti rispetto ai nodi Master che, al contrario, rivestono un ruolo di vitale importanza nel coordinare e controllare l'intero cluster Kubernetes. La prassi consigliata include la creazione di copie di backup dei nodi Master. La perdita di uno di questi potrebbe compromettere l'intera struttura del cluster, impedendo la sua ricostruzione. Al contrario, la perdita di un nodo Worker è più gestibile poiché il nodo Master, attraverso il Controller Manager, può avviare la ricostruzione in modo automatico.

Esempio

Di seguito un esempio pratico per capire meglio il funzionamento:

1. Lo sviluppatore (Dev) prepara un file manifest in formato YAML che descrive lo stato desiderato dell'applicazione, specificando quanti pod (istanze) sono richiesti, le risorse necessarie e altre configurazioni. Il file manifest segue la sintassi della Desire State Configuration (DSC), indicando come l'applicazione dovrebbe essere piuttosto del come fare per ottenerla.

2. Utilizzando la riga di comando (ad esempio, con kubectl), lo sviluppatore interagisce direttamente con l'API Server di Kubernetes. Con il comando

   ```
   $ kubectl apply
   ```

   del manifest, specifica al server le modifiche desiderate nello stato del cluster. I comandi di Kubectl verranno trattati ampiamente in seguito.

3. L'API Server riceve la richiesta, verifica la validità del comando e la persiste nell'etcd, il database distribuito che conserva lo stato del cluster. Successivamente, l'API Server mette in coda la richiesta allo Scheduler di Kubernetes.

4. Lo Scheduler è responsabile di selezionare i nodi appropriati all'interno del cluster dove eseguire i pod dell'applicazione: esamina i vincoli hardware, le risorse disponibili e le policy di scheduling definite dagli amministratori per prendere decisioni.

5. Il Controller Manager monitora continuamente lo stato del cluster. In questo contesto, i controller specifici (come il ReplicaSet Controller) verificano che il numero di pod desiderato sia mantenuto e rispetti le specifiche di configurazione.

Le applicazioni devono essere esposte all'esterno da un *Load Balancer* (ovvero un dispositivo hardware o software progettato per avere una distribuzione equa del traffico e ottimizzare le prestazioni complessive del sistema). Il Load Balancer raggruppa e comunica con i pod, permettendo inoltre di creare un endpoint (un punto di accesso su un sistema o su una rete) verso l'esterno con i vari dispositivi.

Nelle successive sotto sezioni, verranno spiegati dettagliatamente i vari componenti di Kubernetes. Durante la scrittura di queste parti si è fatto riferimento unicamente alla documentazione ufficiale [9], al libro **Kubernetes Up & Running** [2] e al video in bibliografia [15]. Le immagini sono state realizzate utilizzando le icone disponibili su repository Kubernetes[10].

3.3.2 Pod

Un Pod rappresenta un insieme di container e volumi di applicazioni che operano all'interno dello stesso contesto di esecuzione (figura 7). Nei cluster Kubernetes, i Pod costituiscono le unità più piccole distribuibili. Questo implica che tutti i container all'interno di un Pod siano sempre eseguiti sulla stessa macchina. Ciascun container all'interno di un Pod funziona nel proprio *control group* [3],

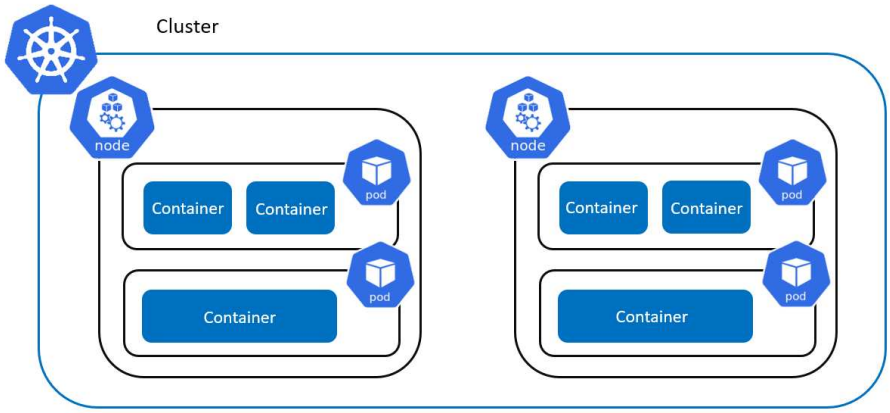

Figura 7: Pods

ma condivide lo stesso indirizzo IP e spazio delle porte, condividono lo stesso hostname e possono comunicare attraverso canali di comunicazione interprocesso nativi.

Tuttavia, le applicazioni in Pod diversi sono isolate l'una dall'altra, con indirizzi IP, hostname e altri attributi distinti. I container di Pod diversi che operano sulla stessa macchina sono trattati essenzialmente come se fossero su server separati. In generale, la domanda corretta da porsi quando si progettano i Pod è: "Questi container funzionerebbero correttamente se si trovassero su macchine diverse?" Se la risposta è "no", allora un Pod è il raggruppamento corretto per i container. Se la risposta è "sì", probabilmente la soluzione corretta sarebbe utilizzare Pod diversi.

Creazione di un Pod

Un pod è interamente descritto in un file Manifest usando il formato YAML o JSON (il primo è preferito in quanto considerato più leggibile). Il server API di Kubernetes accetta ed elabora i file manifest dei Pod prima di archiviarli nell'archiviazione persistente (etcd). Lo scheduler utilizza l'API di Kubernetes per individuare i Pod che non siano stati pianificati su un nodo. Successivamente,

[3] Un meccanismo essenziale in Linux consente di limitare, misurare e isolare le risorse di sistema allocate a processi o gruppi di processi.

colloca i Pod nei nodi in base alle risorse e ad altri vincoli espressi nei manifest. Lo scheduler può collocare più Pod sulla stessa macchina purché ci siano risorse sufficienti. Tuttavia, pianificare più repliche della stessa applicazione sulla stessa macchina risulta inaffidabile, poiché la macchina rappresenta un punto di guasto . Di conseguenza, lo scheduler di Kubernetes cerca di garantire che i Pod della stessa applicazione siano distribuiti su macchine diverse per garantire l'affidabilità in presenza di tali guasti. Una volta pianificati su un nodo, i Pod non si spostano e devono essere esplicitamente distrutti e rieseguiti.

Di seguito è riportato un esempio di pod costituito da un contenitore che esegue l'immagine nginx:1.14.2:

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.14.2
    ports:
    - containerPort: 80
```

Per creare il pod mostrato sopra, si può eseguire il comando:

```
$ kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
```

Health Checks Quando viene eseguita un'applicazione come un container in Kubernetes, questa viene automaticamente mantenuta attiva attraverso un *process health check*. Questo controllo garantisce semplicemente che il processo principale dell'applicazione sia sempre in esecuzione. Se ciò non accade, Kubernetes lo riavvia.

Tuttavia, nella maggior parte dei casi, una semplice verifica potrebbe non essere sufficiente. Ad esempio, se il processo è in *deadlock*[4]e non è in grado di gestire le richieste, un process health check continuerà a ritenere che l'applicazione sia in uno stato sano, poiché il suo processo è ancora in esecuzione. Per affrontare questa situazione, Kubernetes ha introdotto i *liveness health check*. Questi controlli verificano la logica dell'applicazione, come il caricamento di una pagina web, per constatare che l'applicazione non solo sia in esecuzione, ma funzioni correttamente. Poiché questi health check sono specifici dell'applicazione, è necessario definirli nel file manifest del Pod.

[4]Due o più processi o thread nel sistema rimangono bloccati indefinitamente perché ciascuno sta aspettando che l'altro rilasci una risorsa.

Considerazioni La creazione diretta di singoli pod in Kubernetes è un'operazione che si verifica raramente, anche nel caso di pod singleton. Questa limitata frequenza deriva dal fatto che i pod sono concepiti come entità temporanee "usa e getta". Quando un nuovo pod viene creato, che sia iniziativa diretta dell'utente o indirettamente gestita da un controller, il sistema di orchestrazione decide su quale nodo del cluster allocarlo. Il pod persiste su quel nodo fino a quando non si verifica una delle seguenti condizioni: la sua esecuzione termina, l'oggetto pod viene eliminato volontariamente, il pod viene rimosso a causa di una carenza di risorse, o il nodo ospitante cessa di funzionare.

È importante sottolineare che il riavvio di un container all'interno di un pod non deve essere confuso con il riavvio dell'intero pod. Un pod non è un processo autonomo, ma piuttosto un ambiente che ospita l'esecuzione di container. Pertanto, il pod mantiene la sua esistenza finché non viene esplicitamente eliminato, evidenziando il concetto che la sua vita persiste oltre i singoli riavvii dei container al suo interno.

Un occhio più esperto può aver osservato un'incongruenza nel processo: poiché ogni pod possiede un indirizzo IP e altre caratteristiche univoche che, in caso di distruzione, devono essere riassegnate, questo potrebbe comportare un problema significativo, in quanto richiederebbe la riassegnazione o la modifica di ogni altro pod per garantire la continuità della comunicazione. È per questo che vengono impiegati i **Service**.

3.3.3 Service

Deployment e ReplicaSet (verranno analizzati in seguito) possono dinamicamente creare e distruggere pod. In qualsiasi momento, potrebbe non essere possibile sapere quanti di questi pod siano in esecuzione o sani, e risulterebbe difficile conoscere i loro nomi specifici. I pod Kubernetes vengono creati e distrutti per riflettere lo stato desiderato del cluster. Va ricordato che i pod sono considerati risorse effimere, il che significa che non è garantito che un singolo pod sia affidabile o persistente. Ogni pod ottiene il proprio indirizzo IP. Per una specifica distribuzione nel cluster, l'insieme di pod in esecuzione in un dato momento potrebbe differire da quello che eseguirà l'applicazione un istante dopo e di conseguenza anche l'indirizzo IP potrebbe cambiare (senza Service). Questa situazione presenta una sfida: se alcuni set di pod (chiamati "backend") forniscono funzionalità ad altri pod (chiamati "frontend") all'interno del cluster, come i frontend possono sapere e tracciare a quale indirizzo IP connettersi? I Service sono stati pensati esattamente per questo caso d'uso.

In Kubernetes, un Service costituisce un metodo per esporre un'applicazione di rete in esecuzione come uno o più pod all'interno del cluster. L'API del Service, parte integrante di Kubernetes, fornisce un'astrazione che facilita l'esposizione di gruppi di pod su una rete. Ogni oggetto Service definisce un insieme logico di endpoint insieme a una politica su come rendere accessibili tali pod. Quando

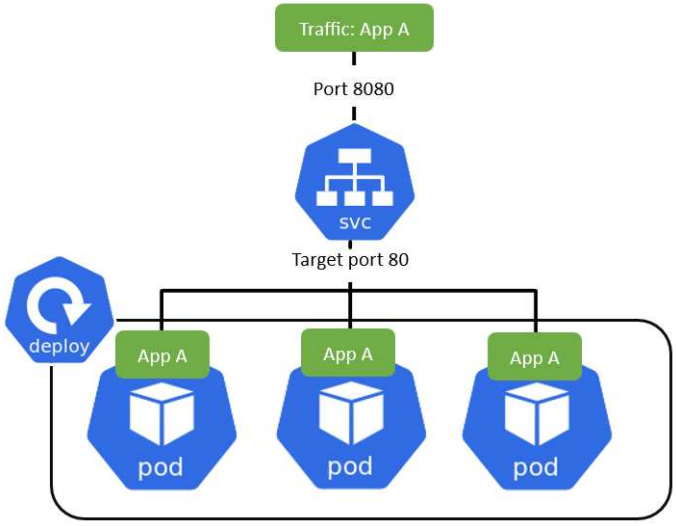

Figura 8: Service

i pod vengono raggruppati in un Service, ottengono un indirizzo IP virtuale e una porta, e il Service funge da bilanciatore di carico per la comunicazione con questi pod (nella figura 8 e nel file .yaml illustrato in seguito si può vedere come la porta 8080 sia mappata sulla 80 dei pods. L'indirizzo IP virtuale è interamente gestito in maniera trasparente rispetto a chi deve comunicare).

Per esempio, si consideri un backend di elaborazione senza stato che esegue 3 repliche. Queste repliche sono fungibili, il che significa che i frontend non devono preoccuparsi di quale backend stiano utilizzando. Sebbene i pod effettivi all'interno del set di backend possano cambiare, i client frontend non dovrebbero esserne consapevoli, né dovrebbero tenere traccia del set di backend stesso. L'astrazione del Service consente questo tipo di disaccoppiamento.

Di seguito un piccolo esempio del manifest di un Service minimale:

```
apiVersion: v1
kind: Service
metadata:
  name: my-Service
spec:
  selector:
    app.kubernetes.io/name: MyApp
  ports:
    - protocol: TCP
      port: 8080
```

```
    targetPort: 80
```

L'apply del file manifest crea un nuovo Service denominato "my-service". Il Service ha come target la porta TCP 80 su qualsiasi pod con l' *app.kubernetes.io/name*: MyApp.

In parole povere, il Service è come un indirizzo IP statico persistente con un DNS (Domain Name System), in modo da non dover regolarmente modificare l'endpoint quando un pod muore. Inoltre, è anche un *Load Balancer* in quanto intercetta le richieste e le inoltra al pod meno occupato.

3.3.4 Volume

Si supponga che un pod sia in esecuzione e generi dati rilevanti. È stato ampiamente discusso che i pod sono entità effimere, usa e getta, che potrebbero essere riavviate in caso di errori. Se per qualche motivo il pod dovesse essere riavviato dove finirebbero i dati raccolti fino a quel momento? Si presenta dunque un problema: se il container del database o il pod dovesse venir riavviato, i dati andranno persi, causando disagi e inconvenienti. Ovviamente, si desidera che i dati del database o i dati di registro siano conservati in modo affidabile nel lungo termine. In Kubernetes, è possibile affrontare questa sfida utilizzando un altro componente chiamato **Volume** (illustrato in figura 9). Il funzionamento

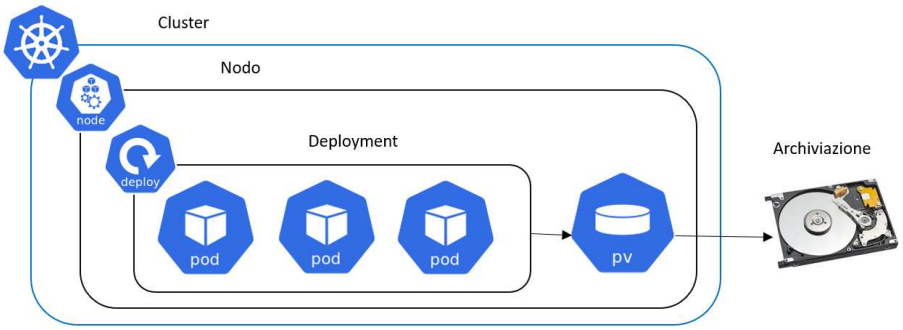

Figura 9: Volume (pv)

di Volume consiste nel collegare uno spazio di archiviazione al pod. Questo spazio di archiviazione può trovarsi sia sulla stessa macchina locale (cioè sullo stesso nodo del server) in cui è in esecuzione il pod, sia su uno spazio di archiviazione remoto al di fuori del cluster Kubernetes. Questo spazio potrebbe essere un'archiviazione cloud o un'archiviazione locale, che non fa parte del cluster Kubernetes, ma viene semplicemente referenziata esternamente. Quando il container o il pod del database viene riavviato, tutti i dati saranno conservati e persistenti. Si può dunque pensare ad uno spazio di archiviazione come a un disco rigido esterno collegato al cluster Kubernetes.

Il punto chiave è che il cluster Kubernetes non gestisce esplicitamente la persistenza dei dati. Di conseguenza, come utente o amministratore di Kubernetes, si è responsabili del backup, della replica, della gestione e dell'assicurazione che i dati siano conservati su hardware adeguato, indipendentemente dal fatto che si tratti di archiviazione locale o remota.

3.3.5 Ingress

Ingress (illustrato in figura 10) è un oggetto API che gestisce le richieste di routing del traffico HTTP e HTTPS verso i servizi all'interno del cluster. L'Ingress fornisce un modo per esporre servizi HTTP e HTTPS tramite un singolo indirizzo IP pubblico, consentendo di gestire il routing del traffico in base al percorso dell'URL o ad altri criteri. Le principali funzionalità offerte da un

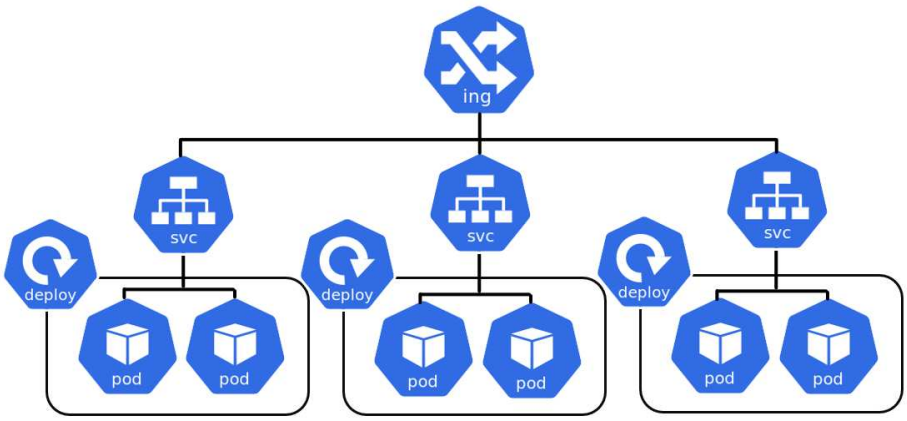

Figura 10: Ingress

Ingress includono:

- **Routing del Traffico:** l'Ingress permette di definire regole per instradare le richieste HTTP o HTTPS a servizi specifici in base a percorsi specifici dell'URL o altri criteri come l'host.
- **TLS/SSL Termination:** supporta la gestione del traffico sicuro attraverso TLS/SSL termination, consentendo di gestire in modo centralizzato la crittografia e la decrittografia del traffico HTTPS.

  > *Transport Layer Security (TLS) e il suo predecessore Secure Sockets Layer (SSL) sono dei protocolli crittografici usati nel campo delle telecomunicazioni e dell'informatica che permettono una comunicazione sicura dalla sorgente al destinatario (end-to-end) su reti TCP/IP.* [16]

- **Load Balancing:** può essere integrato con servizi di bilanciamento del carico per distribuire il traffico tra più repliche dei servizi backend.

- **Rewrite di URL:** permette di riscrivere gli URL delle richieste, consentendo la manipolazione dinamica del percorso dell'URL.

Affinché la risorsa Ingress funzioni, il cluster deve avere un ingress controller in esecuzione. A differenza di altri tipi di controller eseguiti come parte del *controller-manager*, i controller Ingress non vengono avviati automaticamente con un cluster.

Di seguito possiamo vedere come creare una risorsa Ingress minimale:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: minimal-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx-example
  rules:
  - http:
      paths:
      - path: /testpath
        pathType: Prefix
        backend:
          service:
            name: test
            port:
              number: 80
```

3.3.6 Config Map

Nelle sezioni precedenti è stato descritto come i pod comunichino tra di loro attraverso il Service. Se, ad esempio, l'endpoint del Service dovesse cambiare? Si dovrebbe ricostruire l'applicazione e ricaricarla nel repository, effettuare la push dell'immagine nel pod e riavviare il tutto. Questa procedura sarebbe impensabile se fosse adottata per ogni singolo cambiamento. La soluzione adottata per evitare ciò è la Config Map.

Una Config Map (figura 11) consente di disaccoppiare la configurazione specifica dell'ambiente dalle immagini del container, in modo che le applicazioni siano facilmente trasportabili. La configurazione dell'ambiente contiene informazioni di configurazione come l'URL di un database o di altri servizi utilizzati. Collegando la Config Map ad un pod si consente allo stesso di accedere ai dati contenuti nella mappa. Quindi, se si modifica il nome o l'endpoint del servizio, è sufficiente regolare la Config Map senza la necessità di creare una nuova immagine o di passare attraverso l'intero processo di distribuzione. Questo approccio

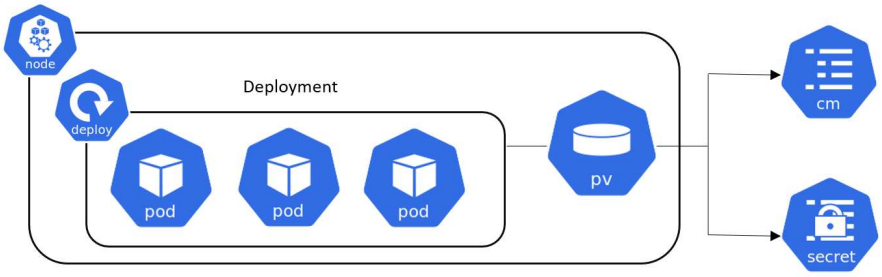

Figura 11: Config Map e Secret

semplifica la gestione delle configurazioni esterne, offrendo maggiore flessibilità e rapidità nel gestire modifiche come cambiamenti di URL dei servizi.

3.3.7 Secret

Parte della configurazione esterna potrebbe coinvolgere elementi come l'username o la password, specifici per la configurazione dell'applicazione, e tali elementi possono subire variazioni durante il processo di distribuzione. Tuttavia, inserire una password o altre credenziali in una Config Map in formato di testo normale rappresenterebbe una vulnerabilità, anche se si tratta di configurazione esterna. Per affrontare questa sfida, Kubernetes fornisce un altro componente noto come **Secret**.(figura 11)

Il Secret è simile a una Config Map, ma si differenzia per il suo utilizzo specifico. Viene impiegato per archiviare dati sensibili i quali vengono memorizzati in formato codificato in base64, anziché in formato di testo normale. È fondamentale sottolineare che la semplice codifica in base64 non garantisce automaticamente la sicurezza di un Secret. I componenti di Secret sono progettati per essere crittografati tramite strumenti di terze parti in Kubernetes, dal momento che Kubernetes non offre di default funzionalità di crittografia. Esistono strumenti forniti dai provider cloud o soluzioni di terze parti, che possono essere implementate su Kubernetes per crittografare i Secret, rendendoli sicuri.

All'interno di un Secret, vengono inclusi dati sensibili, quali credenziali, soprattutto password e certificati, elementi ai quali non si desidera che altre persone abbiano accesso diretto. Proprio come per una Config Map, è sufficiente collegare il Secret al pod in modo che possa accedere a tali dati e leggerli dal Secret. Questo approccio consente una gestione sicura e flessibile dei dati sensibili nell'ambiente Kubernetes.

3.3.8 Deployment

Si consideri il caso in cui un pod dell'applicazione venisse arrestato, andasse in crash o dovesse essere riavviato perché è stata creata una nuova immagine del container. Fondamentalmente, si verificherebbe un periodo di inattività in cui

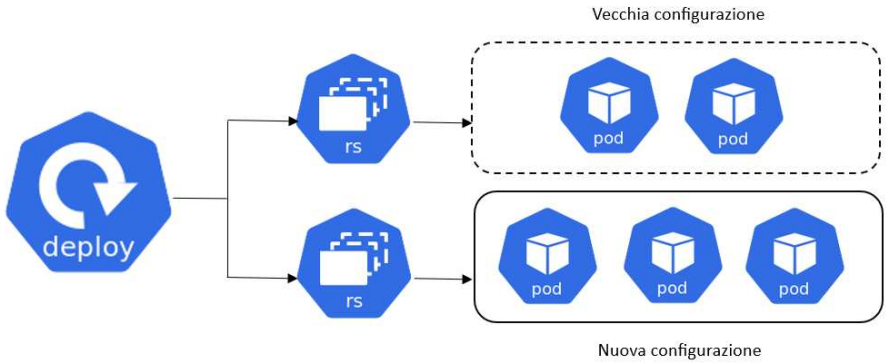

Figura 12: Deployment

un utente non potrebbe raggiungere l'applicazione, il che rappresenta un problema che si dovrebbe impedire a tutti i costi. Ed è proprio qui che risiede il vantaggio dei sistemi distribuiti e dei container. Invece di fare affidamento su un singolo pod e un solo database, si replica tutto su più pod. Per creare la seconda replica del pod non ne serve un nuovo, bensì occorre definire un *blueprint*[5] per un pod dove si deve specificare quante repliche di quel pod si vorrebbero eseguire. Questo componente (o questo blueprint) è chiamato **Deployment**. Nella pratica, si lavorerebbe principalmente con i deployment anziché con i pod, poiché consentono di specificare il numero desiderato di repliche e di regolare dinamicamente tale numero. Se, ad esempio, un pod dell'applicazione dovesse fallire, il servizio indirizzerà automaticamente le richieste ad un'altra replica, garantendo l'accessibilità dell'applicazione per gli utenti. Nella figura 12 si può notare come è il deployment stesso che gestisce, attraverso i **Replica Sets**, la configurazione dei pod. In questo caso è come se la configurazione fosse passata da due a tre repliche. I deployment non eliminano le configurazioni passate (a meno di effettuare i cambiamenti con la modalità *Pruning*) in modo che si possa fare il rollback dei cambiamenti.

E se il pod non fosse stateless (ovvero produce o immagazzina dati)? Nel caso in cui il pod del database dovesse arrestarsi, l'applicazione risulterebbe inaccessibile. Pertanto, è necessario avere una replica del database. Tuttavia, la replicazione del database tramite un Deployment non è possibile poiché il database contiene uno stato, dati. Ciò implica che, se si avessero cloni o repliche

[5] Un modello, un piano.

del database, tutti dovrebbero accedere alla stessa area di archiviazione dati condivisa. Qui entra in gioco uno strumento specifico: lo **Statefulset**.

Con n repliche bilanciate del pod dell'applicazione ed n repliche sincronizzate del database, la configurazione risulta più robusta. Questo significa che, anche in caso di riavvio o arresto di un nodo del server, c'è un secondo nodo con repliche dell'applicazione e del database in esecuzione. L'applicazione rimane accessibile agli utenti finché almeno una delle n repliche resta attiva, evitando così tempi di inattività.

Nelle sezioni successive parleremo dei **ReplicaSet**. Questi oggetti altro non sono che il motore dietro alle quinte alla base del Deployment.

3.3.9 StatefulSet

Questo componente è progettato appositamente per applicazioni con stato come ad esempio MySQL. Gli Statefulset gestiscono la replicazione dei pod e la loro scalabilità, garantendo al contempo che le operazioni di lettura e scrittura nel database siano sincronizzate per evitare inconsistenze.

Va sottolineato che la distribuzione di applicazioni database utilizzando Statefulset in un cluster Kubernetes può risultare complessa, più impegnativa rispetto all'utilizzo di Deployments dove le sfide sono minori. Di conseguenza, è comune ospitare le applicazioni database al di fuori del cluster Kubernetes e utilizzare Deployments o applicazioni senza stato che si replicano e scalano senza problemi all'interno del cluster, comunicando con il database esterno.

3.3.10 ReplicaSet e DaemonSet

ReplicaSet Un ReplicaSet ha lo scopo di garantire la presenza costante di un numero specificato di repliche identiche di pod in esecuzione. Per raggiungere questo obiettivo, ReplicaSet utilizza un selettore per identificare i pod, specifica il numero di repliche desiderato e un modello per la crearne di nuovi. Il ReplicaSet mantiene la coerenza tra il numero desiderato di repliche e l'effettivo numero, creando o eliminando i pod secondo necessità.

DaemonSet In genere, potrebbe essere utile replicare un pod su ogni nodo e distribuire un qualche tipo di agente o demone su ciascuno di essi. L'oggetto Kubernetes per ottenere ciò è il DaemonSet.

Un DaemonSet assicura che una copia di un Pod sia in esecuzione su un insieme di nodi in un cluster Kubernetes. I DaemonSets vengono utilizzati per distribuire, ad esempio, raccoglitori di log e agenti di monitoraggio, che devono essere in esecuzione su ogni nodo.

Analogie e differenze I DaemonSets condividono funzionalità simili con i ReplicaSets; entrambi creano pod progettati per essere servizi a lunga durata e assicurano che lo stato desiderato e lo stato osservato del cluster siano allineati.

Date le somiglianze tra DaemonSets e ReplicaSets, è importante capire quando utilizzare uno rispetto all'altro. I Deployment e ReplicaSets sono generalmente utilizzati per creare un servizio (come un server web) con diverse repliche per garantire la ridondanza. I ReplicaSets dovrebbero essere utilizzati quando l'applicazione è completamente disaccoppiata dal nodo e si possono eseguire più copie su un dato nodo senza particolari considerazioni. I DaemonSets dovrebbero essere utilizzati quando una singola copia dell' applicazione deve essere in esecuzione su tutti o su un sottoinsieme dei nodi nel cluster.

In generale, non si dovrebbero utilizzare restrizioni di pianificazione o altri parametri per garantire che i pod non siano collocati sullo stesso nodo. Se si ha il bisogno di un singolo pod per nodo, allora un DaemonSet è la risorsa corretta da utilizzare. Allo stesso modo, se si sta costruendo un servizio replicato omogeneo per gestire il traffico dell'utente, allora un ReplicaSet è probabilmente la risorsa Kubernetes giusta da utilizzare.

3.4 Kubectl

Kubernetes fornisce uno strumento a riga di comando per comunicare con il Control Plane di un cluster, utilizzando l'API di Kubernetes: **Kubectl**.

Per installare Kubectl basterà digitare il comando:

```
$ curl -LO "https://dl.k8s.io/release
  /$(curl -L -s https://dl.k8s.io/release/stable.txt)
  /bin/linux/amd64/kubectl"
```

Di seguito verranno esposti alcuni comandi utili.

3.4.1 Creazione risorse

I file manifest Kubernetes possono essere definiti in formato YAML o JSON. È possibile utilizzare l'estensione del file .yaml, .yml, e .json.

```
$ kubectl apply -f ./my-manifest.yaml
```

Il seguente comando crea una risorsa in base alla struttura del file manifest (come pod, servizi, deployment, ecc.)

La caratteristica più interessante è che lo stesso comando può essere utilizzato anche per aggiornare la stessa risorsa, infatti:

- se la risorsa specificata nel file non esiste ancora nel cluster, verrà creata.

- se la risorsa già esiste, il comando applica le modifiche specificate nel file al cluster.

```
$ kubectl apply -f
  https://git.io/SbattellaMattia/Tesi/my-manifest.yaml
```

Il seguente comando crea una risorsa da un url.

3.4.2 Distruzione risorse

```
$ kubectl delete -f ./my-pod.json
```

Elimina il pod relativo alla risorsa specificata (in questo esempio my-pod.json)

```
$ kubectl delete pod,service foo
```

Elimina pods e services con il nome "foo"

3.4.3 Aggiornare e Visualizzare risorse

```
# List all services in the namespace
$ kubectl get services

# List all pods in all namespaces
$ kubectl get pods --all-namespaces

# List a particular deployment
$ kubectl get deployment my-dep

# List all pods in the namespace
$ kubectl get pods

# Get a pod's YAML
$ kubectl get pod my-pod -o yaml
```

Un comando molto utilizzato per mostrare le risorse principali di tutti i namespace è il seguente:

```
$ kubectl get all -A
```

o equivalentemente:

```
$ kubectl get all --all-namespaces
```

In output avremo qualcosa di simile a:

```
NAMESPACE           NAME                                                 READY   STATUS    RESTARTS      AGE
kube-system         pod/coredns-5d78c9869d-92tmt                         1/1     Running   0             24m
kube-system         pod/coredns-5d78c9869d-hdp8w                         1/1     Running   0             24m
kube-system         pod/etcd-kind-control-plane                          1/1     Running   0             25m
kube-system         pod/kindnet-4lxr8                                    1/1     Running   0             24m
kube-system         pod/kube-apiserver-kind-control-plane                1/1     Running   1 (25m ago)   25m
kube-system         pod/kube-controller-manager-kind-control-plane       1/1     Running   0             25m
kube-system         pod/kube-proxy-9mvs6                                 1/1     Running   0             24m
kube-system         pod/kube-scheduler-kind-control-plane                1/1     Running   0             25m
local-path-storage  pod/local-path-provisioner-6bc4bddd6b-vz9gv           1/1     Running   0             24m

NAMESPACE     NAME                 TYPE        CLUSTER-IP   EXTERNAL-IP   PORT(S)                  AGE
default       service/kubernetes   ClusterIP   10.96.0.1    <none>        443/TCP                  25m
kube-system   service/kube-dns     ClusterIP   10.96.0.10   <none>        53/UDP,53/TCP,9153/TCP   25m

NAMESPACE     NAME                        DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR            AGE
kube-system   daemonset.apps/kindnet      1         1         1       1            1           kubernetes.io/os=linux   24m
kube-system   daemonset.apps/kube-proxy   1         1         1       1            1           kubernetes.io/os=linux   25m

NAMESPACE           NAME                                  READY   UP-TO-DATE   AVAILABLE   AGE
kube-system         deployment.apps/coredns               2/2     2            2           25m
local-path-storage  deployment.apps/local-path-provisioner   1/1     1            1           24m

NAMESPACE           NAME                                             DESIRED   CURRENT   READY   AGE
kube-system         replicaset.apps/coredns-5d78c9869d               2         2         2       24m
local-path-storage  replicaset.apps/local-path-provisioner-6bc4bddd6b   1         1         1       24m
```

Da ciò si possono estrapolare numerose informazioni rilevanti come lo stato della risorsa, quante volte sia stata riavviata, il numero di repliche richieste e molte altre.

```
$kubectl -v=999 get pods -A
```

Per analizzare richieste HTTP all'API server di Kubernetes (ad esempio quali richieste HTTP fa un certo comando kubectl) basterà aggiungere *-v=999*.

```
$kubectl cluster-info dump
```

Mostra informazioni dettagliate sul cluster Kubernetes.

```
$kubectl api-resources
```

Mostra tutte le risorse Kubernetes note.

```
kubectl explain <resource>
```

Mostra la documentazione riguardante la <resource>(e.g. kubectl explain pods). Si possono ottenere informazioni anche sui fields interni alla risorsa selezionata:

```
$ kubectl explain pods.spec
```

(Quando sono stati trattati in dettaglio i Pod è stata data una dimostrazione pratica di come crearlo tramite file manifest. Da notare come il campo *spec* sia presente. Con il comando *explain* si avranno tutte le informazioni relative a quel determinato campo.)

3.5 Kind

Kind è uno strumento che semplifica la creazione di un cluster Kubernetes in locale su una singola macchina. È utile per sviluppatori e operatori che vogliono testare applicazioni Kubernetes in un ambiente controllato e isolato. Sebbene Kind offra una buona simulazione di un cluster Kubernetes, è principalmente pensato per lo sviluppo locale, l'apprendimento e l'esperimento. Poiché opera solo in una VM su un singolo nodo, non fornisce l'affidabilità di un cluster Kubernetes distribuito. Per installare Kind (facendo riferimento al sistema operativo Linux) si può digitare da cli il comando:

```
$[ $(uname -m) = x86_64 ] && curl -Lo
 ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
```

Figura 13: Kind create cluster

Una volta installato si potrà creare un cluster locale eseguendo il comando in figura 13:

```
$ kind create cluster
```

Questo creerà una VM locale, provvederà a Kubernetes e creerà una configurazione locale di kubectl che punta a quel cluster. Quando si desidera rimuovere il cluster, potrà essere eseguito:

```
$ kind delete cluster
```

La precedente sezione è stata scritta utilizzando l'ausilio della documentazione ufficiale di Kind [8].

3.6 ArgoCD

Argo CD (logo in figura 14), come suggerisce già il nome, è uno strumento di Continuous Delivery (CD) per Kubernetes. Prima di approfondire Argo CD come strumento CD, è utile comprendere come la pipeline (Continuous Integration/Delivery) è implementata nella maggior parte dei progetti.

Figura 14: ArgoCD (immagine ripresa dal sito ufficiale [1])

- **Continuous Integration (CI):** si consideri un'applicazione basata su microservizi eseguita in un cluster Kubernetes. Quando ci sono modifiche nel codice dell'applicazione, ad esempio l'aggiunta di una nuova funzionalità o una correzione di bug, la pipeline di CI viene automaticamente attivata: verranno testate le modifiche, si costruirà l'immagine Docker, verrà effettuata la push in un repository Docker e infine verrà aggiornato il file manifest Kubernetes, come deployment.yaml.
- **Continuous Delivery (CD):** utilizzando strumenti come kubectl, verrà fatto l'apply del file di deployment aggiornando così il cluster Kubernetes.

Tuttavia, ci sono alcune sfide con questo approccio. Se, ad esempio, venissero effettuate delle modifiche su file differenti, occorrerebbe effettuare l'apply delle modifiche su ognuno di questi. Inoltre, probabilmente il problema più rilevante, è che una volta che dovesse venir distribuita l'applicazione su Kubernetes o vengono applicate eventuali modifiche alla configurazione di Kubernetes, non si avrebbe un'ulteriore visibilità sullo stato del deployment. Dopo l'esecuzione di "kubectl apply", non si saprebbe lo stato effettivo di quell'esecuzione: l'applicazione è stata effettivamente creata? È in uno stato sano o sta fallendo all'avvio? Queste informazioni possono essere conosciute solo eseguendo passi di test successivi. Pertanto, la parte di Continuous Delivery della pipeline, quando si lavora specificamente con Kubernetes, può essere migliorata e resa più efficiente. Argo CD è stato costruito per questo specifico caso d'uso.

Come riesce Argo CD a rendere il processo di Continuous Delivery più efficiente e affrontare le sfide sopra citate? Fondamentalmente, è stato invertito il flusso: anziché accedere esternamente al cluster dagli strumenti CI/CD, è lo strumento stesso fa parte del cluster. Invece di effettuare la push delle modifiche al cluster, si utilizza un flusso di lavoro pull, dove un agente nel cluster (Argo CD) effettua la pull delle modifiche e le applica.

Come prima cosa, occorre integrare Argo CD nel cluster utilizzando il comando:

```
$ kubectl create namespace argocd

$ kubectl apply -n argocd -f https://raw.githubusercontent.com/
```

```
argoproj/argo-cd/stable/manifests/install.yaml
```

In questa configurazione, Argo viene istruito ad accedere a un repository Git e monitorare le modifiche. Quando queste vengono rilevate, ArgoCD si occupa automaticamente di applicarle nel cluster. Ad esempio, quando gli sviluppatori effettuano un commit delle modifiche nel repository del codice sorgente dell'applicazione, la pipeline CI avvia automaticamente un processo di build. Questo processo include il testing delle modifiche, la creazione e il push dell'immagine nel repository e, infine, l'aggiornamento del file manifest Kubernetes, come deployment.yaml, con la nuova versione dell'immagine. Argo CD nel cluster rileva immediatamente queste modifiche nel repository Git e le applica automaticamente nel cluster.

Ora non importa che la modifica venga fatta da processi automatizzati o da team di sviluppo, Argo CD manterrà sempre il cluster sincronizzato. Come risultato, si avranno pipeline CI e CD separate, dove la pipeline CI viene gestita principalmente dagli sviluppatori e la pipeline CD è gestita dagli Ops o dai team DevOps e configurata utilizzando Argo CD. In questo modo si avrà comunque una pipeline CI/CD automatizzata, ma con una separazione delle responsabilità, dove team diversi sono responsabili di diverse parti della pipeline.

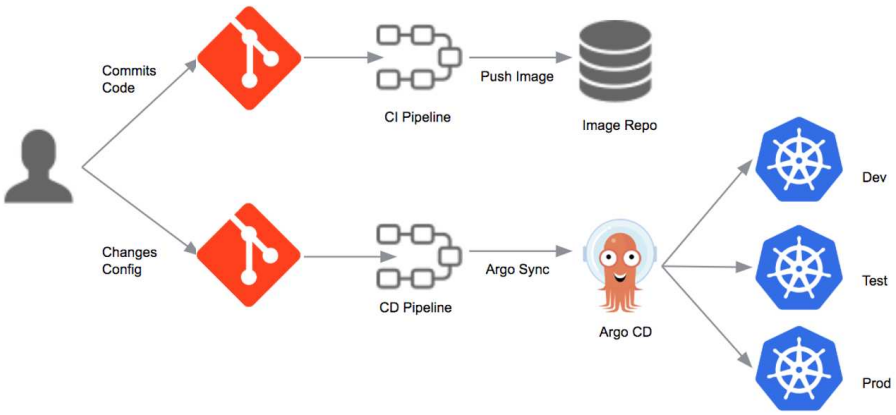

Figura 15: CI/CD pipeline

Di seguito sono riportati alcuni dei concetti specifici di Argo CD:

- **Application:** un gruppo di risorse Kubernetes definite da un file manifest.
- **Application source type:** il tipo di strumento utilizzato per compilare l'applicazione.

- **Target state:** lo stato desiderato di un'applicazione, rappresentato dai file in un repository Git.
- **Live state:** lo stato attuale dell'applicazione.
- **Sync status:** indica se lo stato attuale corrisponde allo stato desiderato. L'applicazione distribuita è identica a quanto indicato in Git?
- **Sync:** il processo che porta un'applicazione al suo stato desiderato, ad esempio, applicando modifiche a un cluster Kubernetes.
- **sync operation status:** Indica se un'operazione di sincronizzazione è riuscita.
- **refresh:** Confronta il codice più recente in Git con lo stato attuale. Capisce cosa è diverso.
- **health:** L'applicazione, sta funzionando correttamente? Può gestire le richieste?

La precedente sezione è stata scritta utilizzando l'ausilio della documentazione ufficiale di ArgoCd [1] e del video [14].

3.7 GitOps

GitOps (figura 16) è un paradigma operativo che sfrutta il sistema di controllo versione Git per gestire l'intero ciclo di vita delle applicazioni. In questo approccio, Git diventa la singola fonte di verità per lo stato del sistema, e le operazioni di deployment sono integrate direttamente nel repository Git. Nel

Figura 16: Logo (immagine ripresa da [17] e modificata).

contesto di GitOps, il repository Git contiene lo stato desiderato del sistema, compresi i file manifest Kubernetes, le configurazioni delle applicazioni e altri artefatti di deployment. Ogni modifica nello stato del sistema viene riflessa nei file del repository Git.

Questo approccio offre diversi benefici:

1. **Maggiori osservabilità:** poiché lo stato desiderato del sistema è chiaramente definito nel repository Git, è possibile monitorare le modifiche nel tempo e comprendere facilmente l'evoluzione del sistema.

2. **Sicurezza:** le modifiche vengono apportate attraverso pull request e revisioni del codice, garantendo un controllo e una revisione accurati prima del deployment. Inoltre, le modifiche sono tracciate storicamente nel repository Git.

3. **Produttività:** GitOps semplifica il deployment delle applicazioni, in quanto le operazioni di deployment sono gestite tramite Git. Gli sviluppatori possono utilizzare gli stessi strumenti e processi per gestire sia lo sviluppo che il deployment.

Precedente è stato descritto come ArgoCD inverta proprio questo processo: invece di mappare ogni modifica dello stato in un repository Git, viene mappato quest'ultimo nello stato di Kubernetes. Nonostante ciò, se per avere uno sviluppo più veloce e testare delle modifiche direttamente in locale, uno sviluppatore effettuasse i cambiamenti direttamente in Kubernetes? Ebbene, Argo DC rileva i cambiamenti bidirezionalmente. Se le versioni di GitOps e Kubernetes non combaciano, vengono sovrascritti tutti i cambiamenti effettuati in Kubernetes, lasciando Git come unica forma di verità. Questo elimina il problema dell'inconsistenza dei dati, anche perché potrebbero esserci più cluster che attingono allo stesso repository.

Una buona norma divenuta pratica ottimale è quella di avere repository separati per il codice sorgente dell'applicazione e il codice di configurazione dell'applicazione. Uno dei motivi principali è che il codice di configurazione dell'applicazione non riguarda solo il file di deployment, ma potrebbero esserci configmap, secret, service, ingress e così via. I file manifest Kubernetes per l'applicazione dovrebbero essere ospitati nel proprio repository Git, mentre tutto ciò di cui l'applicazione potrebbe aver bisogno per essere eseguita nel cluster in un repository separato. Questo perchè i file manifest possono cambiare completamente in modo indipendente e con maggiore frequenza rispetto al codice sorgente. In questo modo, quando si aggiorna un file di configurazione dell'applicazione non si effettua l'intera pipeline CI in quanto il codice stesso non è cambiato, ma verranno tracciati unicamente i cambiamenti dei manifest interessati, lasciando immacolato il sorgente.

La precedente sezione è stata scritta utilizzando l'ausilio della documentazione ufficiale di GitLab [5].

4 Realizzazione della rete e Comandi

In questo capitolo verranno affrontati tutti i passaggi che porteranno alla verifica della vulnerabilità. Molte informazioni sono state riprese dalle fonti sopra citate, in particolare molti passaggi sono stati estrapolati e modificati dalla documentazione ufficiale di ArgoCD [1].

4.1 Creazione cluster

Alla base del cluster ci sono dei componenti essenziali. In questa sezione verranno installati tutti i pezzi del puzzle per poi poterli mettere insieme. I componenti sono stati adeguatamente discussi nei capitoli precedenti, dunque verrà fornito solo un metodo pratico per lavorare con essi.

- **Docker**: alla base dell' applicazione abbiamo docker, piattaforma cruciale per la gestione e l'esecuzione di container. Per l'installazione rimandiamo alla pagina ufficiale `https://www.docker.com/`;
- **Kind**: questo strumento permetterà di ricreare un cluster kubernetes in locale. Kind è stato affrontato nel capitolo 3.5. Per la sua installazione è necessario effettuare da cli:

  ```
  $[ $(uname -m) = x86_64 ] && curl -Lo
   ./kind https://kind.sigs.k8s.io/dl/v0.20.0/kind-linux-amd64
  ```

  A seguito dell'installazione basterà digitare il comando:

  ```
  $ kind create cluster
  ```

  per realizzare il cluster completamente funzionante
- **Kubectl**: fornisce uno strumento a riga di comando per comunicare con il Control Plane di un cluster Kubernetes. Kubectl è stato trattato nel capitolo 3.4 insieme a numerosi comandi (ne verranno descritti molti altri in seguito nel seguente capitolo). Di seguito è riportata l'installazione tramite CLI:

  ```
  $curl -LO "https://dl.k8s.io/release/
    $(curl -L -s https://dl.k8s.io/release/stable.txt)/
    bin/linux/amd64/kubectl"
  ```

4.2 Configurazione ArgoCD

In questa sezione verrà integrato ArgoCd al cluster Kubernetes.

4.2.1 Installazione

Come prima cosa l'installazione.

```
$ kubectl create namespace argocd
```

Una volta creato un namespace per ArgoCD si procede all'installazione vera e propria. La documentazione ufficiale consiglia di installare sempre l'ultima versione stabile attraverso il comando:

```
$ kubectl apply -n argocd -f https://raw.githubusercontent.com/
  argoproj/argo-cd/stable/manifests/install.yaml
```

La vulnerabilità però affligge solo determinate versioni dell'applicazione. In questo caso specifico dalla 2.8.0 alla 2.8.2 incluse. Quindi occorre modificare unicamente la parola **stable** con la versione desiderata preceduta da una 'v', come nel seguente esempio (nel progetto è stata installata la versione la 2.8.2):

```
$ kubectl apply -n argocd -f https://raw.githubusercontent.com/
  argoproj/argo-cd/v2.8.2/manifests/install.yaml
```

Una volta completata l'installazione di ArgoCD nel cluster è consigliabile verificare la corretto completamento della procedura attraverso il comando kubetcl:

```
$ kubectl get deployment argocd-server -n argocd -o
  jsonpath='{.spec.template.spec.containers[0].image}'
```

L'output dovrebbe essere una cosa del tipo:

```
PS C:\Users\Mattia\OneDrive\Desktop>  kubectl get deployment argocd-server -n argocd
-o jsonpath='{.spec.template.spec.containers[0].image}'
quay.io/argoproj/argocd:v2.8.2
```

Figura 17: ArgoCD version

4.2.2 Interfaccia grafica e login

Una volta effettuata l'installazione è possibile gestire Argo sia da linea di comando che tramite interfaccia grafica, semplificando le interazioni.

Come prima cosa si digiti il comando:

```
$ kubectl port-forward svc/argocd-server -n argocd 8080:443
```

in questo modo si mappa la porta 443 del server di ArgoCD sulla 8080. Cercando poi l'indirizzo *https://localhost:8080* nell'url di un qualsiasi motore di ricerca è possibile vedere la pagina illustrata in figura 18. L'username iniziale è sempre *admin*, mentre la password è differente ogni volta. Per ottenerla occorre digitare il comando:

```
$ kubectl get secret argocd-initial-admin-secret -n argocd -o yaml
```

Figura 18: ArgoCD login

o equivalentemente:

```
$ kubectl get secret argocd-initial-admin-secret -n argocd -o json
```

L'output sarà una cosa del tipo:

```
apiVersion: v1
data:
  password: eVFQRWh3cDhJVFB5OVhYaQ==
kind: Secret
metadata:
  creationTimestamp: "2023-12-18T09:33:59Z"
  name: argocd-initial-admin-secret
  namespace: argocd
  resourceVersion: "17412"
  uid: da75f40a-a0c8-4181-9917-4213b04ab3c3
type: Opaque
```

(Questo output è in formato yaml, il formato json sarà molto simile). La stringa della password è codificata in base64, per decodificarla basta digitare il comando:

```
$ echo eVFQRWh3cDhJVFB5OVhYaQ== | base64 --decode
```

o equivalentemente utilizzare un convertitore online. Una volta ottenuta la password decodificata si effettui il login.

Una volta effettuato l'accesso si dovrebbe avere una schermata simile a quella riportata in figura 19.

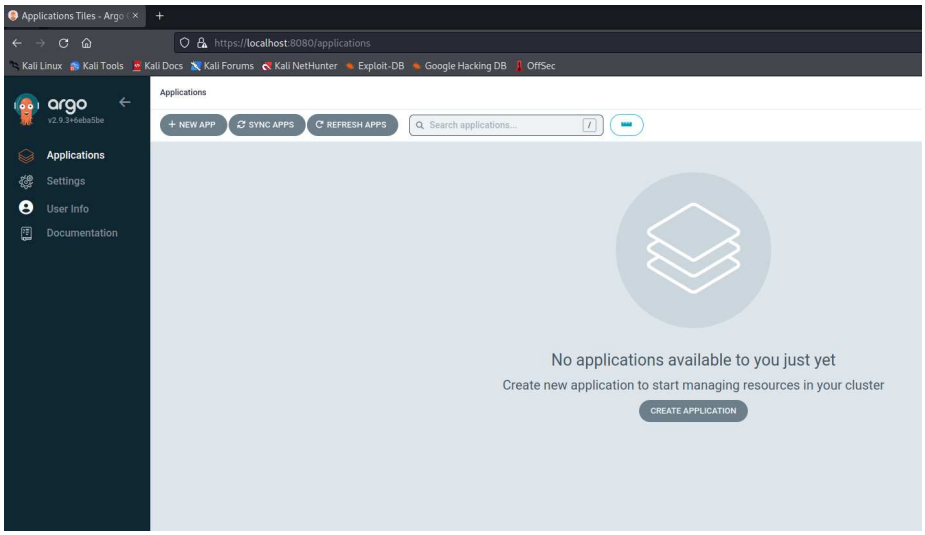

Figura 19: ArgoCD home

4.3 Creazione repository GitOps

In questa sezione verrà creato il repository privato su GitOps (la gestione risulta molto simile a GitHub). Una volta effettuata la registrazione, creato il primo progetto e collegato il repository con la directory in locale (in modo da poter lavorare sulla propria macchina e fare la push delle modifiche in qualsiasi momento), è stato svolto tutto il necessario.

In questo modo si dovrebbe avere un "contenitore" vuoto per i file di configurazione. I prossimi passi sono:

- Creazione dei file manifest per la nostra applicazione.
- Collegamento repository ad ArgoCD e al cluster kubernetes.

Il repository è consultabile all'indirizzo [6].

4.4 Creazione manifest applicazione

Seguendo quanto imparato nei capitoli precedenti, in questa sezione verrà configurata l'applicazione vera e propria. Per quanto molto semplice, si cercherà di renderla il più completa possibile. Il tipico *microservizio stateless*[6] che verrà ricreato sarà composto da:

- Deployment

[6]Il microservizio non conserva informazioni sullo stato dell'applicazione tra diverse richieste.

- Service
- Config Map
- Secret

Un consiglio molto importante: durante la scrittura dei file .yaml, molte volte capiterà di effettuare errori di sintassi in quanto il linguaggio risulta *case-sensitive*[7], sensibile agli spazi bianchi e anche all'indentazione. Per evitare errori banali la cosa più giusta sarebbe installare plugin specifici per interpretare il formato direttamente nell'IDE (ambiente di sviluppo integrato) che si sta utilizzando oppure (come effettuato per questo progetto) installare un validatore quale **kubeconform**.

Per analizzare un file basterà digitare:

```
$ kubeconform <nome_file.yaml>
```

Un esempio pratico:

```
$ kubeconform .\Tesi\vuln\ConfigMap.yaml

.\Tesi\vuln\ConfigMap.yaml - failed validation:
    error unmarshalling resource:
    error converting YAML to JSON: yaml:
    line 2:
    mapping values are not allowed in this context
```

Come si può notare, il validatore non riesce a fare il parsing del file, restituendo la linea in cui è presente l'errore e una piccola descrizione. Per avere maggiori informazioni riguardanti il file analizzato si può digitare (con relativo output come nell'esempio):

```
$ kubeconform -summary -output json <nome_file.yaml>
```

Un esempio:

```
$ kubeconform -summary -output json .\Tesi\vuln\Deployment.yaml
{
  "resources": [],
  "summary": {
    "valid": 1,
    "invalid": 0,
    "errors": 0,
    "skipped": 0
  }
}
```

Con l'ausilio di questo strumento verranno creati ed analizzati uno per uno i file di configurazione.

[7] Sensibile alle lettere maiuscole.

4.4.1 Deployment

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-deployment
  labels:
    app: my-application
spec:
  selector:
    matchLabels:
      app: my-application
  replicas: 5
  strategy:
    type: RollingUpdate
  template:
    metadata:
      labels:
        app: my-application
    spec:
      containers:
      - name: nginx
        image: nginx
        resources:
          limits:
            memory: 200Mi
          requests:
            cpu: 100m
            memory: 200Mi
        ports:
        - containerPort: 80
      livenessProbe:
        httpGet:
          path: /
          port: 80
        initialDelaySeconds: 5
        periodSeconds: 5
```

Come è possibile vedere dall'etichetta *kind: Deployment* si sta dichiarando un oggetto di tipo Deployment. Il Deployment è denominato "my-deployment", e ha un'etichetta "app: my-app" associata.

Passando alla sezione delle specifiche vere e proprie (*spec*). In questa sezione si sta dicendo a Kubernetes di gestire i Pod con l'etichetta *app: my-application* e di mantenere 5 repliche di questi Pod (*replicas: 5*). La strategia di aggiornamento è di tipo *RollingUpdate*, che significa che gli aggiornamenti vengono

applicati gradualmente, evitando downtime.

Nel blocco *template* all'interno di *spec* si definisce il modello per la creazione di nuovi Pod. Ogni Pod avrà l' etichetta *app: my-application*. All'interno del Pod, si dichiara un singolo container chiamato "nginx" basato sull'immagine "nginx". Si sta anche impostando un limite alla memoria utilizzata e un massimo alle richieste di risorse per il container, esponendo la porta 80.

Infine, si ha una sezione per controllare vitalità del container (*livenessProbe*). Questa configurazione stabilisce che Kubernetes verifichi la vitalità del container eseguendo una richiesta HTTP al percorso "/" sulla porta 80, iniziando dopo un ritardo iniziale di 5 secondi e ripetendo la verifica ogni 5 secondi. Se la verifica dovesse fallire consecutivamente per più volte il pod verrà riavviato.

4.4.2 Service

```yaml
apiVersion: v1
kind: Service
metadata:
  name: My-Service
spec:
  selector:
    app: my-application
  ports:
  - protocol: TCP
    port: 80
    targetPort: 8080
  type: ClusterIP
```

Nel file Service.yaml, viene definito un oggetto di tipo Service denominato "My-Service".

Nella sezione *spec*, viene configurato il comportamento del servizio. La chiave *selector* definisce quali Pod saranno selezionati dal servizio. In questo caso, vengono selezionati i Pod che hanno l'etichetta *app: my-application* (era stata applicata questa etichetta attraverso la chiave *labels* nel Deployment).

La chiave *ports* specifica come i servizi saranno esposti. In questo caso si sta dicendo che si vuole esporre il servizio sulla porta 80 del Service stesso, inoltrando il traffico ai Pod sulla porta 8080. Il protocollo utilizzato è TCP.

Infine, la chiave *type* definisce il tipo di servizio. In questo caso, è impostato su *ClusterIP*, che assegna un indirizzo IP al service accessibile solo all'interno del cluster.

4.4.3 ConfigMap

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: My-ConfigMap
data:
  key1: value1
  key2: value2
```

Nel file ConfigMap.yaml, viene definito un oggetto di tipo ConfigMap denominato "My-ConfigMap".

Il file è stato pensato unicamente a scopo didattico, rimane dunque minimale. Infatti ha unicamente due coppie chiave-valore puramente a scopo didattico che sono la chiave "key1" è associata al valore "value1" e la chiave "key2" è associata al valore "value2". Non avendo bisogno di un metodo per configurare o fornire dati all'interno del nostro cluster, va benissimo lasciare la config map così, disponibile per futuri aggiornamenti.

4.4.4 Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: My-Secret
type: Opaque
stringData:
  username: Username
  password: Password
```

Nel file Secret.yaml viene definito un oggetto di tipo Secret denominato "my-secret". Questo è di tipo *Opaque*, indicando che può contenere dati arbitrari non strutturati.

All'interno del Secret, sono inclusi i seguenti dati sensibili:

- Username: il nome utente associato al Secret, in questo caso specificato come **"Username"**.
- Password: la password associata al Secret, in questo caso specificata come **"Password"**.

Queste informazioni possono essere utilizzate all'interno di Kubernetes, consentendo alle applicazioni di accedere in modo sicuro a queste credenziali senza doverle esporre direttamente nei manifest delle applicazioni stesse. Seppur si possa essere portati a credere che la vulnerabilità sia rappresentata dai dati contenuti nei Secret di Kubernetes, non è in realtà così. Questi infatti sono

estremamente sicuri ed è possibile la lettura solo da utenti privilegiati. Una volta creati i file manifest ed effettuato la push, si dovrebbe avere qualcosa di simile alle figure 20 e 21.

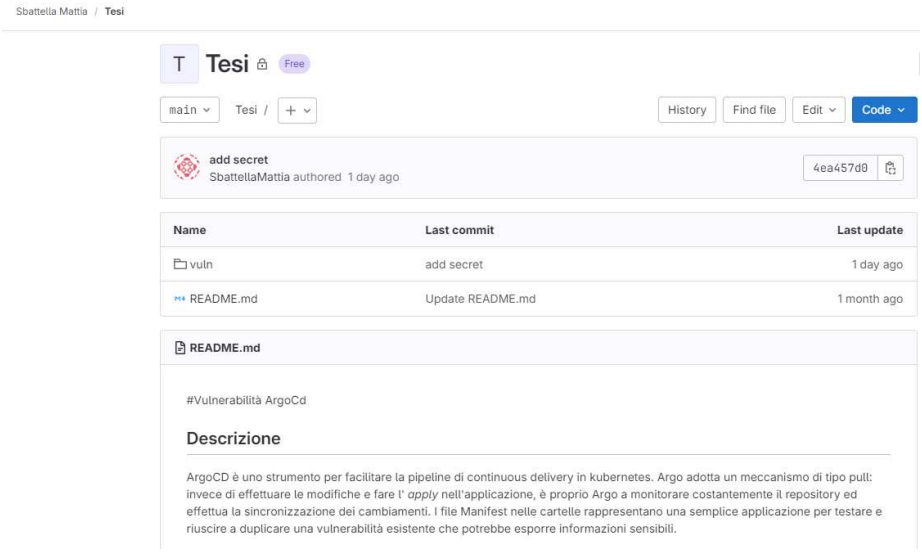

Figura 20: GitOps repository

Figura 21: GitOps repository

4.5 Creazione del Cluster Secret

La vulnerabilità vera e propria non risiede nei Secret kubernetes dell'applicazione, ma nei Secret gestiti da ArgoCD. In Argo, credenziali appartenenti al repository, vengono salvate in questi file. Per questo motivo si ha il bisogno di un ulteriore file denominato *cluster-secret.yaml* (qualsiasi altro nome va bene). Creato il file bisogna aggiungere il seguente codice di configurazione:

```
apiVersion: v1
kind: Secret
metadata:
```

```
  name: mycluster-secret
  namespace: argocd
  labels:
    argocd.argoproj.io/secret-type: cluster
type: Opaque
stringData:
  name: in-cluster
  server: https://kubernetes.default.svc
  config: |
    {
      "Username": "SbattellaMattia",
      "Password": "PASSW0RD_SUP3R_SEGRETA"
    }
```

Il Secret deve avere:

- **Namespaces**: in questo caso verrà aggiunto al namespace *"argocd"*.
- **Labels**: fondamentale, impostato su *argocd.argoproj.io/secret-type: cluster*. Solo i file Secret di tipo **cluster** sono soggetti alla vulnerabilità.
- **Name**: nome del gruppo, impostato come *in-cluster*.
- **Server**: URL del server API del cluster, impostato come *"https://kubernetes.default.svc"*.
- **Config**: rappresentazione JSON della struttura dati: in questo caso abbiamo Username e Password del repository kubernetes.

Una volta creato il file deve essere effettuato l'apply (non si tratta di un file Kubernetes, non deve essere immesso nel repository online) con il comando:

```
$kubectl apply -f ./cluster-secret.yaml.yaml
```

Molto spesso i Secret contengono dati simili a quelli immessi, potenzialmente pericolosi se dovessero trapelare. La pericolosità della vulnerabilità è rappresentata proprio da questo piccolo (per modo di dire) particolare. Come descritto nel capitolo 2.2 ogni volta che viene effettuato l'*apply* di un file (indifferentemente che sia da riga di comando o da interfaccia grafica di Argo, in modo automatico o meno) verrà sovrascritto il contenuto di *last-applied-configuration annotation* con il contenuto del vecchio file, esponendo tutte le informazioni sensibili in esso. Nel prossimo capitolo verrà spiegato meglio quanto accennato.

4.6 Collegamento del Repository GitOps ad ArgoCD

Se tutti i passaggi descritti finora sono stati effettuati correttamente, si dovrebbe avere il repository GitOps contenente i file manifest e un cluster in locale con Kubernetes e ArgoCd correttamente configurati. Il prossimo passo è collegare questi due ambienti.

4.6.1 Generazione della chiave SSH

Ci sono numerosi modi per farlo, ma in questo progetto è stato utilizzato il protocollo SSH (Secure Shell). Il protocollo consiste in una coppia di chiavi crittografiche asimmetriche utilizzate per autenticare utenti e consentire l'accesso sicuro a server remoti attraverso una rete. La coppia di chiavi è composta da una chiave privata e una chiave pubblica.

- **Chiave privata**: è segreta e deve essere mantenuta in modo sicuro sul dispositivo. Viene utilizzata per firmare digitalmente le richieste di accesso e per decriptare le informazioni inviate alla chiave pubblica corrispondente.
- **Chiave pubblica**: può essere distribuita liberamente e viene associata al proprio account su un server remoto. Viene utilizzata per verificare le firme digitali create dalla chiave privata e per crittografare le informazioni che solo la chiave privata può decifrare.

Quando si vuole accedere ad un server remoto, si carica la chiave pubblica sul server. Quando ci si connette, il server verifica l' identità richiedendo una firma digitale con la chiave privata. Se la firma è valida (cioè corrisponde alla chiave pubblica associata all'account), l'accesso viene concesso. Nell'applicazione però, è Argo che deve continuamente monitorare lo stato del repository. Dunque ArgoCD dovrà essere in possesso della chiave privata e GitOps della chiave pubblica.

L'utilizzo di chiavi SSH è più sicuro rispetto alle password tradizionali, poiché fornisce un metodo di autenticazione basato su crittografia a chiave pubblica e privata, riducendo il rischio di accessi non autorizzati.

Come prima cosa bisogna generare un paio di chiavi SSH. Digitando tramite cli il comando:

```
$ ssh-keygen -t rsa -b 4096 -C "example@email.com"
```

si genera una chiave RSA (algoritmo di cifratura asimmetrica) di 4096 bit associata all'indirizzo email specificato (l'indirizzo e-mail può essere omesso tranquillamente, ma è buona prassi metterlo per differenziare diverse chiavi). Una volta digitato il comando basterà accettare le posizioni predefinite del file chiave e lasciare vuota (o immettere) la passphrase. La passphrase è una password opzionale che può essere associata alla chiave privata per fornire un ulteriore livello di sicurezza. Se si sceglie di impostare una passphrase, dovrà essere inserita ogni volta che si utilizza la chiave privata.

Fatto ciò, il paio di chiavi sarà disponibile nel file di output predefinito (solitamente .ssh/id_rsa.pub per la chiave pubblica e .ssh/id_rsa per la privata).

4.6.2 Aggiunta chiave SSH a GitOps

È possibile aggiungere la chiave **pubblica** a GitOps seguendo i seguenti passaggi:

1. Accedere al profilo su GitOps
2. Premere l'icona del profilo e cliccare la voce: *Edit profile –>SSH Keys*;
3. Premere il pulsante *Add new key*;
4. Nel campo *Key* aggiungere la chiave pubblica;
5. Nel campo *Title* mettere un nome a scelta. Nel progetto è stato scelto il titolo "Tesi key";
6. Nel campo *Usage type* lasciare la voce "Authentication e Signing";
7. Nel campo *Expiration date* impostare la validità che ritenuta opportuna. Nel progetto è stato impostato il periodo di un anno.

Il risultato sarà simile a come riportato in figura 22.

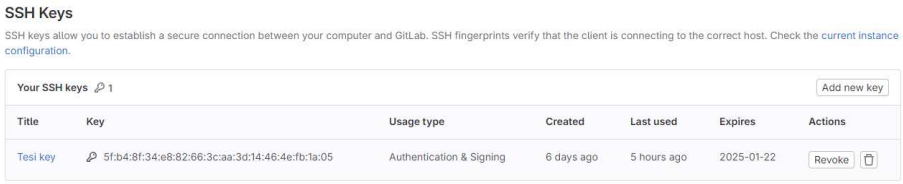

Figura 22: GitOps keys

4.6.3 Collegamento di ArgoCD tramite chiave SSH

È possibile collegare ArgoCD al repository GitOps tramite la chiave **privata**, seguendo i seguenti passaggi:

1. Effettuare il login come descritto nel capitolo 4.2.2;
2. Cliccare la voce: *Settings –>Repositories*;
3. Premere il pulsante *Connect repo*;
4. Nel campo *Connection method* selezionare "Via SSH";
5. Nel campo *Name* immettere un nome a scelta o ometterlo;
6. Nel campo *Project* selezionare "default";
7. Nel campo *Url* copiare l'URL del repository GitOps creato precedentemente. Occore precisare che è necessario prendere l'URL relativo alla voce *Clone with SSH* e non il comune Https.

8. Nell'ultimo campo aggiungere la chiave SSH privata e premere il pulsante *Connect*

Se configurato tutto correttamente il risultato sarà come in figura 23. Inoltre,

Figura 23: ArgoCD connesso al repository GitOps

tornando alla home, dovremmo avere la seguente schermata (figura 24):

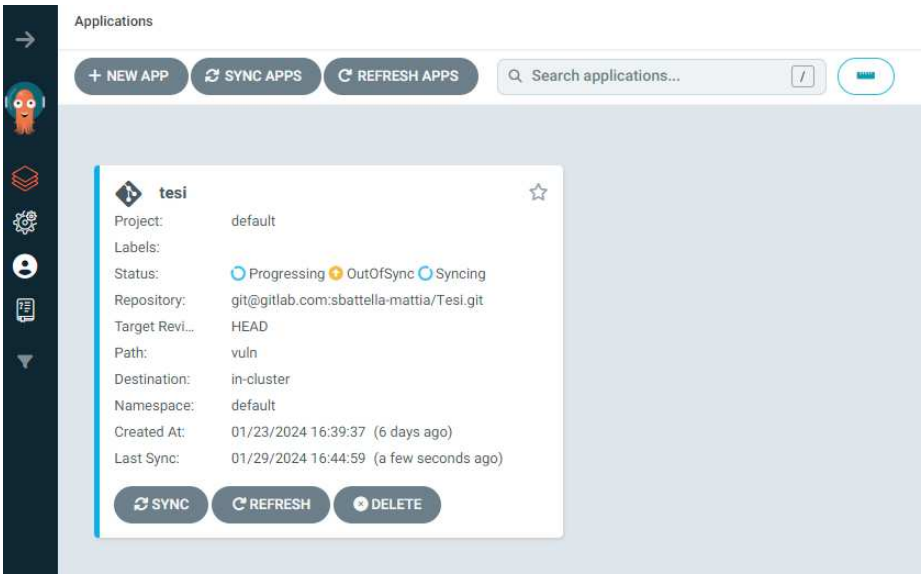

Figura 24: Home ArgoCd

4.7 Creazione utente non privilegiato

Da admin del cluster, si ha la possibilità di vedere in chiaro le informazioni contenute in qualsiasi file Secret. Questo non sarebbe possibile (senza vulnerabilità) se si effettuasse l'accesso come utente non privilegiato. Quindi per verificare quanto detto, occorre creare un nuovo utente.

Argo CD implementa un **C**ontrollo degli **A**ccessi **B**asato su **R**uoli (**RBAC**) per garantire agli utenti l'accesso con i minimi privilegi necessari. È importante notare che, di default, Argo CD fornisce solo due ruoli:

- **Admin**: utente super-amministratore con accesso completo alle risorse di Argo CD.

- **Readonly**: utente con privilegi di sola visualizzazione su tutte le risorse.

Digitando il comando:

```
$argocd account list
```

è possibile notare come sia presente solo un account **admin**. Come prima operazione occorre cambiare la password dell'admin, per evitare di replicare ogni volta il procedimento nella sezione 4.2.2. La procedura è molto semplice tramite UI, basterà andare nell'area riservata, cliccare su *Update passoword* e compilare i campi come mostrato in figura 25, inserendo la password utilizzata per il primo accesso, e scegliendo la nuova password (nell'esempio riportato è stata scelta la password: **'superadmin'**).

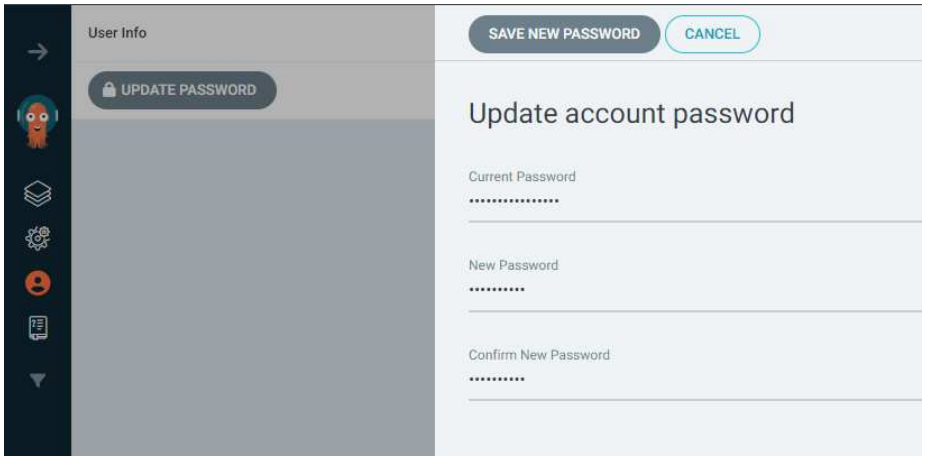

Figura 25: Update password (UI)

Una volta cambiata la password, si deve creare l'utente non privilegiato.

L'utente può essere creato sia manualmente, tramite CLI, sia tramite UI. Questa volta è stato adottato un approccio più tecnico. Per il passo successivo bisogna collegarsi al server di Argo inizializzato sulla porta *8080* di *localhost*. Per effettuare il login tramite admin basterà digitare il comando:

```
$argocd login localhost:8080
```

e successivamente dare la conferma di connessione non sicura, seguita da username e password appena modificati. Avendo effettuato l'accesso, si crei il nuovo utente digitando:

```
$ argocd account update-password \
  --account <name> \
  --current-password <current-user-password> \
  --new-password <new-user-password>
```

dove al posto di *current-user-password* deve essere immessa la password dell'admin, in quanto prima volta. Quindi:

```
$ argocd account update-password
    --account prova
    --current-password superadmin
    --new-password prova000
```

Con i seguenti passaggi è stato creato l'utente **prova** con password **prova000** come illustrato anche in figura 26. L'ultimo passaggio fondamentale è quello di impostare il ruolo del nuovo utente come **Readonly**. Per effettuare correttamente questa procedura, occorre modificare la ConfigMap di Argo contenente l'RBAC access. Con il comando:

```
$ kubectl edit cm -n argocd argocd-rbac-cm
```

si aprirà il file contenente lo yaml dell'oggetto. Basterà aggiungere il campo *data* contenente il ruolo dell'utente:

```
data:
  policy.default: role:readonly
```

(Il campo *data* è possibile collocarlo dove si vuole, ma sarebbe preferibile in fondo al file. È importante ricordare come sia essenziale rispettare l'indentazione.)

Una volta terminato, si può effettuare il logout attraverso:

```
$argocd logout
```

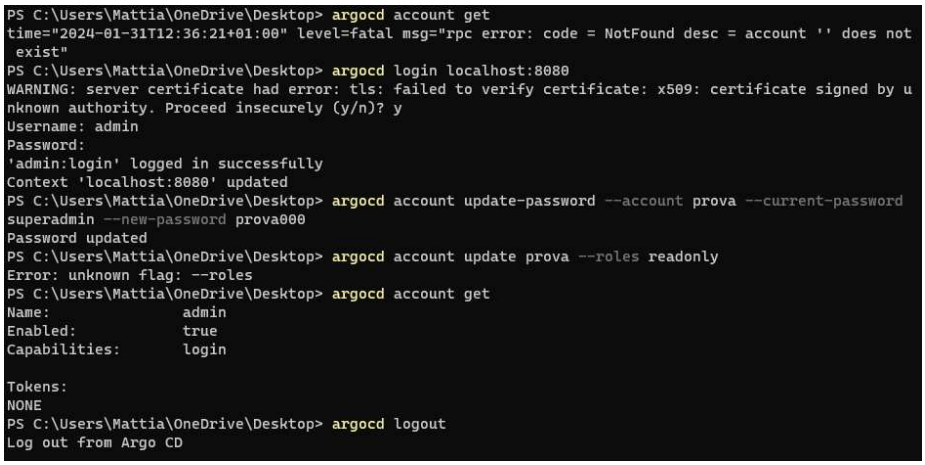

Figura 26: Update password (CLI) e creazione utente 'prova'

È utile notare come sia molto importante il **context**. Altro non è che il contesto che si sta utilizzando, ovvero determina a quale cluster Kubernetes si sta

inviando i comandi. Nell'applicazione era stato impostato su "kind-kind" per test svolti in precedenza. In questo caso, essendo la gestione degli account affidata ad argo, il comando falliva in quanto non venivano trovati dei file essenziali contenuti nel namespace **argocd**.

Una volta creato l'utente, si può digitare nuovamente il comando per avere una lista completa, e se sono stati eseguiti tutti i passaggi correttamente, si avrà qualcosa di simile alla figura 27. Ora, accedendo tramite UI con l'user di

```
PS C:\Users\Mattia\OneDrive\Desktop>  kubectl config set-context --current --namespace argocd
Context "kind-kind" modified.
PS C:\Users\Mattia\OneDrive\Desktop> argocd account list
NAME   ENABLED  CAPABILITIES
admin  true     login
prova  true     apiKey, login
PS C:\Users\Mattia\OneDrive\Desktop> argocd account get --account prova
Name:               prova
Enabled:            true
Capabilities:       apiKey, login

Tokens:
ID                                   ISSUED AT                  EXPIRING AT
b962e1db-f695-4d19-ac84-04eeba7e33b4  2024-01-27T12:58:28+01:00  never
PS C:\Users\Mattia\OneDrive\Desktop>
```

Figura 27: Visualizzazione utenti

prova, è possibile notare come sia presente la stessa applicazione dell'admin, ma con l'accesso in sola lettura. Il risultato sarà dunque quello mostrato in figura 28

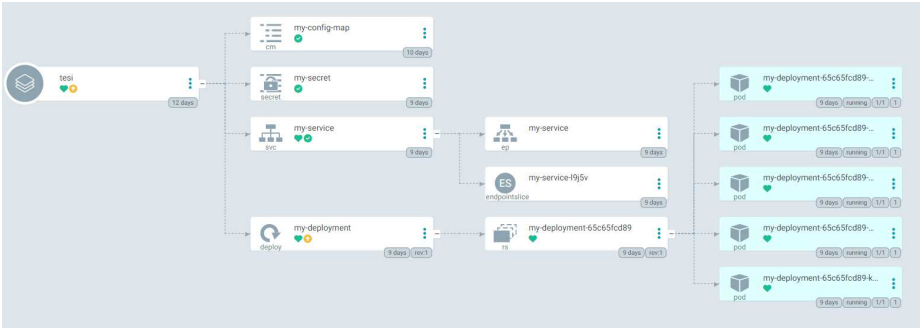

Figura 28: Cluster

5 Analisi della vulnerabilità

In questo ultimo capitolo verrà replicata e analizzata la vulnerabilità in questione. Nei precedenti capitoli è stato implementato tutto il necessario per verificare

quanto descritto nel capito 2. Non resta altro che constatare. È possibile farlo in 3 modi diversi:

1. Da interfaccia grafica (UI);
2. Da linea di comando (CLI);
3. Attraverso chiamate API;

5.1 UI

Effettuando l'accesso all'interfaccia grafica di argo con l'utente **prova** e andando nella sezione "cluster" si potrà vedere qualcosa di simile alla figura 29. Come

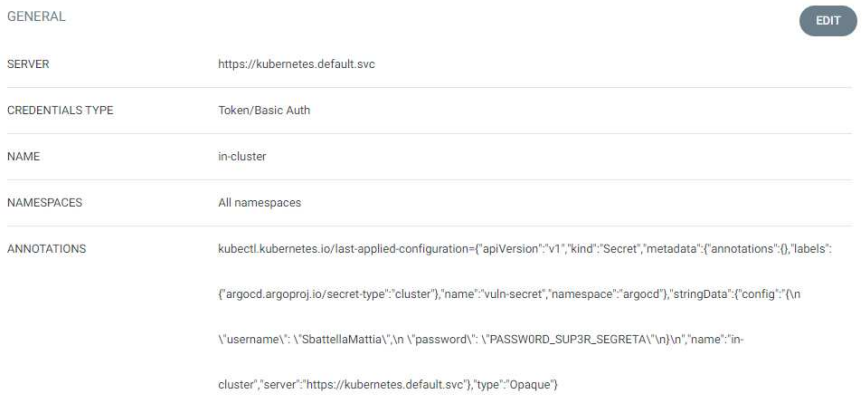

Figura 29: Vulnerabilità

mostrato, non ci sono informazioni sensibili direttamente nelle informazioni principali, ma nella sezione **"Annotation"** che riporta tutti i dati sensibili, salvati nel campo subito dopo aver fatto il comando apply.

5.2 CLI

Una volta effettuato il login tramice CLI (come affrontato nel capitolo 4.7), da terminale basterà digitare il comando:

```
$argocd cluster get in-cluster
```

Il risultato sarà simile a quanto mostrato in figura 30.

5.3 API

Per effettuare le chiamate API corrette è consigliabile utilizzare la documentazione ufficiale delle API di argo, incorporata direttamente con l'interfaccia UI.

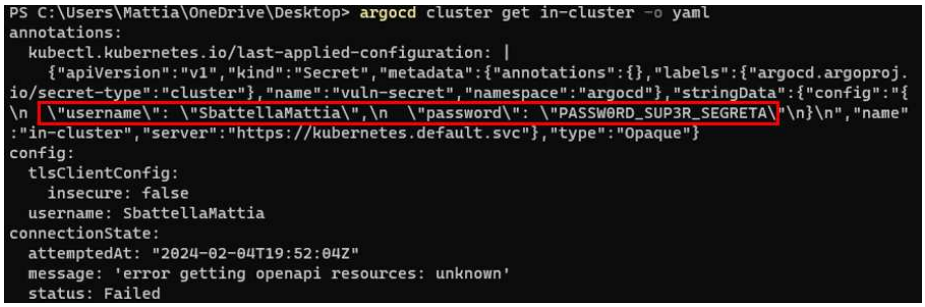

Figura 30: Vulnerabilità

Se infatti si digita nell'URL il path *localhost:8080/swagger-ui* è possibile navigare tra tutte le chiamate API disponibili. Nella sezione *ClusterService* si deve selezionare la chiamata GET: *Return a cluster by service address.*

Per effettuare delle chiamate API occorre essere autorizzati ed avere un token valido. Come prima cosa è utile usare il comando **export** per creare una variabile d'ambiente denominata ARGO_SERVER con il valore *argocd-server.argocd.svc* in modo da poterla riutilizzare :

```
$ export ARGOCD_SERVER=https://argocd-server.argocd.svc
```

Successivamente per generare il token si deve digitare il comando:

```
$ curl $ARGOCD_SERVER/api/v1/session
  -d $'{"username":"prova","password":"prova000"}'
```

in output si avrà così il token che si consiglia di salvare come fatto in precedenza:

```
$ export ARGOCD_TOKEN=<token>
```

Una volta effettuati questi passaggi basterà effettuare:

```
$ curl --insecure "$ARGOCD_SERVER/api/v1/clusters/incluster?
  id.type=name" -H "Authorization: Bearer $ARGO_TOKEN"
```

In output si avrà una schermata simile alla figura 30.

5.4 Analisi

La vulnerabilità è rappresentata dal fatto che nell'oggetto **Cluster Secret** di ArgoCd vengono salvate informazioni importanti come, ad esempio, nel caso in considerazione *username* e *password* del cluster Kubernetes. Attraverso il comando **apply** del file creato, vengono sovrascritte nella sezione **annotation** le informazioni contenute nel campo **last-applied-configuration**, immettendo la configurazione precedente del file in chiaro (ed esponendo quindi la PASSW0RD_SUP3R_SEGRETA). Questo permette ad un utente con accessi in sola

lettura di diventare a tutti gli effetti utente privilegiato di Kubernetes, potendo fare di conseguenza una **privilege escalation** (sfruttamento di vulnerabilità all'interno di un sistema per riuscire ad avere il controllo di risorse normalmente non accessibili). L'elevata pericolosità è dovuta proprio al fatto che qualsiasi utente possa riuscire ad ottenere i privilegi da admin ed accedere a dati sensibili dell'intero cluster Kubernetes. Di conseguenza potrà accedere anche a ciò che è contenuto nei Secrets, potendoli vedere senza alcun problema essendo spesso crittografati solo in base64.

6 Conclusioni

6.1 Impressioni personali

Durante l'inizio della mia ricerca, la mia conoscenza di ArgoCD (come dell'intero ecosistema di Kubernetes) era limitata. Tuttavia, man mano che approfondivo gli aspetti tecnici e le varie componenti, ho guadagnato una comprensione più dettagliata della complessità di questo strumento e del suo ruolo critico nelle implementazioni basate su Kubernetes. Personalmente, seppur la chiave di volta di questo studio sia incentrata sulla vulnerabilità, ho considerato il progetto anche un'esperienza unica per avere a che fare con strumenti all'avanguardia ed apprendere abilità spendibili in futuro.

Il processo di identificazione e analisi della vulnerabilità è stato impegnativo, ma allo stesso tempo gratificante. Le sfide incontrate, hanno contribuito al mio sviluppo professionale. Seppur sembri molto semplice e banale riuscire a replicare la vulnerabilità proposta (in quanto bastava controllare il campo delle annotazioni), quando ci si imbatte in qualcosa di nuovo emergono sempre degli inconvenienti che impiegano tempo per essere capiti e risolti (rilevandosi per la maggior parte delle volte errori banali).

La vulnerabilità individuata ha chiaramente implicazioni pratiche sulla sicurezza delle implementazioni di ArgoCD. Seppur nella descrizione del NIST sia stato espressamente detto che:

> *Note: In many cases, cluster secrets **do not** contain any actually-secret information. But sometimes, as in bearer-token auth, the contents **might** be very sensitive.*

in realtà , nella maggior parte dei casi (se non sempre), le informazioni contenute nei Secret sono estremamente sensibili e potrebbero generare ingenti danni se dovessero trapelare. Questo solleva domande importanti sulle best practice per garantire una gestione sicura delle risorse e sull'importanza di aggiornamenti regolari per affrontare potenziali falle. La divulgazione della vulnerabilità è stata essenziale come campo di studio e, anche grazie alle numerose community attive, è stata immediatamente corretta.

I risultati ottenuti rispecchiano le aspettative iniziali, evidenziando l'importanza di affrontare attivamente la sicurezza non solo in Argo, ma in ogni campo informatico. Precedentemente ho discusso dei numerosi problemi affrontati, che alla fine si sono rivelati spesso delle banalità. Mi piacerebbe comparare proprio questo dettaglio alla vulnerabilità stessa, in quanto risulta avere un pizzico di ironia il fatto che una banalità del genere possa compromettere a tal punto un sistema così rilevante.

6.2 Ringraziamenti

Vorrei prendere un momento per ringraziare coloro che sono stati al mio fianco e che hanno offerto supporto durante tutto il mio percorso universitario.
(So che state leggendo unicamente questa sezione e mi dispiace, ma non sono bravo con le parole).

- Ai miei **Genitori**. Non hanno mai avuto nulla dalla vita per dare comunque tutto a me, nonostante le avversità che hanno dovuto affrontare.
  Vorrei ringraziare mio **Padre** per avermi dato l'opportunità di studiare non pensando alla parte economica. Avendo cominciato a lavorare molto presto, sa bene cosa voglia dire ed è per questo che mi sprona a studiare e continuare il mio percorso. Quello per cui vorrei veramente ringraziarlo però, è per avermi trasmesso il senso del dovere, la curiosità, quella cattiveria e testardaggine che ti costringono a risolvere quel problema specifico, a riprovare quell'esame fino allo sfinimento, se necessario, per riuscire a superarlo. Grazie. Mentre mio padre è il braccio, mia **Madre** è la mente, a Lei devo tutto. Mi ha cresciuto, insegnato ed educato. È stata presente in ogni singolo istante della mia vita e so che sarà così per sempre. Quello per cui vorrei ringraziarla maggiormente è per avermi insegnato ad essere razionale, ma al contempo non trascurare la parte emotiva. Entrambi mi hanno appoggiato nel mio percorso universitario, spronandomi senza alcun tipo di pressione. Vorrei infine ringraziare mia **Zia**. Farebbe di tutto per me, per vedermi felice. Da lei ho imparato a mettere sempre prima gli altri, credendo che prima o poi torni tutto indietro. Un grazie anche a tutti i miei familiari non citati in questa tesi. Siete tutto per me.
- Ai miei **Nonni**, venuti a mancare durante questi tre anni universitari. Ricordo nei loro occhi luccicanti il dispiacere mescolato alla fierezza di chi vede un nipote andare via per studiare. Nella loro vita hanno avuto la necessità di crescere nella fame e nel lavoro, diventando umili contadini, riuscendo dal nulla a costruire una casa, ma soprattutto una famiglia con dei valori importanti ai quali non rinuncerò per nulla al mondo. Spesso guardarsi indietro per rendersi conto della strada fatta è essenziale per poter andare avanti nel modo corretto. Facendolo però, mi viene spontaneo alzare gli occhi al cielo e chiedere: "Avrei la forza di fare la metà di ciò che siete riusciti a fare partendo da zero?". Da Voi ho imparato che un fiore, a volte, riesce a nascere anche nel cemento. Grazie.
- A mio **Fratello**. Non ti ho mai conosciuto, eppure ci penso spesso. Durante questo traguardo è come se fossi con me. Pensare a quello che sei stato costretto a rinunciare, pensare che quello che sto facendo dovesse essere semplicemente il tuo percorso. Non è stato così. Per questo ho il dovere di farcela io, per te, con te. Passo dopo passo, insieme, grazie per avermi concesso l'opportunità di vivere la tua vita e grazie di accompagnarmi nella mia.

- A mio fratello **Michele De Carolis**. Non lo siamo veramente, ma è come se lo fossimo. Il mio fan numero uno, mi ha sostenuto non solo nello studio, ma durante i periodi più brutti. Ti voglio bene.

- Un doveroso ringraziamento al Professore **Spalazzi Luca**, relatore di questa tesi, sempre disponibile, mi ha aperto le porte alla cybersecurity, guidato nel percorso di tirocinio ed è stato il docente del corso di Sistemi Operativi. Spero vivamente che le nostre strade si possano ricollegare durante la magistrale.

- Un ringraziamento (e una birra) a **Leonardo Taccari**. Mi ha guidato sia durante la cyberchallenge che durante la tesi. Vorrei sottolineare la sua disponibilità, la bontà d'animo e la genuinità della persona, sempre disponibile per offrire una mano nei momenti di difficoltà anche su argomenti in cui non era ferrato e unicamente per il piacere di farlo. Una persona fantastica.

- A **Valerio Morelli**, **Mihail Bobeica**, **Sara Sumcutean**, **Luca Renzi**, il loro impegno e sostegno in ambito universitario sono stati fondamentali, mi hanno aiutato a crescere come persona e come studente, avendo svolto insieme numerosi progetti e passato insieme ore di infinito studio.

- A voi, **Nicolò Achilli**, **Claudio Vesprini**, **Roberto Sarachetti**. Amici da una vita e spero per la vita. Abbiamo condiviso molto se non tutto, dalle elementari fino al primo anno di università, sempre insieme. Nonostante le nostre strade si siano divise per motivi di studio, sono convinto che il legame che esiste tra di noi rimanga qualcosa di speciale ed eterno. Grazie ragazzi.

- Al mio cucciolone, **Michele Iobbi**. Amo la tua voglia di non rimanere con le mani in mano, mi ha dato spesso la carica per continuare a lavorare. So che quando non mi scriverà nessuno arriverà il tuo meme nei miei dm, so che quando starò per fare una cazzata tenterai di fermarmi, so che ci sei sempre stato per me e che sempre ci sarai, grazie.

- A **Ludovica Ortolani** (~) che mi ha supportato e sopportato in questi anni, con la quale ho condiviso molto. Sono felice di aver trovato una grande grande amica.

- Ai miei ex coinquilini, **Riccardo Piersanti** e **Jacopo Ciotti**. Vi ringrazio per aver condiviso molto con me durante la nostra convivenza ed aver reso le giornate da fuori sede meno noiose e monotone.

- **A tutti i miei amici**, che non ho potuto citare, ma a cui tengo moltissimo. A tutti quanti, che siano presenti o meno in questo giorno speciale per me, siete come una seconda famiglia. Grazie mille a tutti di cuore. (Al mio gruppo di Monte Urano, a Ciaovona x Scattola).

- Per ultimo, con l'ego da futuro ingegnere, a *Me*, a ciò che ho passato per arrivare fin qui. Abbiamo tutti la premura di fare o ricevere tutto e subito. "Come riesco a farlo in modo che sia facile e veloce?". Bhe, non sempre tutto è facile e veloce, e d'altronde perché dovrebbe esserlo. Se qualcosa fosse facile e veloce da ottenere, finiremmo per averla tutti, e di conseguenza, perderebbe tutto il suo valore, a nessuno importerebbe più nulla di averla. È la fatica, il sudore, l'impegno, la curiosità, la voglia di apprendere e crescere, che rende speciale un percorso. Percorso che risulta essere sempre in salita poiché la ricerca della perfezione è un processo, non un traguardo. Il tutto sta nel festeggiare i piccoli traguardi, guardando sempre avanti.

La sofferenza è necessaria per la crescita.

Riferimenti bibliografici

[1] Argo CD Documentation. URL: https://argo-cd.readthedocs.io/en/stable/.

[2] Brendan Burns, Joe Beda e Kelsey Hightower. Kubernetes: Up & Running. O'Reilly Media, 2017.

[3] Definition of the weakness. URL: https://cwe.mitre.org/data/definitions/532.html.

[4] Docker Documentation. URL: https://docs.docker.com/.

[5] GitOps Documentation (GitLab). URL: https://about.gitlab.com/topics/gitops/.

[6] GitOps repository. URL: https://gitlab.com/sbattella-mattia/Tesi.git.

[7] ITGovernance.eu. URL: https://www.itgovernance.eu/it-it.

[8] KIND Documentation. URL: https://kind.sigs.k8s.io/docs/.

[9] Kubernetes Documentation. URL: https://kubernetes.io/docs/.

[10] Kubernetes icons. URL: https://github.com/kubernetes/community/tree/master/icons.

[11] Michele Ferracin, Architettura di un cluster Kubernetes per principianti. URL: https://www.youtube.com/watch?v=WdEBQt3bqBM.

[12] NIST. URL: https://www.nist.gov/.

[13] Repository della vulnerabilità. URL: https://github.com/argoproj/argo-cd/security/advisories/GHSA-fwr2-64vr-xv9m.

[14] TechWorld with Nana, ArgoCd tutorial for beginners. URL: https://www.youtube.com/watch?v=MeU5_k9ssrs&t=2147s.

[15] TechWorld with Nana, Kubernetes Crash Course for Beginners. URL: https://www.youtube.com/watch?v=s_o8dwzRlu4&t=1008s.

[16] Wikipedia. URL: https://it.wikipedia.org/.

[17] Wikipedia Commons. URL: https://commons.wikimedia.org/.