The Ultimate Guide to Secrets Management in Containers

Last Updated: June 2017

Introduction

A secret, per one dictionary definition, is something that is "*kept from the knowledge of any but the initiated or privileged*". Put simply, a secret is something you don't want anyone to know, apart from those who *should* know it or have sufficient privileges to know it. Common examples of secrets are passwords, tokens, and private keys. The definition of a secret is rather simple, but the practice of it (at least in technical terms) is anything but simple.

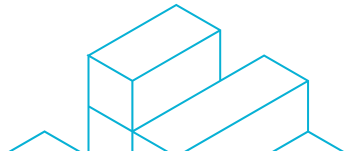
The task of managing and securing secrets is of course not new. There are many solutions that facilitate secure secrets storage, distribution and management. However, these solutions are not tailored for containerized environments. In such environments, the user gives up some control to an orchestrator, in favor of simplicity and efficiency. The orchestrator is responsible for scheduling each task in a virtual container, passing parameters to the container, adding mounts, controlling privileges, scaling the application, passing secrets and more. In such environments, we would like to keep the flexibility and simplicity we get from orchestrators, while not conceding any security when it comes to secrets.

In this guide, we examine the security aspects of several solutions that support secrets in a containerized environment. First we clearly define the security requirements of such solutions, then we deep-dive into each solution and examine its approach. We also give a short word about which type of tools we left out and why.

The Secrets Life Cycle

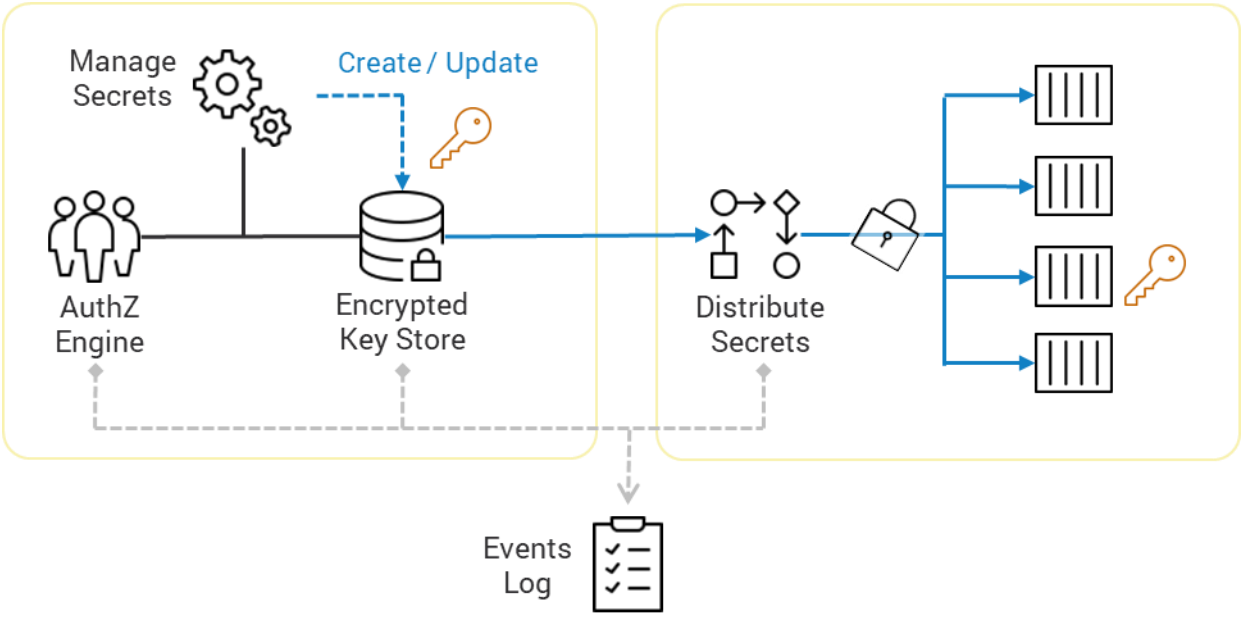
Before defining the security requirements of managing secrets, let's map the essential components and interfaces for a secure secret solution:

1. A secret is created / updated by a user with sufficient privileges. The communication channel between the user and API should be secure and mutually authenticated.
2. The API server is responsible of authenticating and authorizing users and services.
3. Every administrative operation on the secret is logged.
4. A secret may be created, updated or revoked via a backend application. In such case, the user that created the secret is not exposed to the secret. Communication with the backend application should be encrypted and mutually authenticated.
5. The secret is stored in a sealed storage, which can only be decrypted using an unseal key. The unseal key is stored in a secure location, separate from the API server.
   For additional security, the secret itself may also be encrypted for its consuming container.
6. Nodes receive the secret (or encrypted secret) over an encrypted and mutually authenticated channel. The API server should authenticate and authorize nodes.



7. Log each time a secret is transferred, updated or removed from a container
8. Secret is available for reading only to the consuming container. The container can decrypt the secret using its private key for additional security.

The following diagram illustrates which actions can be taken by which component in an ideally secure secret management solution:



Security Requirements

Before we can start examining how orchestrators protect secrets, we need some definitions to serve as a basis for comparison:

Access Control

Administrators should be able to granularly control WHO (users) and WHAT (services) can access secrets, and what type of access they have: read / write / delete / list.

Write Only

Except for the consumer of the secret, no-one should be able to read it. Even the user that created the secret has no real reason to read it.

Rotation

A secret should be rotated on regular basis (e.g. every 90 days). Once a secret value is changed, it should propagate to all containers that are currently using it.

Revocation

In case a secret is exposed, or is no longer needed - it should be possible not only to delete the secret from storage and all containers using it, but also revoke the access that a secret grants to a resource.

Channel Protection

Secrets should only be transmitted over a secured communication channels, that ensure privacy, integrity and the identity of the communicating parties. Usually this is accomplished by using encrypted communication channels, that are also mutually authenticated.

Readable only in Container

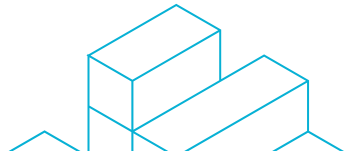
A Secret should only be exposable in the context of its consuming container: only processes running inside the container should have access to the secrets. It should not be *trivial* for processes running outside the container to expose the secret (while it is impossible to fully protect the secret from the underlying host, it should not be “easy”).

Administration Audit

All administrative access and attempted access to Secrets should be logged (creation, revocation, rotation, reading, listing).

Usage Audit

Log when a secret is injected, rotated or removed from a container.



Approaches to Secrets Management

As we mentioned earlier, this guide does not cover the entirety of approaches to managing secrets. We focus on those that we believe are an organic match to container-based environments, and we offer a deep dive into those. However, in this section we also include other approaches, and explain why we think they were not a good match for comparative analysis.

Hard-Coded / Hard-Configured

The naïve approach to secrets is to include them in your source code or configuration files. We hope that the reader of this guide does not need any convincing on why this is a terrible idea. Good, let's move on.

Image / Build Secrets

If it is a bad idea to keep secrets in the code or configuration, why not simply include them in the image? Or better yet, integrate secret generation during the building stage of an image?

The reality is that using secrets inside images is not much different from hard-coding secrets. Their storage is not secure, and access to secrets themselves cannot be administered separately from access to the repository. Additionally, the process of updating or revoking a secret would require building a new image.

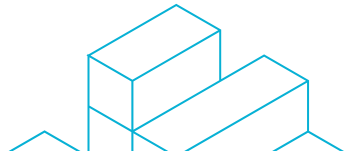
Mounted Files / Environment Variables

One solution to keep secrets out of code, configuration, and the image itself is to inject secrets into containers via mounted files or environment variables. This is a step in the right direction, and is supported in container engines. However, it is not easy to do right: mounted files need to only exist in memory and never be saved to disk, environment variables should not be easy to extract from the host, and the injection process itself shouldn't reveal the secret (for example: when secrets are written in the container engine logs if given as arguments). Additionally, storing the secrets, updating them and securely passing them to the container remains a problem.

Key Management Services (KMS)

There are several enterprise-grade solutions that offer users a method of storing secrets and accessing them securely. Some solutions can be used as standalone, such as Hashicorp Vault and or CyberArk Enterprise Password Vault. Others are integrated offerings from cloud providers, such as those from Amazon, Google Cloud and Azure.

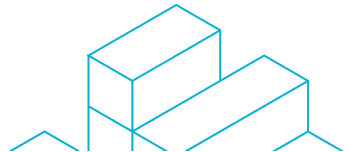
While these solutions address many of the requirements for secure secrets management - such as securing secrets at rest and providing detailed audit - we will not do a comparative analysis on these solutions by themselves. They do not provide a holistic platform for securely distributing



secrets to containers, and require that a container directly interacts with a key store. For example: to access secrets, a container in a cloud platform needs to use an IAM role of its host, or some other user / service account to retrieve secrets. This makes the container identity tightly coupled with the cloud platform, and does not enable a secret to be confined to a single container. Also, changing the KMS requires changes to containers themselves.

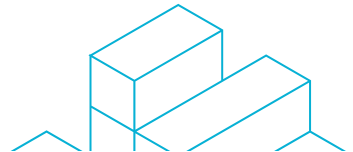
Orchestrators

Most orchestration tools today offer some level of secrets storage and management. These solutions are the primary focus of this paper. However, it is worth noting that orchestrators can create a new problem of secrets management duplication. When an organization decides to move to a containerized environment, it probably already has a KMS for its secrets (outside containers). Now it must either partially duplicate its secrets into the orchestrator, or change its entire secrets infrastructure depending on the orchestrator – unless there's an easy way to integrate the orchestrator secrets features with existing KMSs. It is important to keep this in mind while reading this paper that focuses mainly on security aspects, as some solutions better integrate into existing KMSs.



Solutions Comparison Table

	Kubernetes	Swarm	OpenShift	DC/OS	Nomad	Aqua
Encryption at Rest	No	Only when sealed	No	By default	By default	By default
Access Control	Granular RBAC	Docker EE only	Available	Enabled for CLI & API Interfaces	Yes	Yes
Write Only	Configurable	By default	Configurable	Configurable for CLI & API interfaces	Configurable	By default
Channel Protection	Configurable	By default	By default	Configurable	Configurable	By default
Readable only in Container	Easily accessible from host	Easily accessible from host	Easily accessible from host	Yes	Yes	Yes
Rotation	Only of mounted secrets (not env. vars.)	No	Requires recreation of Pods	Requires restart of services	Container must retrieve rotated secret from Vault	Rotated in Runtime
Revocation	No	No	No	No	On supported back-ends	Yes
Admin Audit	API HTTP Logs	Yes	API HTTP Logs	Via component aggregation	Yes	Yes
Usage Audit	No	Yes	No	Via component aggregation	Via administrative logs	Yes



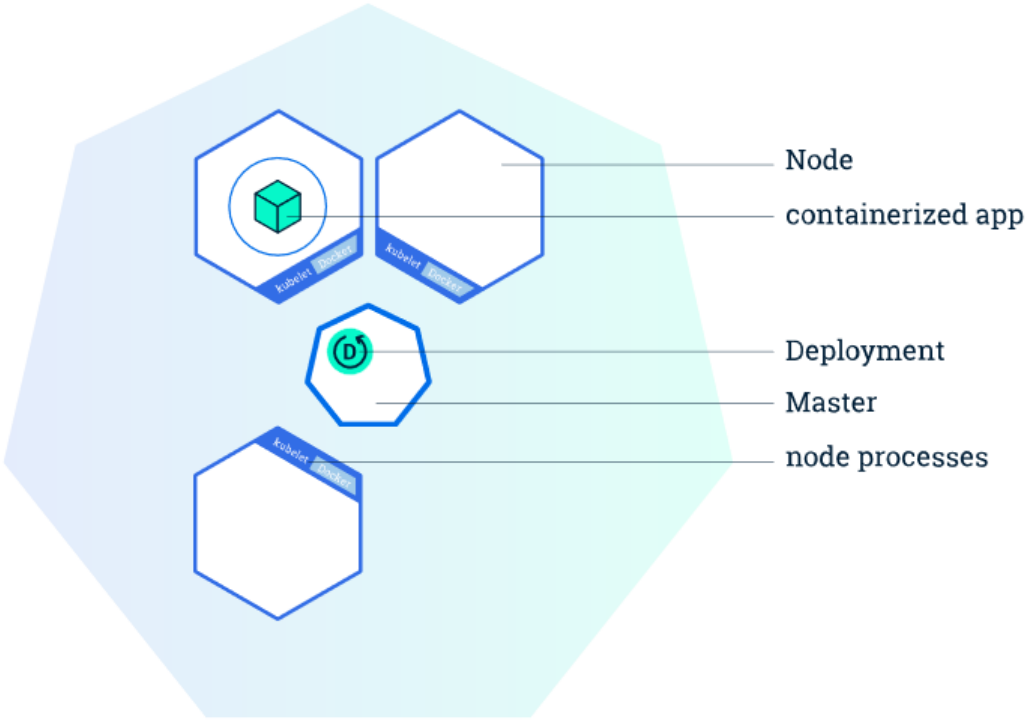
Deep Dive into Secrets Management Solutions

Kubernetes (version 1.6)

Kubernetes is an open source orchestrator that supports many different cluster deployments, including cloud, local VMs and bare metal environments.

A Kubernetes cluster is made up of a *master* and *nodes* (i.e., hosts). The *master* is in charge of running the API server, monitor the cluster state and schedule *pods*. Each *node* in the cluster is made up of one or more *pods*. A *pod* is the smallest unit in Kubernetes. It is made up of one or more containers that share the same resources such as network and storage.

Pods share resources (including secrets) via names in namespaces (not to be confused with linux namespaces). By default, every container in a *pod* is given a secret with the certificate of the master API, its namespace, and a token for authentication as its *service account*. The pod's service account can authenticate the the master's API using the token, and perform whatever action it is authorized to do.



Kuberneters Cluster

Kubernetes offers 2 methods to share secrets with its pods/containers: either via a file or through an environment variable. The files are mounted to tmpfs, so they don't persist on the host after a pod is cleared.

In the case of mounting secrets as files, a container can simply access the mounted volume and read the files. For example, the following figure shows a secret with two files, stored inside */etc/foo* on a pod named *mypod*:

```
root@mypod:/etc/foo# ls /etc/foo
password.txt  username.txt
root@mypod:/etc/foo# cat password.txt
-n "mysecretpassword"
root@mypod:/etc/foo# cat username.txt
-n "admin"
```

Another option is to distribute secrets as an environment variable, in which case the container can reference the variable by name:

```
root@secret-env-pod:/data# echo "$SECRET_STUFF"
-n "supersecretstuffhere"
```

Secrets, like any other state of Kubernetes, are stored in *etcd* in cleartext. Each secret reside in a namespace, and can only be referenced by pods in the same namespace.

Encryption at Rest

Like any other configuration or status of the cluster, the secrets are saved in etcd . The secrets themself are stored in cleartext. By default, etcd is running inside a pod, which is only scheduled on master nodes. Access to etcd should be limited to the *kube-apiserver* user only.

Because etcd is stored in cleartext, a privileged user running on one of the master node may be able to read secrets directly from the hard drive.

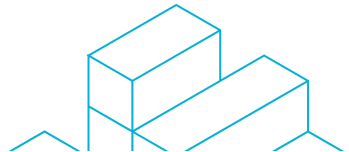
Access Control

Kubernetes supports RBAC (Role Based Access Control) which allows for fine grained authorization for users and service account. Roles are additive, there are no deny roles. Using this feature, it is possible to define which service accounts in which namespaces are allowed to access what resources.

Note: If this feature is not set, any node can read all the secrets in the cluster! To enable the feature, the *apiserver* must be started with the *--authorization-mode=RBAC* option (this is the default setting since version 1.6).

Write Only

To enable write only secrets, it is possible to define a role with only create access. Users with these roles will be able to create secrets, but won't be able to read their content.



Rotation

Secrets that are mounted as files are updated when they are changed on the master. Once a secret is updated, it is quickly propagated to all of its' consuming containers. However, this is not true for secrets that are set as environment variables: those are not updated!

To make the updated environment variable secret propagate to the pod, you will need to delete the old pod and start a new one.

Revocation

Secrets cannot be revoked directly from the API. They first need to be revoked wherever they were configured, and then deleted from all pods. In either method, secret deletion does not propagate to the pod - you must kill and restart all affected pods.

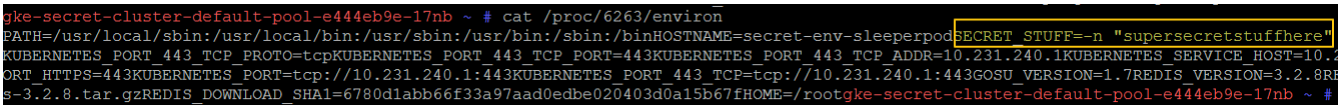
Channel Protection

Prior to version 1.6, if not specifically configured, communication between the apiserver and other entities in the cluster occurred over HTTP. In instances where it used HTTPS, the certificates were not verified, making the connection encrypted, but still vulnerable to man-in-the-middle attacks. Since 1.6, TLS is enabled by default. The server API identifies itself using a certificate, and the client can authenticate using multiple authentication schemas such as basic authentication, bearer tokens, certificates or authenticating proxy.

Readable only in Container

Both types of secrets can be relatively easy to extract if an attacker gains access to the host's filesystem as root. In the case of mounted volumes, the secret has a predictable location: */var/lib/kubelet/pods/<id>/volumes/kubernetes.io~secret/<folder name>/<filename>*

When a secret is deployed as environment variable, it can also be extracted by root or by user that is used by the Docker daemon: the attacker would only need the *pid* of the container that uses the secret and read its */proc/<pid>/environ* :



Administration Audit

It is possible to audit the api-server component. The audit is made up of all HTTP requests to the server, and responses returned from it. Changes to secrets are also audited, for example GET requests are requests to view secrets, while PUT requests change secrets value. The return value (the HTTP response value) indicates whether the request was successful or not.

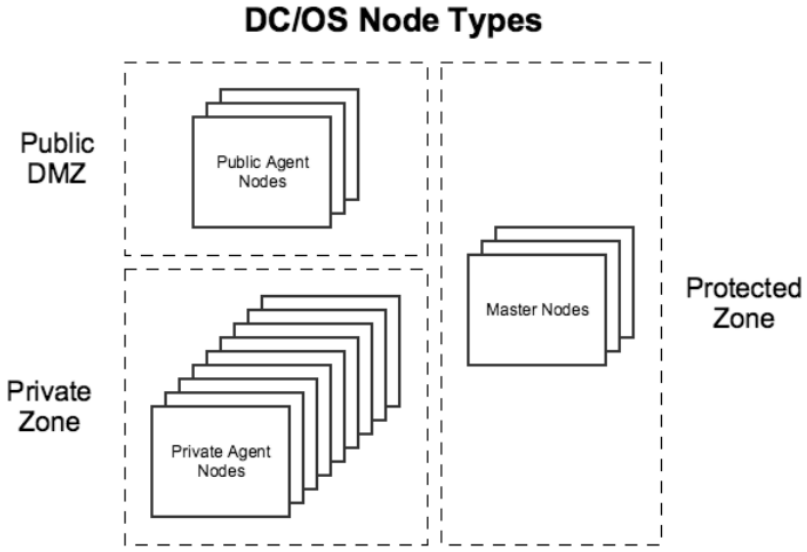
The audit is not “human friendly”, as not all people can intuitively follow HTTP methods and response codes.

Usage Audit

There is no audit on which secrets are used by which containers.

DC/OS Enterprise (version 1.9)

DC/OS is a cluster management and a container platform that abstracts the cluster to resemble the management of a single machine (via one of its core components: Mesos). It uses task schedulers such as Marathon to execute tasks as a container (mainly over Docker). Similarly to kernel space and user space, DC/OS abstracts services such as logging, networking and security in the cluster. Each task either runs in user space or in system space, and is distinguished by the owner and the service manager. System administrators run tasks in system space, while the rest of the users run in user space. A DC/OS cluster is made up of nodes: masters, private agents and public agents. User tasks run on agent nodes, public or private; depending on whether they require external access (such as reverse proxy load balancers). Most tasks run on private agent nodes.



DC/OS supports three security modes: *disabled*, *permissive* (default) and *strict*. While strict offers the best security, it is not the default one and is more complicated to configure. Each security mode has also implications on secrets - especially with regards to transmission.

Secrets in DC/OS are injected as environment variables to containers. They are stored encrypted in Zookeeper.

Encryption at Rest

Secrets are stored in Zookeeper, encrypted with an unseal key. The unseal key is also encrypted under a public GPG key. The private GPG key should be kept separately in a safe location, and used whenever the secrets store becomes sealed. Secrets are encrypted / decrypted whenever they enter / leave the store, respectively.

As long as you keep the unseal key safe, the secrets storage is protected.

Access Control

Via the web API, only a superuser can access and create secrets. The CLI and API interfaces can include permission for secrets for specific users & services to *read* / *create* / *update* or *delete* secrets. The Web API should not be regularly used for managing secrets, as it is too permissive. Instead, the CLI and REST API should be the main methods users access secrets.

Services running on nodes can be configured to use secrets as environment variables. It is possible to restrict a service's access to secrets by their path. Services can only use secrets that have a secrets path that matches their service group (or simply the service name):

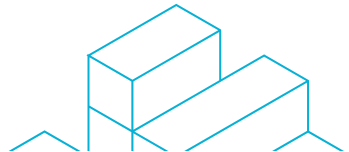
Secret	Service	Can service access secret?
`group/secret`	`/marathon-user/service`	No
`group/secret`	`/group/hdfs/service`	Yes
`group/hdfs/secret`	`/group/spark/service`	No
`hdfs/secret`	`/hdfs`	Yes

Write Only

As mentioned earlier, managing secrets should be done via the API or CLI interface. Via these interfaces, permissions per user can be configured so users can only create secrets, without the ability to view them. This enables an administrator to create users that can interact with "Write Only" secrets.

Rotation

Secrets can be updated from DC/OS interfaces. However, a service needs to be restarted to receive the new secret value.



Revocation

Revocation is not supported.

Channel Protection

DC/OS ensures that only encrypted secrets enter the store. This means that for whichever channel secrets use, they travel encrypted. This is useful, as communication between nodes is not always secure.

By default, connection to the *admin router* is not secure (HTTP). To secure the connection, you first need a valid TLS certificate and a private key on each master node. The admin router should be configured to allow only HTTPS traffic.

Another problem with the default security mode is that in cluster requests do not require authentication, meaning that deployed services in permissive mode may have too much (the *dcos:superuser*) permissions.

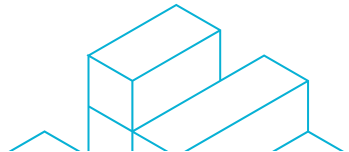
To truly secure secrets in transit, either *strict* mode must be enabled, or HTTPS configured and service authentication enabled for *permissive* mode. Even in strict mode, internode communication between Zookeeper instances is not encrypted.

Usage Audit

Similarly to administration, it is possible to audit usage of secrets by using logs. An example would be to monitor the *dcos-mesos-slave.service* component of a node running a container, which shows the usage of a secret:
result=allow object="Space: /mycontainer Request: [{\"environment\":{\"name\":\"MYSECRET\"},\"name\":\"supersecret\",\"type\":\"ENVIRONMENT\"}]" ...

RedHat Openshift (version 3.5)

OpenShift system wraps Docker containers and Kubernetes. Docker allows for packaging services as containers, and Kubernetes allows for orchestration and cluster management. The OpenShift platform adds support for managing and scaling your cluster, as well as user management and tracking. Because much of OpenShift features rely on Kubernetes, the architecture is similar. The cluster is made up of a master and nodes, which run pods. There are also projects, which are Kubernetes namespaces with additional annotations. Projects allow for specific units (such as developers and users) to have their own policies and constraints.



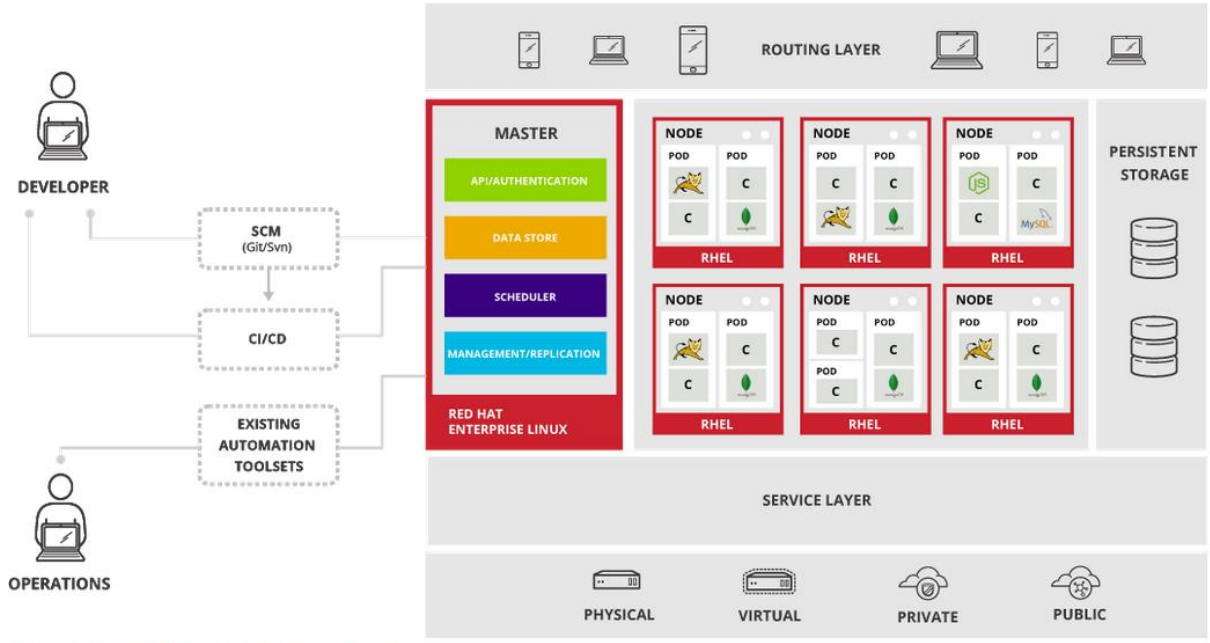


Figure 1. OpenShift Origin Architecture Overview

Secrets can be used as mounted tmpfs volumes or referenced as environment variables. Secrets reside in a namespace, and can only be referenced by pods in the same namespace. Under the hood, OpenShift uses Kubernetes for managing secrets. While it is more secure out of the box, and easier to configure, it shares many characteristics with Kubernetes.

Encryption at Rest

As with in Kubernetes, secrets are stored in *etcd* store, unencrypted.

Access Control

Access control is based on Kubernetes RBAC. Policies define which users and service accounts are allowed to access which resources. There are *cluster policies*, that apply to all projects, and *local policies*, which apply to a specific project. By default, service accounts have no access to resources: they cannot read all secrets for example.

Write Only

It is possible to allow a write only access to secrets for specific users / accounts.

Rotation

As in Kubernetes, secrets that are mounted as files can be propagated to pods when updated. However, secrets that are used as environment variables require that a pod is restarted.

Revocation

OpenShift does not support revoking secrets.

Channel Protection

Similarly to Kubernetes, communication with the API server is validated by nodes / pods with the server certificate. Each service account authenticates with a token to prove its identity.

Readable only in Container

Just as with Kubernetes, both types of secrets can be relatively easy to extract from the host.

Administration Audit

While the web interface supplies an Events tab, this is mainly for seeing which pods were created in the project. To monitor access to secrets, we need to use Kubernetes apiserver logs. As discussed in Kubernetes, these are a good start, but not robust enough.

For example, this is a log of a service account trying to list secrets, which it has no access to:

*<date time> AUDIT: id="0c0...d" ip="172.31.36.244" method="GET" user="**system:serviceaccount:my-project:default**" as="<self>" asgroups="<lookup>" namespace="my-project" uri="/**api/v1/namespaces/my-project/secrets**"*

*<date time> AUDIT: id="0c0..d" response="**403**"*

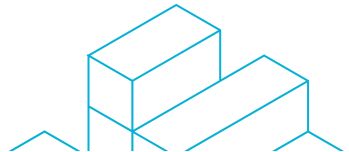
Usage Audit

There are no detailed logs that report which secrets are used by containers at runtime.

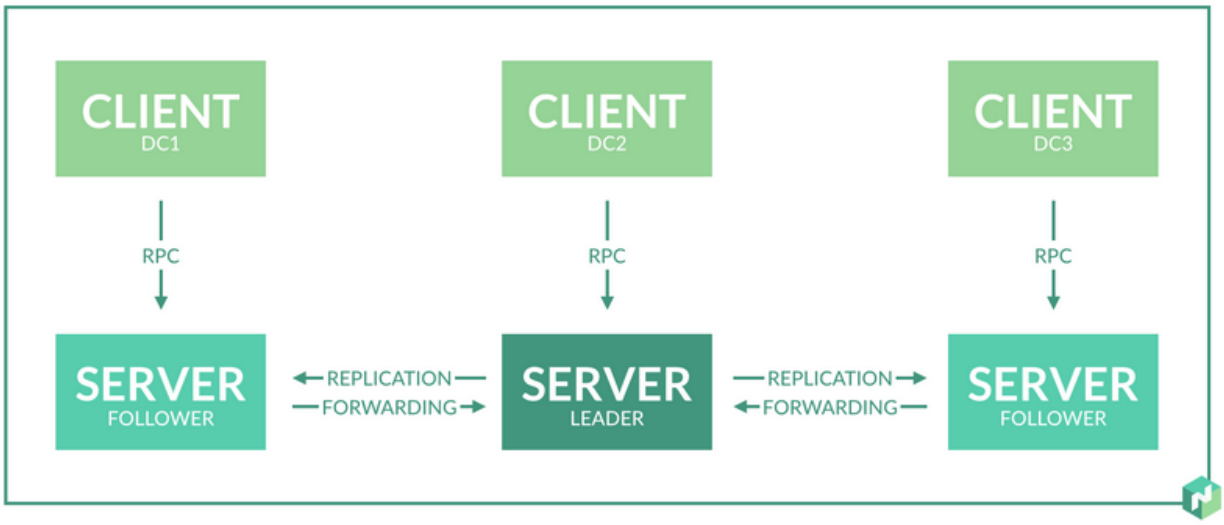
HashiCorp Nomad (version 0.5.6)

HashiCorp offers an orchestration solution called Nomad that can be integrated with HashiCorp Vault for secrets. A Nomad cluster is made up of clients and servers. Servers are responsible to keep the state of the cluster, and run tasks on clients. Each task can run with a different driver, so it is not limited to containers. Clients communicate with the servers using RPC calls where they register, report their status and receive allocations. Communication between all nodes (HTTP, RPC, Raft and Gossip) can be encrypted via TLS; except for gossip which uses a symmetric key.

To manage secrets, Nomad can integrate with Vault. Vault secrets read and write operations are similar to simple file access. A secret can also be generated dynamically (called *dynamic secret*) from the vault for supported infrastructures such as AWS. To integrate with Vault, each node server must be provided with a vault token that it can use to retrieve secrets. This token can have a “role” that gives the servers’ more fine grained access to Vault. Jobs that are submitted to the server, can be configured to use the vault. The Nomad server retrieves a token for the job, which is



retrieved by the clients, and then injected to the *secret/vault_token* file and VAULT_TOKEN environment variable of each task in the job. A job can also include a set of requested policies for the token. It can also be required from users that submit jobs to also include their Vault token.



Encryption at Rest

Vault data is always encrypted on disc. When a vault is initialized, it generates several unseal keys. Every time a Vault starts, it needs to be unsealed by a threshold (more than 1) number of unseal keys – like simultaneously turning keys to nuclear missile launch in a submarine. Only then are the secrets available in memory. The unseal keys should be stored in a separate safe location.

Access Control

To manage access to secrets, Vault uses access tokens. All the tokens are derived from the root token. A child token can also be used to generate more child tokens. Each child token inherits the permissions of its parent, but can also have a subset of the parent's permissions. When a parent token is revoked, all of the child tokens are revoked.

Write Only

It is possible to configure a policy that enables a user (with a token) to create secrets, but not read them.

Rotation

Tasks use secrets via access tokens that they are given. Each task must access the vault with its' token to get a secret. To rotate a secret, all tasks must periodically check the vault for

changes in their secrets. Rotation is therefore possible, but it requires platform specific awareness on the side of the containers.

Revocation

Secrets in Vault can have a lease time, after which they are automatically revoked. For dynamic secrets (configured on supported backends) the secret is not only deleted from the vault, it is also invalidated in the backend. For example, revoked AWS backend secrets will remove access keys from AWS.

Channel Protection

HTTP and RPC communication between clients and servers can be configured to be encrypted and mutually authenticated via TLS. To enable this feature, all clients and servers must be provided with key pairs that are generated and signed by a CA.

Readable only in Container

Containers running in the cluster don't have direct access to secrets. They have access to tokens that enable them to retrieve secrets from the Vault. These tokens are stored as environment variables and mounted as *tmpfs* to containers. The token is only readable inside the container.

Administration Audit

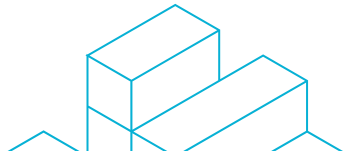
Each access to Vault can be audited. Data includes which token was used to authenticate, the operation that took place, and the address from which the Vault was accessed.

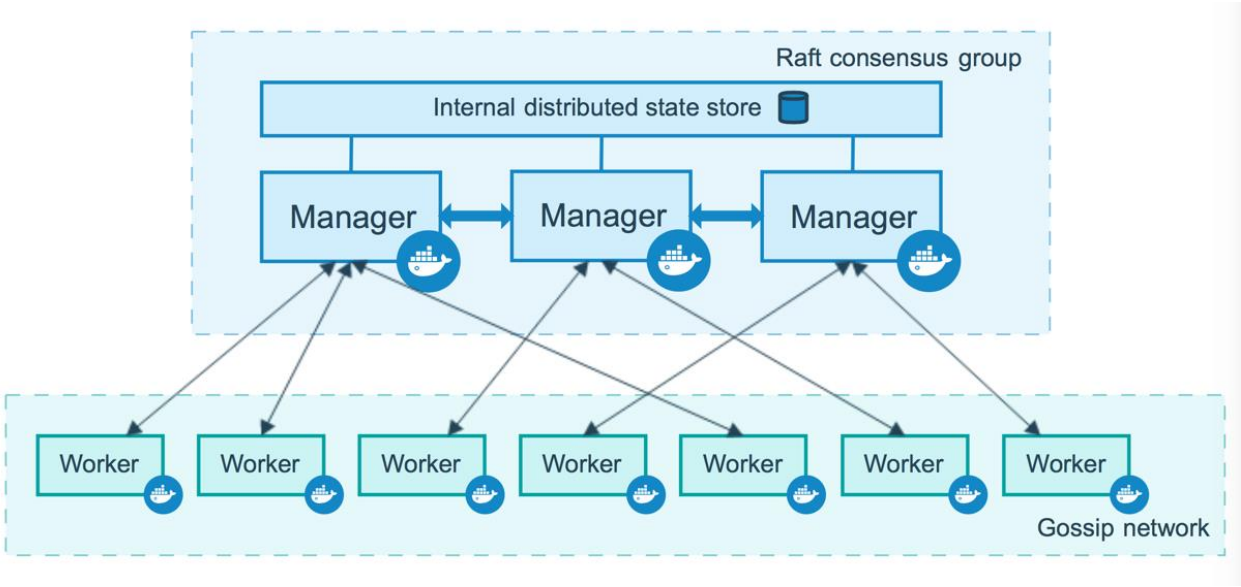
Usage Audit

Because each access to a secret has to go through Vault, the administration logs also allow to audit which containers accessed which secrets.

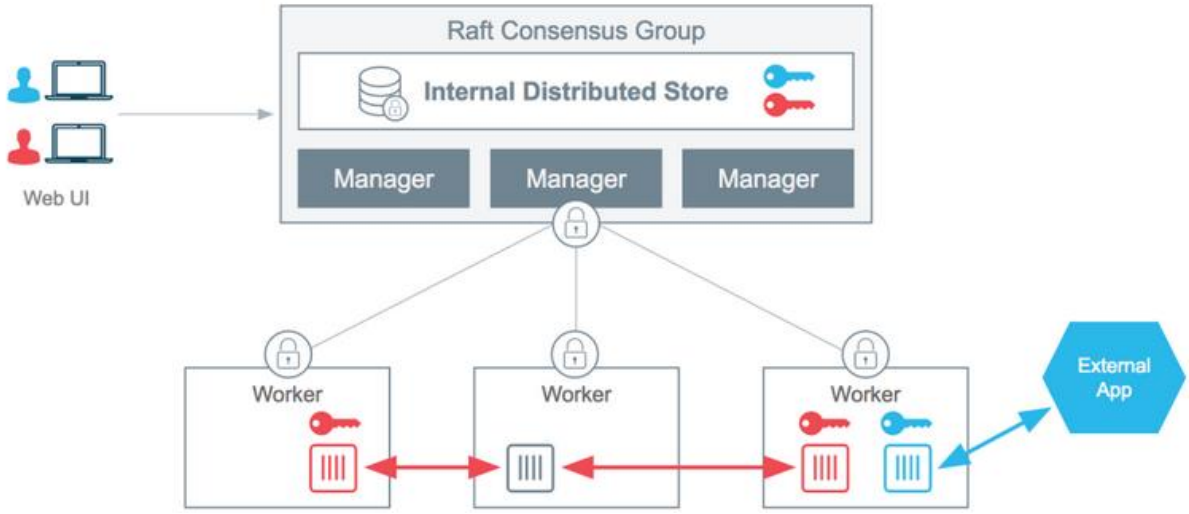
Docker Swarm / DEE (version 17.03)

Docker *Swarm* is a cluster orchestration application, made up from *nodes*. Each node in the cluster is a *Docker Engine* that participates in the *swarm*. In *swarm*, the user orchestrates *services* - which are *tasks* scheduled to worker nodes. One *manager* node is responsible for dispatching *tasks* to *worker* nodes. A service specifies which container images to use, along with the configuration of that container, number of replicas, command line parameters etc.





Swarm features secrets, which can be mounted as *tempfs* to containers in the swarm, which can then be read as plaintext by the container at */run/secrets/<secret name>*. The secrets themselves are stored in Raft, that can also be sealed to secure the secrets storage. Secrets are then distributed over secure channels to workers.
For enterprise environments, Docker Enterprise Edition should be used, as it offers some crucial functions such as *access control* and *audit*.



Encryption at Rest

On manager nodes, secrets are decrypted in memory and encrypted on disk. On worker nodes, only the relevant secrets for the services on the node can be found, mounted as *tmpfs* at */run/secrets/<secret name>* inside the container. The host's root can also access these secrets in cleartext from */var/lib/docker/containers/<container-id>/run/secrets.*

Secrets, and other sensitive information, are stored encrypted in a Raft log. These logs can be decrypted using the encryption key of the node, found in */var/lib/docker/swarm/certificates* (accessible to root). To dump the contents of the Raft log, we can use a tool called swarm-rafttool, revealing the unencrypted secrets:

```
Entry Index=29, Term=2, Type=EntryNormal:
id: 100250812792091
action: <
  action: STORE_ACTION_CREATE
  secret: <
    id: "dddi7ob32jvafljasyvl76jzw"
    meta: <
      version: <
        index: 28
      >
      created_at: <
        seconds: 1491118906
        nanos: 958910904
      >
      updated_at: <
        seconds: 1491118906
        nanos: 958910904
      >
    >
    spec: <
      annotations: <
        name: "my_secret"
      >
      data: "don't tell anyone!\n"
    >
  >
>
```

When the encryption key is right besides the encrypted data, one can't sincerely state that that secrets are encrypted on the disk. To truly protect the private encryption keys of the swarm, there is an autolock feature. It provides the user with an *unlock key*, which can be used to unlock the swarm.
The *unlock key* should be stored in a different, safe location. It is also possible to rotate this key: *docker swarm unlock-key --rotate*

Access Control

Out of the box, Docker Swarm offer binary access control: You either have access to everything, or to nothing at all.

If you use the Docker Enterprise Edition, you can manage secrets using RBAC and labels. Labels can be given to a secret (or any other resource), then different permissions can be given to users for that secret, based on the labels.

There are 4 levels of permissions: *Full Control*, *Restricted Control*, *View Only* and *No Access*.

Write Only

Once a secret is created, it cannot be viewed or edited from the API.

Rotation

Secrets cannot be updated. In order to rotate a secret, you first need to delete it from all services using it, then remove and recreate it.

Revocation

Secrets can be easily removed from services, and then deleted. However, the secret is not revoked.

Note that the *encrypted* secret may still reside in the raft log, even when deleted.

Channel Protection

Communication between all node types is done over TLS, and is mutually authenticated. Each node receives a private key and certificate, plus the root certificate of the swarm. An external CA can also be used. By default, each node renews its certificate every 3 month; this value can be configured to be as frequent as once every 1 hour.

Readable only in Container

Each service / container in the swarm can only read its relevant secrets. The secrets are “read only”.

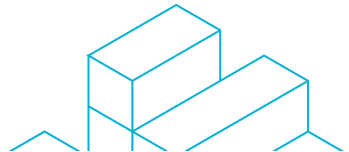
The host’s root can access these secrets in cleartext, as they are mounted on the host at : */var/lib/docker/containers/<container-id>/secrets.*

Administration Audit

Available in the Enterprise Edition. Logs can be configured to propagate to a remote logging service. Each log has a different logging level (DEBUG, INFO, WARN etc.). Secrets administration is logged at the INFO level. For example, this is an entry of a secret being deleted by user *aqua (msg field contains the ID of the secret)*:
*{"level":"info","license_key":"8R...k_w","msg":"puno0hth3s6gjbss92ihzyqhk","remote_addr":"37...:54294" ,"tags":["api","secret.delete"],"time":"2017-05-07T12:59:28Z","type":"**secret.delete**","username":"**aqua**"}*

Usage Audit

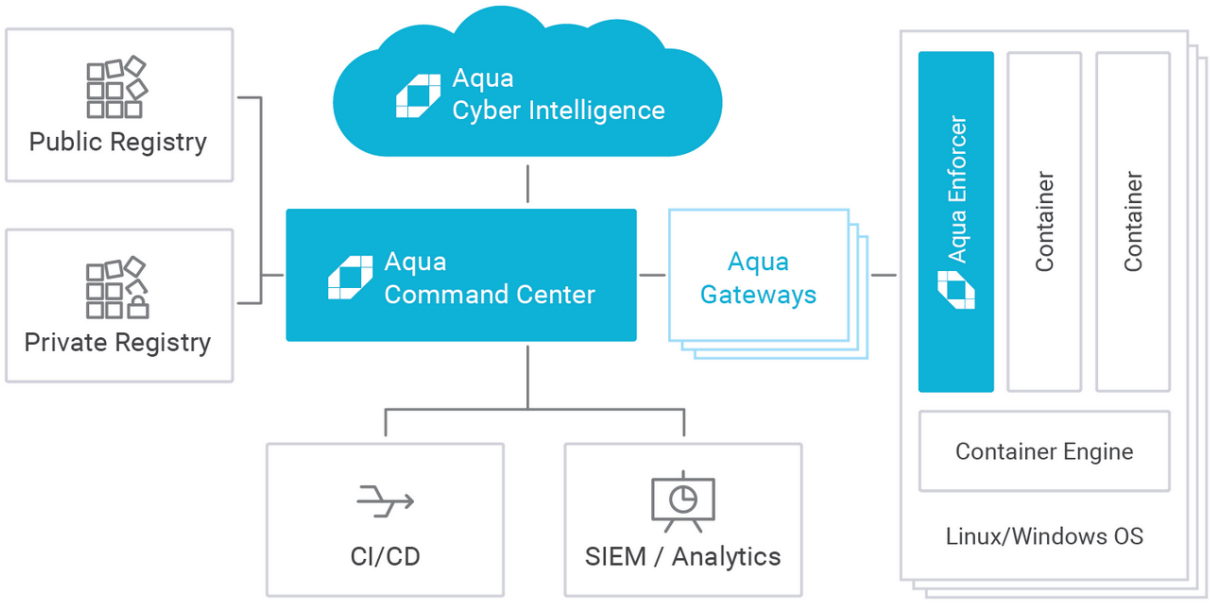
Detailed logs on secrets deployment is available in the Enterprise Edition.



It is possible to get usage information from Docker logs in Swarm, but it will require some additional processing.

Aqua Security

Aqua is a security platform for containers. It can be integrated into existing orchestrators and cloud platforms, such as Kubernetes, Openshift, DC/OS and Swarm. An aqua agent is installed on all hosts, and retrieves configuration and policies via the Aqua Gateways.



Logical diagram of Aqua main components

One of Aqua's security features is Secrets management. A secret can be injected into containers via environment variables or mounted as files, where it is only visible in plaintext to the container itself. Outside the container, the secret is encrypted. Aqua integrates into other secret stores such as HashiCorp Vault, Amazon KMS, Azure Key Vault or CyberArk Enterprise Password Vault. Secrets and communication between containers are enabled by grouping containers into a *service*.

Encryption at Rest

Secrets are always encrypted at rest via 3rd party storage. Access to secrets is enabled via access tokens or existing IAM roles of a host, while the encryption is taken care of by the storage solution.

Access Control

Access to secrets can be controlled for users (optionally integrated with Active Directory) and services; which group together containers that share similar security characteristics.

Write Only

Once a secret is created, it cannot be viewed from the web interface or the API. You can only update the secret's content of delete it.

Rotation

When a secret is changed, the updated value is injected into a running container. A container does not have to be restarted for changes in secrets to be propagated.

Revocation

When integrated with platforms that support revocation, such as HashiCorp Vault, deletion of a secret in the Aqua Command Center will also revoke the secret.

Channel Protection

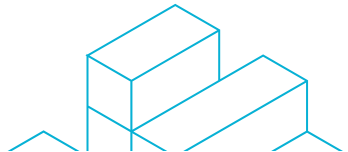
Communication between Enforcers and Gateways is SSH encrypted and mutually authenticated. The Enforcer verifies the Gateway via its public key, and the Enforcer identifies itself using an authentication token.

Readable only in Container

Secrets are mounted as files to tmpfs file system, and are also available as environment variables. They cannot be trivially read from the underlying host. When environment variables are read from an outside process, the calling process only sees the name of the secret, and not the secret itself.

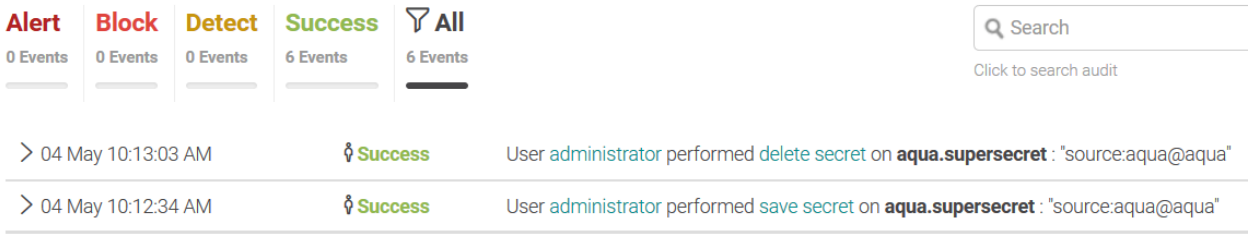
For example, the 22018 process is running inside a container that is using the *DBPASSWORD* environment variable. When accessing these variables from outside we only see the name of the secret:

```
root@docker:~# cat /proc/22018/environ
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/binHOSTNAME=85e7dde11188PASSWORD={aqua.DBPASSWORD}LD_PREL
```

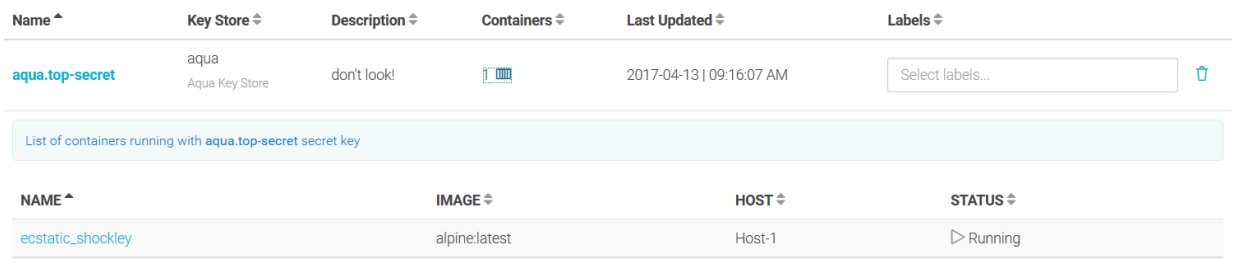
Administration Audit

Each administrative access event to a secret is logged.



Usage Audit

During runtime, Aqua shows which containers have access to which secrets:



* * *

Questions? Comments? Email us at contact@aquasec.com

Aqua enables enterprises to secure their virtual container environments from development to production, accelerating container adoption and bridging the gap between DevOps and IT security.

The Aqua Container Security Platform provides full visibility into container activity, allowing organizations to detect and prevent suspicious activity and attacks, providing transparent, automated security while helping to enforce policy and simplify regulatory compliance.

For more information, visit www.aquasec.com

aqua