BEST PRACTICES, SICUREZZA E OTTIMIZZAZIONI

Mauro Patano (INFN Bari)
mauro.patano@ba.infn.it

27 novembre 2025

Outline

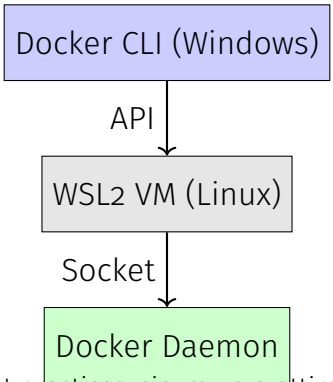
Sicurezza del Sistema Operativo Host

Docker su Windows/macOS?

- I container richiedono funzionalità del kernel Linux (cgroups, namespaces).
- Windows e macOS non hanno queste feature native.
- Soluzione: creare una VM Linux leggera per eseguire il daemon Docker.

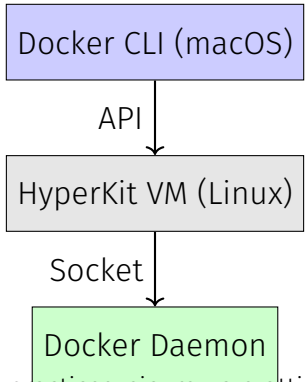
Docker su Windows

- Backend principale: **WSL 2** (Windows Subsystem for Linux).
- WSL 2 usa una VM Hyper-V con kernel Linux ottimizzato.
- Versioni precedenti: VM **MobyLinuxVM** su Hyper-V.



Docker su macOS

- Usa **HyperKit**, un hypervisor leggero basato su macOS.
- Crea una VM Linux minimal (MobyLinuxVM).
- Il daemon Docker gira dentro questa VM.



Flusso di comunicazione CLI → VM → Daemon

- Il client Docker invia comandi al daemon tramite API REST.
- La VM Linux ospita il daemon e il socket `/var/run/docker.sock`.
- Tutte le operazioni passano attraverso la VM.

Proteggere il Sistema Operativo Host

- Se gli attaccanti compromettono il sistema operativo (OS) host, possono compromettere tutti i processi sull'OS, incluso il runtime dei container.
- Il sistema operativo di base dovrebbe essere progettato per eseguire solo il motore dei container, senza altri processi che potrebbero essere compromessi.

Scelta del Sistema Operativo

BEST PRACTICE: OS specifico per container. Tipicamente include di default:

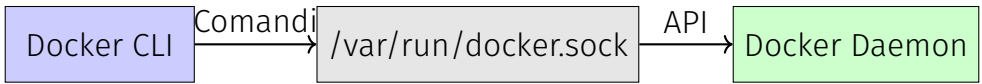
- Funzionalità di sicurezza
- Aggiornamenti automatici
- Hardening dell'immagine

SCELTA PIÙ POPOLARE: distribuzione Linux (OS general-purpose)

- Necessità di gestire ogni funzionalità di sicurezza indipendentemente
- Presenza di servizi di sistema non necessari o applicazioni non containerizzate
- Necessità di scansionare e monitorare costantemente l'OS host per vulnerabilità

Proteggere il socket Docker

- Il socket `/var/run/docker.sock` è un socket Unix, punto di comunicazione tra Docker CLI e Docker Daemon.
- Chi ha accesso al socket può eseguire comandi Docker con privilegi root.
- Rischio di escalation: montaggio file di sistema, container privilegiati.
- Accesso al socket = controllo totale sul sistema.



Best Practice per la protezione

1. Limitare i permessi del socket (solo gruppo docker).
2. Non esporre il socket via rete senza autenticazione.
3. Usare TLS e certificati per accesso remoto.
4. Monitorare accessi con auditd.
5. Evitare di montare il socket in container non fidati.

Verifica permessi del socket

Verifica versione e info Docker

```
ls -l /var/run/docker.sock
# Output tipico:
# srw-rw---- 1 root docker 0 Nov 27 10:00
 ↪  /var/run/docker.sock
```

- Il socket deve essere di proprietà di root e del gruppo docker.
- Solo utenti nel gruppo docker possono accedere al socket.
- Evitare permessi globali (es. `srw-rw-rw-`).

Passo 1: Creare la Regola di Watch

La regola deve monitorare il file per gli accessi (`a`) e le scritture/modifiche (`w`).

- **File da Monitorare:** `/var/run/docker.sock`
- **Permessi:** `wa` (Write e Access/Read)
- **Chiave di Ricerca:** `docker-socket-access`

Regola da Aggiungere a /etc/audit/rules.d/docker-sock.rules

```
sudo auditctl -w /var/run/docker.sock -p wa -k
docker-socket-access
```

Passo 2: Rendere la Regola Attiva e Persistente

La regola deve essere caricata nel kernel e resa persistente al riavvio del sistema.

1. **Caricamento Immediato (Testing):**

Comando `auditctl`

```
sudo auditctl -w /var/run/docker.sock -p wa -k
docker-socket-access
```

2. **Riavvio per Persistenza (Consigliato):**

Riavvio del Servizio

```
sudo systemctl restart auditd
```

Passo 3: Verificare lo Stato e Interrogare i Log

Utilizzare gli strumenti di auditd per confermare che la regola sia attiva e per cercare gli eventi registrati.

- **Verifica Regola Caricata:**

Lista Regole Attive

```
sudo auditctl -l | grep docker-sock
```

- **Ricerca Eventi nei Log:** Usare la chiave definita (`-k`) per filtrare gli eventi.

Ricerca con `ausearch`

```
sudo ausearch -k docker-socket-access
```

Generazione certificati

```
mkdir -p /etc/docker/certs
cd /etc/docker/certs

# CA
openssl genrsa -aes256 -out ca-key.pem 4096
openssl req -new -x509 -days 365 -key ca-key.pem
 ↪  -sha256 -out ca.pem

# Server
openssl genrsa -out server-key.pem 4096
openssl req -subj "/CN=server" -new -key server-key.pem
 ↪  -out server.csr
openssl x509 -req -days 365 -sha256 -in server.csr -CA
 ↪  ca.pem -CAkey ca-key.pem -CAcreateserial -out
 ↪  server-cert.pem
```

Passo 1b: Generazione certificati

Generazione certificati

```
# Client
openssl genrsa -out key.pem 4096
openssl req -subj "/CN=client" -new -key key.pem -out
↪  client.csr
openssl x509 -req -days 365 -sha256 -in client.csr -CA
↪  ca.pem -CAkey ca-key.pem -CAcreateserial -out
↪  cert.pem
```

Passo 2: Avvio del daemon con TLS

- Avviare Docker daemon con verifica TLS e certificati.

Avvio daemon con TLS

```
dockerd --tlsverify \
  --tlscacert=/etc/docker/certs/ca.pem \
  --tlscert=/etc/docker/certs/server-cert.pem \
  --tlskey=/etc/docker/certs/server-key.pem \
  -H=0.0.0.0:2376
```

Passo 3: Configurazione lato client

Configurazione lato client

```
export DOCKER_HOST=tcp://<IP_SERVER>:2376
export DOCKER_TLS_VERIFY=1
export DOCKER_CERT_PATH=~/.docker

docker --tlsverify \
  --tlscacert=~/.docker/ca.pem \
  --tlscert=~/.docker/cert.pem \
  --tlskey=~/.docker/key.pem \
  -H=tcp://<IP_SERVER>:2376 info
```

Alternative sicure

- Usare proxy API con controlli (es. docker-socket-proxy).
- Implementare RBAC per limitare comandi.
- Contenitori rootless per ridurre privilegi.

Sicurezza delle Immagini Docker

Ridurre la Superficie d'Attacco

Mantenere le immagini minimali.
Per ridurre la superficie d'attacco:

- Evitare di includere pacchetti non necessari
  - Più componenti includi in un container, più il sistema sarà esposto e più difficile sarà la manutenzione
- Esposizione delle porte
  - Ogni porta aperta nel container è una porta aperta al sistema
  - Esporre solo le porte necessarie all'applicazione ed evitare porte come SSH (22)

Usare Immagini Ufficiali e Affidabili

- Scegliere attentamente la base per le immagini (istruzione FROM):
  - Costruire su immagini non affidabili o non mantenute erediterà tutti i problemi e le vulnerabilità
- Invece di prendere un'immagine OS di base e installare i pacchetti necessari, usare l'immagine ufficiale (es. node) per l'applicazione, se disponibile

Miglioramenti:

- Dockerfile più pulito
- Immagine ufficiale e verificata, già costruita con le best practices

Usare Immagini di Piccole Dimensioni

Scegliendo un'immagine basata su una distribuzione OS completa:

- Avrai molti strumenti già impacchettati
- La dimensione dell'immagine sarà maggiore, ma potresti non aver bisogno della maggior parte di questi strumenti
- Devi considerare l'aspetto della sicurezza

Scegliendo immagini più piccole:

- Serve meno spazio di archiviazione e trasferimento più veloce
- Minimizzi la superficie d'attacco e costruisci immagini più sicure
- Le immagini *distroless* contengono solo il set minimo di librerie per eseguire Go, Python o altri framework

Non Installare Pacchetti Non Necessari

Non installare pacchetti fuori dall'ambito e dallo scopo del container.
Riduce:

- Dimensione del container
- Superficie d'attacco
- Complessità
- Dipendenze
- Dimensione dei file
- Tempi di build

Utilizzare Build Multi-Stage

- Ci sono contenuti nel progetto necessari per costruire l'immagine ma non per eseguire l'applicazione
- Le build multi-stage separano la fase di build dalla fase di runtime
- Solo lo stage finale è usato per creare l'immagine

Miglioramenti:

- Separazione degli strumenti di build e dipendenze da ciò che serve per il runtime
- Meno dipendenze e riduzione della dimensione finale dell'immagine

Usare l'Utente con Minori Privilegi

- Di default, Docker esegue i processi del container come root all'interno del container
- Questo introduce un problema di sicurezza: il container avrà potenzialmente accesso root sull'host Docker
- Facilita l'escalation dei privilegi per un attaccante

Best practice:

- Creare un utente dedicato e un gruppo dedicato nell'immagine Docker
- Eseguire l'applicazione dentro il container con quell'utente (direttiva USER)

Nota: il daemon Docker e il container stesso continuano a girare con privilegi root

Eseguibili Appartenenti a Root e Non Scrivibili

- Ogni eseguibile in un container deve essere di proprietà dell'utente root (anche se eseguito da un utente non-root) e non deve essere scrivibile da tutti
- L'utente necessita solo dei permessi di esecuzione sul file, non la proprietà
- Questo blocca l'utente in esecuzione dal modificare binari o script esistenti
- È una pratica per garantire l'immutabilità del container

```
WORKDIR $APP_HOME
COPY app-files/ /app
RUN chmod 555 /app/my-app-entrypoint.sh
USER app
ENTRYPOINT /app/my-app-entrypoint.sh
```

Usare il File .dockerignore

- Di solito quando costruiamo l'immagine, non abbiamo bisogno di tutto ciò che abbiamo nel progetto per eseguire l'applicazione
- Non servono cartelle auto-generate, come target o build, non serve il file readme, ecc.
- Usare un file `.dockerignore` dove elencare tutti i file e cartelle che vogliamo ignorare durante la build dell'immagine

Miglioramenti:

- Riduzione della dimensione dell'immagine

Best Practices per Immagini Docker

Usare Versioni Specifiche delle Immagini Docke

Usando il tag `latest`:

- Potresti ottenere una versione dell'immagine diversa rispetto alla build precedente
- La nuova versione dell'immagine potrebbe causare problemi
- È imprevedibile, causando comportamenti inattesi

Distribuisci la tua applicazione con una versione specifica!
La regola è: più specifico è meglio
Miglioramenti:

- Trasparenza per sapere esattamente quale versione dell'immagine base stai usando

Aggiungere l'Istruzione HEALTHCHECK

- La direttiva di istruzione HEALTHCHECK dice a Docker come determinare se lo stato del container è normale
- Aggiungere questa istruzione ai Dockerfile permette a Docker di uscire da un container non funzionante e istanziarne uno nuovo

```
HEALTHCHECK --interval=30s --timeout=3s \
  CMD curl -f http://localhost/ || exit 1
```

Ottimizzare la Cache per i Layer dell'Immagine

Un'immagine è costruita basandosi su un Dockerfile:

- Le parole chiave RUN, COPY e ADD creano layer
- Ogni layer contiene le differenze dal layer precedente
- I layer aumentano la dimensione dell'immagine finale

Best practice:

- Combinare comandi correlati
- Ridurre il numero di layer
- Rimuovere file non necessari nello stesso step RUN che li ha creati
- Minimizzare il numero di volte che si esegue `apt-get upgrade`

Ottimizzare la Cache - Ordinamento Comandi

- Ogni layer viene messo in cache da Docker
- Quando un layer cambia, tutti i layer successivi devono essere ricreati

Best practice: Ordinare i comandi dal meno al più frequentemente modificato

```
FROM python:3.9-slim
WORKDIR /app
COPY sample.py .
COPY requirements.txt .
RUN pip install -r \
    /requirements.txt
```

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r \
    /requirements.txt
COPY sample.py .
```

Ottimizzare la cache

- Combinare sempre `apt-get update` con `apt-get install` nella stessa istruzione RUN
- Pulire la cache apt rimuovendo `/var/lib/apt/lists` riduce la dimensione dell'immagine

```
RUN apt-get update && \
    apt-get install -y \
      build-essential \
      curl \
      git && \
    rm -rf /var/lib/apt/lists/*
```

Evitare di Memorizzare Segreti nelle Immagini

- I segreti sono informazioni sensibili: password, credenziali database, chiavi SSH, token, certificati TLS
- Non devono essere incorporati nelle immagini senza cifratura
- Non inserire mai segreti o credenziali nelle istruzioni Dockerfile

Alternative:

- Variabili d'ambiente (a runtime)
- Argomenti di build (a build-time)
- Strumenti di orchestrazione (Docker secrets, Kubernetes secrets)
- Volumi condivisi cifrati

Preferire COPY rispetto ad ADD

Entrambi i comandi permettono di copiare file da una posizione specifica in un'immagine Docker:

- **COPY**: usato per copiare file o directory locali dall'host Docker all'immagine
- **ADD**: può essere usato anche per:
  - Scaricare file esterni
  - Decomprimere automaticamente file compressi

Usa COPY a meno che non ti serva realmente la funzionalità di ADD
COPY è più prevedibile e meno soggetto a errori

Non Vincolare a un UID Specifico

- OpenShift, di default, userà UID casuali quando esegue container
- Forzare un UID specifico (es. primo utente standard con UID 1000) richiede l'aggiustamento dei permessi di qualsiasi bind mount
- Se esegui il container con l'UID dell'host (opzione `-u` in docker), potrebbe interrompere il servizio quando tenta di leggere o scrivere da cartelle dentro il container

Approccio Continuo

Configurare gli strumenti per analizzare le immagini:

- Git push $\rightarrow$ Docker build $\rightarrow$ Test/Scan $\rightarrow$ Deploy in produzione

Configurare gli strumenti per analizzare periodicamente le immagini:

- Nuove vulnerabilità di sicurezza vengono scoperte
  - docker scan
  - Altri strumenti
- L'immagine base è stata ricostruita (con le ultime patch di sicurezza)
- Ricostruire la propria immagine

Usare Tag di Immagine Statici in Produzione

- Seguire una politica di tagging coerente e consistente
- Documentare la politica di tagging affinché gli utenti dell'immagine possano comprenderla facilmente
- Le immagini container sono un modo di impacchettare e rilasciare software
- Taggare l'immagine permette agli utenti di identificare una versione specifica del software
- Collegare strettamente il sistema di tagging sulle immagini container alla politica di rilascio del software

Esempi:

- Includere un numero di versione seguendo il versionamento semantico nei tag
- Usare l'hash SHA del commit git come tag per il codice

Sicurezza del Runtime dei Container Docker

Best Practices per il Runtime

1. Non usare container privilegiati
2. Non esporre porte non utilizzate
3. Non condividere il namespace di rete dell'host
   - Usare `–net=host` solo quando strettamente necessario
4. Aggiungere limiti alle risorse
   - Evitare che il container consumi tutta la memoria o CPU
5. Impostare la politica di restart del container su on-failure
   - `–restart on-failure[:max-retries]`
6. Impostare filesystem e volumi in sola lettura
7. Eseguire Docker in modalità Rootless

Esempio: Filesystem Read-Only

```
docker run --read-only --tmpfs /tmp alpine \
  sh -c 'echo "whatever" > /tmp/file'
```

- Il container ha il filesystem principale in sola lettura
- `tmpfs` fornisce uno spazio scrivibile temporaneo in memoria
- Previene modifiche permanenti al filesystem del container

Health Check nei Container Docker

Come Aggiungere un Health Check al Container

Gli health check (disponibili da Docker 1.12) permettono a un container di esporre la disponibilità del proprio carico di lavoro.

- Questo è diverso dal semplice verificare se il container è in esecuzione
- Se il database si blocca, l'API server non può gestire richieste, anche se il suo container Docker è ancora in esecuzione
- Quando viene specificato un comando di health check, Docker sa come testare se il container funziona correttamente
- Senza health check, Docker non ha modo di sapere se i servizi nel container funzionano o meno

Configurazione degli Health Check

Gli health check possono essere configurati in diversi modi:

- Direttamente nell'immagine Docker (Dockerfile)
- Quando si lancia un container standalone con `docker run`
- Nel file docker-compose

In tutti i casi, l'health check è un comando che il daemon Docker esegue ogni 30 secondi (intervallo predefinito modificabile).
Docker usa il codice di uscita del comando per determinare lo stato di salute:

- **0** – Il container è healthy e funziona normalmente
- **1** – Il container è unhealthy; il carico di lavoro potrebbe non funzionare
- **2** – Codice riservato da Docker, non deve essere usato

Opzioni di Configurazione Health Check

Opzioni disponibili per personalizzare l'health check:

- **interval** (default: 30s) - tempo tra un controllo e l'altro
- **timeout** (default: 30s) - tempo di attesa per una risposta prima di considerare il controllo fallito
- **start-period** (default: 0s) - secondi di attesa prima di iniziare i controlli, per dare tempo al container di avviarsi
- **retries** (default: 3) - numero di fallimenti consecutivi necessari per dichiarare il container unhealthy

Esempio: Health Check per Nginx

Aggiungiamo un health check a un container nginx usando curl:

Avvio container con health check

```
docker run --name nginx --detach \
  --health-cmd='curl --fail http://localhost:80 || exit
  ↪  1' \
  nginx
```

- Il comando curl fa una richiesta a localhost:80
- Se la richiesta restituisce codice HTTP 200, ritorna exit code 0
- Altrimenti ritorna exit code 1

Stati del Container con Health Check

Un container può avere tre stati:

- **starting** – Stato iniziale quando il container si sta ancora avviando
- **healthy** – Se il comando ha successo, il container è healthy
- **unhealthy** – Se il comando supera il timeout o fallisce per il numero specificato di retry

Visualizzazione con `docker ps`:

```
CONTAINER ID   IMAGE     COMMAND
 ↪  CREATED         STATUS
75beb1db3d27   nginx     "/docker-entrypoint...."   9
 ↪  seconds ago   Up 3 seconds (health: starting)
```

Verifica dei Log Health Check

Visualizza log

```
docker logs nginx
127.0.0.1 - - [06/Sep/2023:06:44:49 +0000] "GET /
↪  HTTP/1.1" 200 615 "-" "curl/7.88.1" "-"
127.0.0.1 - - [06/Sep/2023:06:45:20 +0000] "GET /
↪  HTTP/1.1" 200 615 "-" "curl/7.88.1" "-"
127.0.0.1 - - [06/Sep/2023:06:45:51 +0000] "GET /
↪  HTTP/1.1" 200 615 "-" "curl/7.88.1" "-"
```

Il controllo viene eseguito ogni 30 secondi (modificabile con `–health-interval`)

Simulazione di un Fallimento

Rimuoviamo il file index.html per simulare un errore dell'applicazione:

Simula fallimento

```
docker exec nginx sh -c 'mv
 ↪  /usr/share/nginx/html/index.html \
  /usr/share/nginx/html/index.html.1'
    docker logs nginx
2023/09/06 06:49:27 [error] 29#29: *10 directory index
 ↪  of "/usr/share/nginx/html/"
  is forbidden, client: 127.0.0.1
127.0.0.1 - - [06/Sep/2023:06:49:27 +0000] "GET /
 ↪  HTTP/1.1" 403 153 "-" "curl/7.88.1" "-"
```

Dopo 3 fallimenti consecutivi (default):

Esercizio

Ripristinare il file index.html e verificare lo stato del container

Ripristino

```
docker exec nginx sh -c 'mv
 ↪  /usr/share/nginx/html/index.html.1 \
  /usr/share/nginx/html/index.html'
```

Verifica stato

```
docker ps
```

Dopo alcuni controlli, il container dovrebbe tornare nello stato `healthy`.

Esempio: MariaDB con Health Check Personaliz

Creare un container MariaDB con health check personalizzato:
Passo 1: Impostare le variabili d'ambiente:

Variabili d'ambiente

```
export MYSQL_USER=myuser
export MYSQL_PASSWORD=mypassword
export MYSQL_ROOT_PASSWORD=rootpassword
export MYSQL_DATABASE=mydatabase
```

Esempio MariaDB: Avvio Container

Passo 2: Eseguire il container con health check personalizzato:

Container MariaDB con health check

```
docker run --detach --name db \
  --env MYSQL_DATABASE=${MYSQL_DATABASE} \
  --env MYSQL_USER=${MYSQL_USER} \
  --env MYSQL_PASSWORD=${MYSQL_PASSWORD} \
  --env MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD} \
  --health-cmd='mariadb-admin -p${MYSQL_ROOT_PASSWORD}
  ↪ \
    ping -h localhost' \
  --health-interval=20s \
  --health-retries=3 \
  mariadb:latest
```

Esempio MariaDB: Spiegazione Opzioni

Opzioni utilizzate nel comando:

- `--name db`: Nome del container
- `--env MYSQL_DATABASE`: Nome database come variabile d'ambiente
- `--env MYSQL_USER`: Nome utente MySQL
- `--env MYSQL_PASSWORD`: Password utente MySQL
- `--env MYSQL_ROOT_PASSWORD`: Password utente root MySQL
- `--health-cmd`: Comando personalizzato che usa mariadb-admin per verificare se il server risponde
- `--health-interval=20s`: Intervallo di controllo di 20 secondi
- `--health-retries=3`: Numero di tentativi prima di marcare come unhealthy

Riepilogo Health Check

Vantaggi degli Health Check:

- Monitoraggio automatico dello stato reale dell'applicazione
- Rilevamento di problemi anche quando il container è tecnicamente "running"
- Integrazione con orchestratori (Docker Swarm, Kubernetes)
- Riavvio automatico di container non funzionanti

Best Practices:

- Configurare sempre health check per applicazioni in produzione
- Usare endpoint dedicati per health check nelle applicazioni
- Calibrare correttamente interval, timeout e retries
- Testare gli health check durante lo sviluppo

Apptainer

Cos'è Apptainer?

Apptainer è una piattaforma per container che permette di creare ed eseguire container che impacchettano software in modo portabile e riproducibile.

Caratteristiche principali:

- **Container come singolo file**: facile da trasportare e condividere
- **Portabilità**: costruisci sul laptop, esegui su cluster HPC, cloud o workstation
- **Nessuna preoccupazione per l'OS**: non serve installare software su ogni sistema diverso

Perché usare Apptainer?

Apptainer è stato creato per eseguire applicazioni complesse su cluster HPC in modo semplice, portabile e riproducibile.

Focus di Apptainer:

- **Riproducibilità verificabile e sicurezza**: firme crittografiche, formato immutabile, decrittazione in memoria
- **Integrazione anziché isolamento**: accesso diretto a GPU, reti ad alta velocità, filesystem paralleli
- **Mobilità del calcolo**: formato SIF a file singolo facile da trasportare
- **Modello di sicurezza semplice**: stesso utente dentro e fuori dal container, nessun privilegio aggiuntivo per default

Nato al Lawrence Berkeley National Laboratory, ora usato in ambito accademico e industriale.

Containerization Tech.	Apptainer	Docker
Architettura	Daemonless, single file format	Client-Server with daemon
Sicurezza	Integrazione con il sistema host, senza privilegi root	Richiede privilegi root per alcune features
Tipo di immagine	Immagini in formato Singularity. Compatibile con Docker	Immagini in formato Docker. Supporto per registry esterni
Tipo di utilizzo	Containerizzazione scientifica	Containerizzazione generale
Integrazione con HPC	Ottimizzato per ambienti HPC (High-Performance Computing)	Supporta ambienti cloud e containerizzazione su larga scala
Comunità e Supporto	Comunità focalizzata sulla ricerca scientifica e HPC	Comunità vasta, supporto per DevOps
Facilità d'uso	Comando simile a Docker, ma con alcune differenze in termini di gestione e configurazione	Interfaccia utente ben documentata e molto usata, facile da imparare
Portabilità	Eccellente grazie alla compatibilità con altre piattaforme di containerizzazione	Eccellente, con ampio supporto in ambienti cloud e locali

Tabella: Confronto tra Apptainer e Docker

Supporto per Container Docker e OCI

Il formato container **Open Containers Initiative (OCI)**, nato da Docker, è lo standard dominante per deployment containerizzati nel cloud.

Sebbene il formato container di Apptainer abbia molti vantaggi unici, è probabile che dovrai lavorare con container Docker/OCI.

Apptainer punta alla massima compatibilità con Docker, rispettando i vincoli di un runtime adatto a sistemi condivisi e ambienti HPC.

Cosa puoi fare con Apptainer

Con Apptainer puoi:

- **Pull, eseguire e costruire** dalla maggior parte dei container su Docker Hub, senza modifiche
- **Pull, eseguire e costruire** da container ospitati su altri registry, inclusi registry privati on-premise o cloud
- **Pull e costruire** da container OCI in formati archivio, o cachati in un daemon Docker locale

Nota: Apptainer può anche essere eseguito nested dentro un container Docker, ma richiede opzioni runtime aggiuntive al comando docker.

Container da Docker Hub

Docker Hub è il luogo più comune dove i progetti pubblicano immagini container pubbliche.

Container Pubblici
È facile eseguire un container pubblico di Docker Hub con Apptainer. Basta aggiungere `docker://` davanti al repository e tag del container:

- Apptainer recupera blob e dati di configurazione da Docker Hub
- Estrae i layer che compongono il container Docker
- Crea un file SIF da essi
- Il file SIF viene conservato nella directory cache di Apptainer
- Le successive esecuzioni dello stesso container non richiedono download e conversione

Esempio: Eseguire Container da Docker Hub

Eseguire il container lolcow da Docker Hub

```
$ apptainer run docker://sylabsio/lolcow:latest
 _____________________________
< Mon Oct 4 14:50:30 CDT 2021 >
 -----------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

Pull di Container Docker

Per ottenere il container Docker come file SIF in una posizione specifica, che puoi spostare, condividere e conservare:

```
$ apptainer pull docker://sylabsio/lolcow
INFO:    Using cached SIF image

$ ls -l lolcow_latest.sif
-rwxr-xr-x 1 myuser myuser 74993664 Oct  4 14:55
 ↪  lolcow_latest.sif
```

- **Prima volta**: il container viene scaricato e tradotto
- **Successive volte**: viene copiato dalla cache

Autenticazione / Container Privati

Per utilizzare i limiti API sotto un account Docker Hub, o per accedere a container privati, è necessario autenticarsi a Docker Hub.

Ci sono diversi modi per farlo con Apptainer:

- Comando CLI `apptainer registry`
- Autenticazione tramite Docker CLI

L'autenticazione permette di accedere a container privati e di beneficiare di limiti API più elevati per il download da Docker Hub.

Comando apptainer registry login

Il comando `apptainer registry login` supporta il login a Docker Hub e altri registry OCI.

Per Docker Hub, l'hostname del registry è `docker.io`:

```
$ apptainer registry login --username myuser
 ↪  docker://docker.io
Password / Token:
INFO:    Token stored in
 ↪  /home/myuser/.docker/config.json
INFO:    Token stored in
 ↪  /home/myuser/.apptainer/remote.yaml
```

Importante: Il Password/Token deve essere un **Docker Hub CLI access token**, che puoi generare nella sezione 'Security' del tuo profilo Docker Hub.

Gestione delle Credenziali Registry

Verificare i registry autenticati:

```
$ apptainer registry list
```

Logout da un registry:

```
$ apptainer registry logout docker://docker.io
INFO:    Logout succeeded
```

Per maggiori informazioni su `apptainer registry` e i suoi sottocomandi, incluso il flag `–authfile` per memorizzare e usare credenziali in file specificati dall'utente, consulta la documentazione del comando registry.

Autenticazione tramite Docker CLI

Se hai il CLI `docker` installato sulla tua macchina, puoi usare `docker login` per accedere al tuo account.

Questo memorizza le informazioni di autenticazione in `~/.docker/config.json`.

Il processo che Apptainer usa per recuperare container Docker/OCI tenterà di utilizzare queste informazioni per effettuare il login.

- Metodo conveniente se usi già Docker
- Condivisione automatica delle credenziali tra Docker e Apptainer
- Non richiede configurazione aggiuntiva

Container da Altri Registry

Puoi usare URI `docker://` con Apptainer per scaricare ed eseguire container da registry OCI diversi da Docker Hub.

Come fare:

- Includi l'hostname o l'indirizzo IP del registry nell'URI `docker://`
- L'autenticazione con altri registry funziona allo stesso modo
- A volte è necessario recuperare le credenziali usando strumenti specifici (specialmente con Cloud Provider)

Di seguito alcuni esempi specifici per registry comuni. La maggior parte degli altri registry segue un pattern simile per scaricare immagini pubbliche e autenticarsi per accedere a immagini private.

Quay.io

Quay è un registry OCI usato da molti progetti, ospitato su `https://quay.io`.

Pull di container pubblici:

```
$ apptainer pull docker://quay.io/bitnami/python:3.7
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
...

$ apptainer run python_3.7.sif
Python 3.7.12 (default, Sep 24 2021, 11:48:27)
[GCC 8.3.0] on linux
>>>
```

Quay.io - Repository Privati

Per scaricare container da repository privati:

1. Genera un CLI token nell'interfaccia web di Quay
2. Usa uno dei metodi di autenticazione:

- `apptainer registry login –username myuser docker://quay.io`
- `docker login quay.io` (se docker è installato)
- Flag `–docker-login` per login interattivo una tantum
- Variabili d'ambiente `APPTAINER_DOCKER_USERNAME` e `APPTAINER_DOCKER_PASSWORD`

NVIDIA NGC

Il catalogo NVIDIA NGC (`https://ngc.nvidia.com`) contiene vari software GPU in container.

Molti container sono documentati da NVIDIA come supportati da Apptainer.

Pull senza autenticazione (guest):

```
$ apptainer pull
 ↪  docker://nvcr.io/nvidia/pytorch:21.09-py3
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
```

I container NGC sono ora disponibili come guest senza login.

NVIDIA NGC - Account Privati

Se hai bisogno di accedere con un account NVIDIA (es. NGC Private Registry):

1. Genera una API key nell'interfaccia web
2. Usa username `$oauthtoken` e la tua API key come password

Metodi di autenticazione:

- `apptainer registry login –username \$oauthtoken docker://nvcr.io`
- `docker login nvcr.io` (se docker è installato)
- Flag `–docker-login`
- `APPTAINER_DOCKER_USERNAME="\$oauthtoken"` e `APPTAINER_DOCKER_PASSWORD`

Vedi anche:
`https://docs.nvidia.com/ngc/ngc-private-registry-user-guide/`

GitHub Container Registry (GHCR)

GitHub Container Registry è sempre più usato per fornire container Docker insieme al codice sorgente dei progetti.

Pull di container pubblici:

```
$ apptainer pull
 ↪  docker://ghcr.io/containerd/alpine:latest
INFO:    Converting OCI blobs to SIF format
INFO:    Starting build...
```

Container privati:

- Genera un personal access token nell'interfaccia web di GitHub
- Il token deve avere gli scope richiesti (vedi documentazione GitHub)

GitHub Container Registry - Autenticazione

Usa uno dei seguenti metodi di autenticazione con il tuo username e personal access token:

- `apptainer registry login –username myuser docker://ghcr.io`
- `docker login ghcr.io` (se docker è installato)
- Flag `–docker-login` per login interattivo
- Variabili d'ambiente `APPTAINER_DOCKER_USERNAME` e `APPTAINER_DOCKER_PASSWORD`

Nota: Apptainer può anche fare push di file SIF direttamente su `ghcr.io` usando il protocollo `oras://`. I container condividono lo stesso namespace, ma devono essere scaricati con lo stesso protocollo usato per il push.

AWS ECR (Elastic Container Registry)

Lavorare con AWS ECR generalmente richiede autenticazione.

Metodo più diretto - `get-login-password`:

```
$ aws ecr get-login-password --region region
```

Usa l'AWS CLI per richiedere una password. Lo username è sempre **AWS**.

Warning: Il credential helper Docker di ECR non può essere usato, poiché Apptainer attualmente non supporta credential helper esterni usati con Docker, ma legge solo le credenziali memorizzate direttamente nel file `.docker/config.json`.

AWS ECR - Metodi di Login

Dopo aver ottenuto la password, autentica usando uno dei seguenti metodi:

- `apptainer registry login –username AWS docker://<accountid>.dkr.ecr.<region>.amazonaws.com`
- `docker login –username AWS <accountid>.dkr.ecr.<region>.amazonaws.com` (se docker è installato)
- Flag `–docker-login` per login interattivo
- `APPTAINER_DOCKER_USERNAME=AWS` e `APPTAINER_DOCKER_PASSWORD`

Ora puoi scaricare container dal tuo URI ECR:
`docker://<accountid>.dkr.ecr.<region>.amazonaws.com`

Azure ACR (Azure Container Registry)

Un Azure Container Registry (ACR) ospitato su Azure generalmente contiene immagini private e richiede autenticazione per il pull.

Autenticazione per identità:

- Usa `az acr login` dall'Azure CLI
- Aggiunge credenziali a `.docker/config.json` che possono essere lette da Apptainer

Account Service Principle:

- Hanno username e password espliciti
- Usa i metodi di autenticazione standard

Vedi la documentazione ACR per maggiori informazioni sulle opzioni di autenticazione.

Azure ACR - Metodi di Autenticazione

Per account Service Principle, autentica usando uno dei seguenti metodi:

- `apptainer registry login –username myuser docker://myregistry.azurecr.io`
- `docker login –username myuser myregistry.azurecr.io` (se docker è installato)
- Flag `–docker-login` per login interattivo
- Variabili d'ambiente `APPTAINER_DOCKER_USERNAME` e `APPTAINER_DOCKER_PASSWORD`

Repository-scoped access token (preview):

- Usa `az acr token create` per ottenere una coppia nome token/password
- Può essere usata con i metodi di autenticazione sopra