Red Hat Consulting

OCP 4.x Platform & Infrastructure Security Best Practices

Public Edition, v2

March 02, 2021

Tommer Amber, Red Hat
Cloud Security Consultant
Israel

Executive Summary

In the last couple of years the entire app-dev sector has changed dramatically; Ever since the introduction of container technologies and the high growth rate of the DevOps field - cloud technology has become the standard solution for production environments in many organizations.

One significant expression of this technology evolution process is the wide adoption of container orchestration solutions (such as “Vanilla” Kubernetes [K8s] and OpenShift [OCP]).
In no time multiple deployment options became standards, such as public cloud provider offerings (AWS, GCP, etc.), and on-premise installations.

These platforms are allowing fast and short development-to-production (CI/CD) circles, for large-scale workloads (including different redundancy options for the entire IT stack; i.e. networking, storage, etc.), in a comfortable, centralized, self-service oriented environment.

This trend is just getting stronger and being adopted by more and more organizations, and the migration from “legacy” IT solutions for applications to containerized environments has had a vast impact on the entire field; It won't be an exaggeration to dictate that we will see most (or even all) our applications deployed on these solutions in the near future.

Both Kubernetes and OpenShift are popular container management systems, and each has its unique features and benefits. While Kubernetes helps automate application deployment, scaling, and operations, OpenShift is the container platform that has been built on top of Kubernetes to help applications run more efficiently, and it's (OpenShift) extends Kubernetes’s array of capabilities dramatically, to grant the developers and DevOps teams the best experience and makes the migration to cloud enterprise standard an easier, continuous process overall.

Despite the sharp rise of these platforms, many security teams (that previously had complete control and visibility capabilities over the network and data center) did not close the gap on container technologies knowledge and risks that are threatening those platforms, to secure them properly.

This document describes in detail the different aspects of security on OpenShift Platform, in a way that explains the potential risks, what are the platform features offered as answers to mitigate the risks, and additional information on the platform itself (OCP 4.x); All those topics are relevant to the overall understanding of cluster-admins & security teams on how the platform works differently behind the scenes than older versions, which may be useful when trying to

integrate complement products to enhance the security cover of the environment & applications that run on top of it.

The document displays security features in the following platform's sectors:

- Authentication
- Authorization (RBAC)
- Correct platform & infrastructure resources management
- Networking & applications' exposure
- Infrastructure (nodes & hosts) security suite (SCC)
- Sensitive data encryption (etcd)
- Image sources verification & images signing
- Useful security operators offered by Red Hat
- And much more

Under each section, we will review the topic in general, the relevant risks, and the right ways to cope with them on a technical side, including examples & links to the relevant, official documentation.

By the end of the document, there is a checklist summarizing all technical best practices written throughout this document ..

Table of Contents

1. Preface

1.1. Confidentiality, Copyright, and Disclaimer

This document is not a quote and does not include any binding commitments by Red Hat. If acceptable, a formal quote can be issued upon request, which will include the scope of work, cost, and any customer requirements as necessary.

1.2. Audience

This document is intended for the Client technical staff responsible for the environment.

1.3. Additional Background and Related Documents

This document does not contain step-by-step details of installation or other tasks, as they are covered in the relevant documentation on http://access.redhat.com/. Links to the appropriate documents will be made when required.

1.4. Scripts and playbooks

Any scripts provided are being provided as-is, without any form of support or warranty.
All provided scripts can be modified by the customer at will.

2. Prefix

This document describes how security is addressed at the core layers of the OpenShift 4 technology stack, and how compliance and regulatory concerns can be mitigated.

The cloud infrastructure and security engineering roles are central to establishing and preserving security postures. The document intends to support these roles by providing the proper mixture of conceptual, organizational, and technical guidance, thereby increasing the security vigilance and effectiveness of those with such responsibilities.

The document will not contain explicit information regarding complementary products such as 3Scale, RHSSO, Quay, RHACM, RH Service Mesh, etc. It will solely focus on the capabilities that come built-in within the platform and infrastructure.

Any other relevant materials on add-on products will be referenced briefly and with links to the relevant documentation/articles, or either will require a separate task entirely which will be scoped as well to specific products.

3. Overview

3.1. Disclaimer

This document is intended to elaborate on OpenShift 4.x Security (specifically), therefore it requires proper knowledge of Kubernetes, OpenShift 3.x/4.x (Optional), and a good understanding of containers technology as a whole; It's designed for day-to-day operators/administrators of OpenShift environments. It grants them enhanced capabilities in the field for their clusters' deployments.

Currently, in accordance with the writing of this document, the most updated version of the OpenShift Platform is 4.6.5 and it will include relevant data up to that point in time.

Furthermore, this document is aimed mostly at disconnected environments, as such, many connected-oriented solutions won't be part of it.

3.2. Before You Start! - Document Structure

This document has been written specifically for cluster administrators, and it is separated into the two main disciplines under the responsibility of the OpenShift (OCP) Platform Team that every organization that embraces the microservices approach will require to have;

The two main disciplines are:

1. Platform
2. Infrastructure

Under those disciplines there are multiple sections, each tackles a dedicated subject, including a "best practices" section at the end.

At the end of the document, there is an "Appendix" section, it includes a very detailed explanation of the built-in security capabilities that are an integral part of the OpenShift 4.x specifically, such as explanation on Operators and Cluster Operators and RHCOS that makes the OpenShift 4.x clusters much more resilient and protected by default, in comparison to OpenShift 3.x or even to Kubernetes upstream clusters. This last part is included in the appendix at the end of the document.

And finally, a table that combines all the relevant best practices in a single place for easier management & implementation.

3.3. OCP CIS Benchmark Correlation

On Jan 10th, 2021, Red Hat has released the Red Hat OpenShift 4 Hardening Guide, inspired by the OpenShift CIS Benchmark Draft. As the date of the release of this document, the benchmark is labeled as draft and is work-in-progress.

The following document is correlated with the CIS Benchmark Draft, and adding extra, crucial & easy to implement operations to raise the security cover significantly.

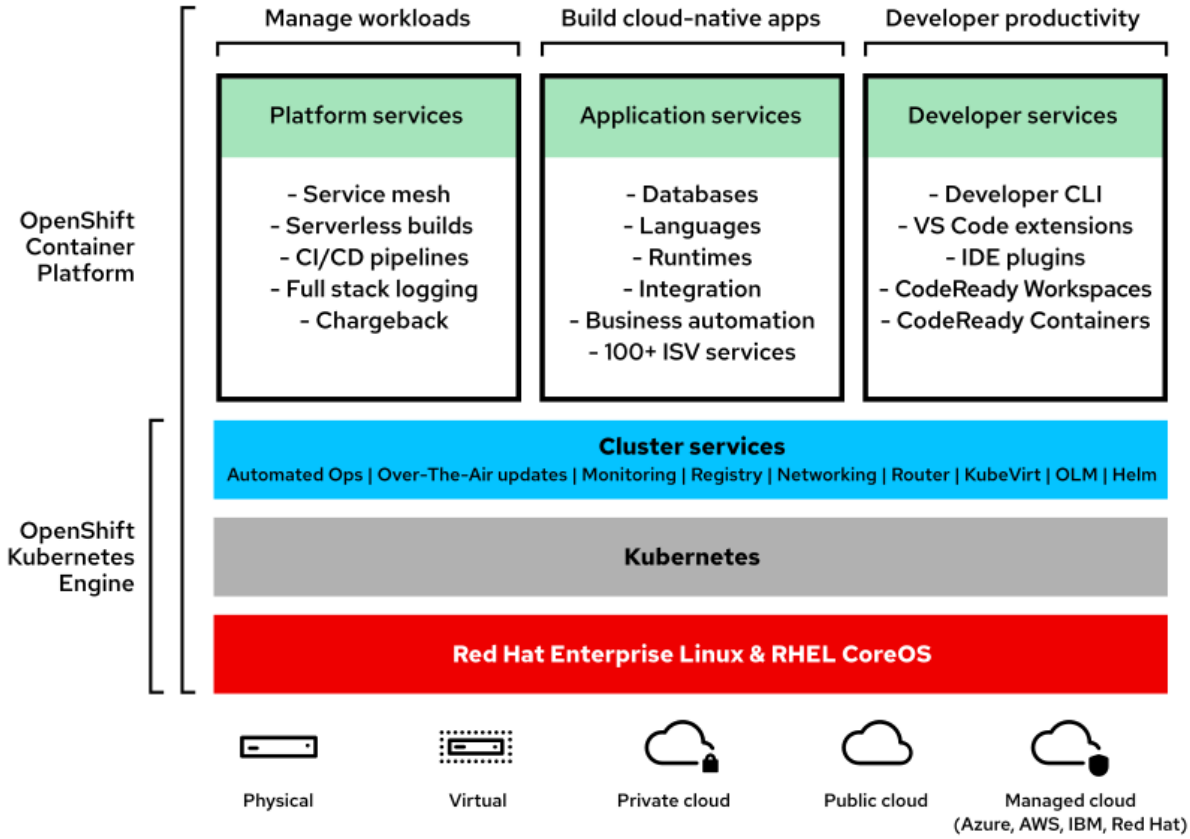
4. OpenShift Security Motivation

4.1. OpenShift Overview

Red Hat OpenShift Kubernetes Engine is a product offering from Red Hat that lets you use an enterprise-class Kubernetes platform as a production platform for launching containers.

Kubernetes provides the basic container orchestration and cluster services providing advanced automation management features. OpenShift, which is built on top of Kubernetes, provides a wider, enriched platform for developers to build, deploy, and manage their migrated applications to microservices architecture in an easier, enterprise-class manner.

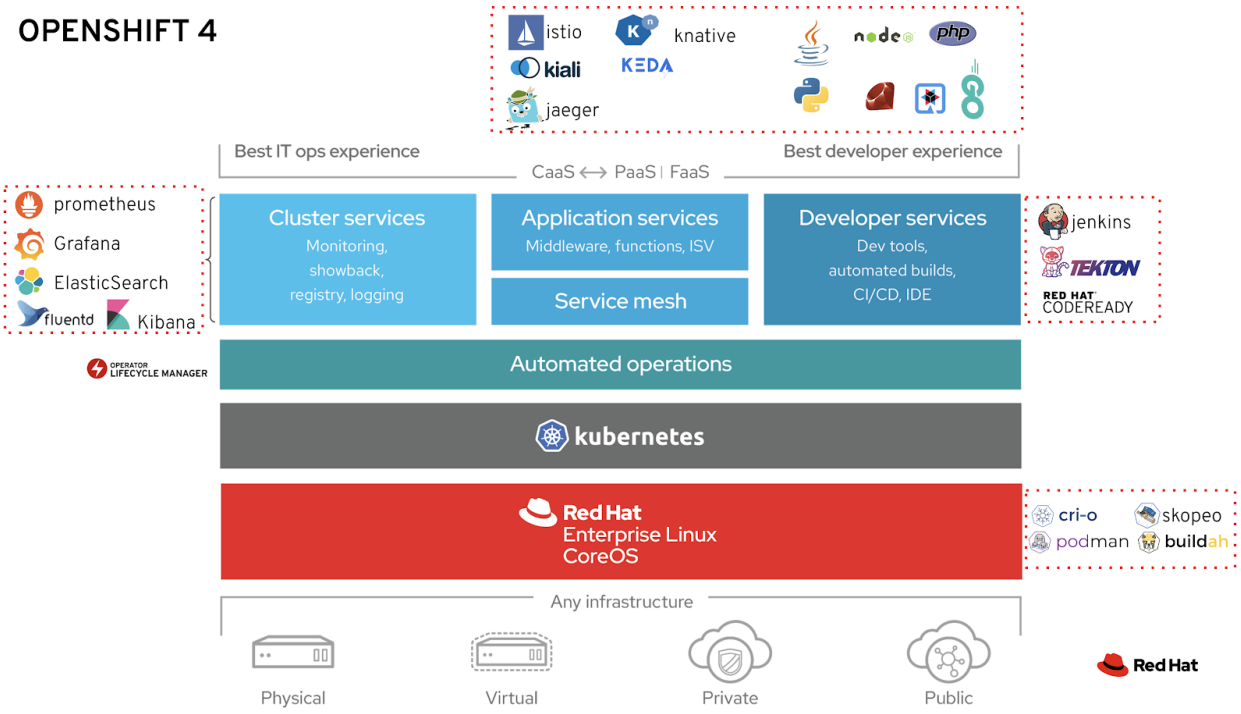
Before we go deeper into the security aspects of OpenShift, first we need to understand what's included in the OpenShift technology stack.



Starting from the lowest layer, OpenShift can be deployed on a variety of infrastructure which includes bare-metal servers, virtualized environments, private, public, managed, and hybrid cloud environments. Red Hat Enterprise Linux (RHEL) and RHEL CoreOS are the underlying operating systems upon which OpenShift is run. The Kubernetes container orchestration engine is at the core of OpenShift.

On top of that OpenShift offers multiple features and capabilities for developers to enhance their productivity by simplifying Kubernetes-Vanilla features (for example by providing “Route” objects that Kubernetes does not have for easy service exposure), providing automation capabilities for continuous CI/CD processes, easier management of multiple services at different scales, etc.

Around OpenShift there is an entire ecosystem of existing & potential compliant products (such as the cri-o engine, podman, Prometheus, operators, etc.) that are bringing even more capabilities to OpenShift to make its management a holistic experience.



4.2. OpenShift security document rational

4.2.1. Why

There are multiple possible attack vectors when we examine the OpenShift Platform security aspect.

An experienced attacker that resides on our network initially scans the network for sensitive resources that he/she can use, for his/her advantage once they are reached, like data leakage and denial of service attacks, just to name a few;
OpenShift is a centralized platform that inhibits multiple applications and their data is a natural place for an attacker to gather information and potentially cause harm.

This document is intended to give a deep perspective on the different types of vectors that might be interesting for an attacker, and how the cluster admins could (& should) mitigate these risks.

4.2.2. Main Attack Vectors around OpenShift

We need to be aware of the entry points that an attacker may be interested in and what risk potential each of them has.

When it comes to reverse engineering an attacker's mind, we need to take into assumption what the attacker already achieved in any given point and how he/she could proceed further with his/her intentions.

Attack Vector #1 - Organizational Network

The attacker has access to our networks, but not to the platform.

The attacker will need to figure its way to the platform by using multiple lateral movement techniques.

Our main concerns in that regard should be:

1. The platform users, that can be stolen. (Authentication & Authorization)
2. Our external image registries (that can be compromised if not protected properly).
3. The allowed network access to the platform nodes.

- This document will not discuss the full-CI/CD process because even though it is tightly related to OpenShift (infra) it is a separate process entirely (app-dev) that should be treated as such.

Attack Vector #2 - Malicious Application

The attacker has access to an exploitable application's container.

This case is interesting because even though the container is encapsulated within the namespace territory and has limited permissions based on its service account's SCC policy, it is possible to make progress (from an attacker perspective) and it's important to know that and to use real-time container's behavior scanning mechanisms.

An attacker at that point can also inject malicious data / interrupt with the data itself, send misleading data & messages to different components and apps, etc.

Another interesting point to note is that if the attacker has standard (developer) access to the cluster with permissions to deploy its services - he/she can get to that exact point (in-container access), but more easily than find an application's weakness.

This will not be elaborated further than discussing SCCs, and verifying external image sources, mostly because it's an attack vector that relates mostly to the application itself; It evolves code scanning (static & dynamic) and testing, which is not related directly to the OpenShift platform and infrastructure.

Attack Vector #3 - Malicious Unprivileged OpenShift User (App-Dev)

The attacker has stolen a legitimate OpenShift identity (standard user), a developer's user for example.

Once this has happened, the attacker will want to escalate its privileges for the sake of getting simple, trivial, covert access to sensitive data.

Or, another vector could be to disturb standard applications from working properly / leak data.

Our main concerns in that regard should be:

1. Object creation with malicious content
2. Object deletion
3. Private services exposure
4. Cross-projects/namespaces lateral movement
5. Changes in Role-Based Access Permissions
6. Object creation with elevated privileges (SCC)
7. Resources exhaustion
8. Access to applications' sensitive data (secrets, service accounts, logs, events, etc.)

Attack Vector #4 - Malicious OpenShift Administrator

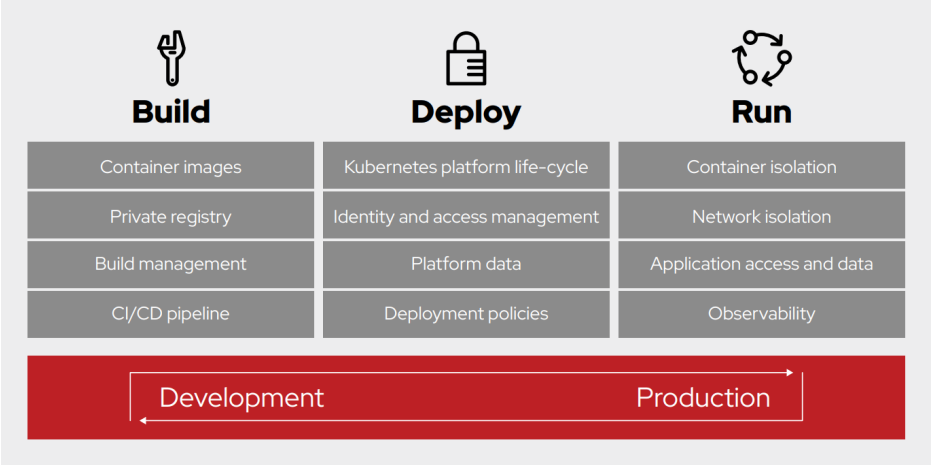
The attacker has access to a cluster-admin user.

At this point it's basically "game over", and the attacker can interrupt with our entire cluster in whatever way he/she would like to.

To minimize possible damage, it's important to integrate a vast monitoring & auditing system, with security-based analysis capabilities, that will detect an attacker based on its non-standard actions (such as editing/deleting conf files on the infrastructure nodes, access attempts to non-OpenShift infrastructure entities at the rest of the network, change users' permissions cluster-wide, etc.) before any irreversible damage will be done.

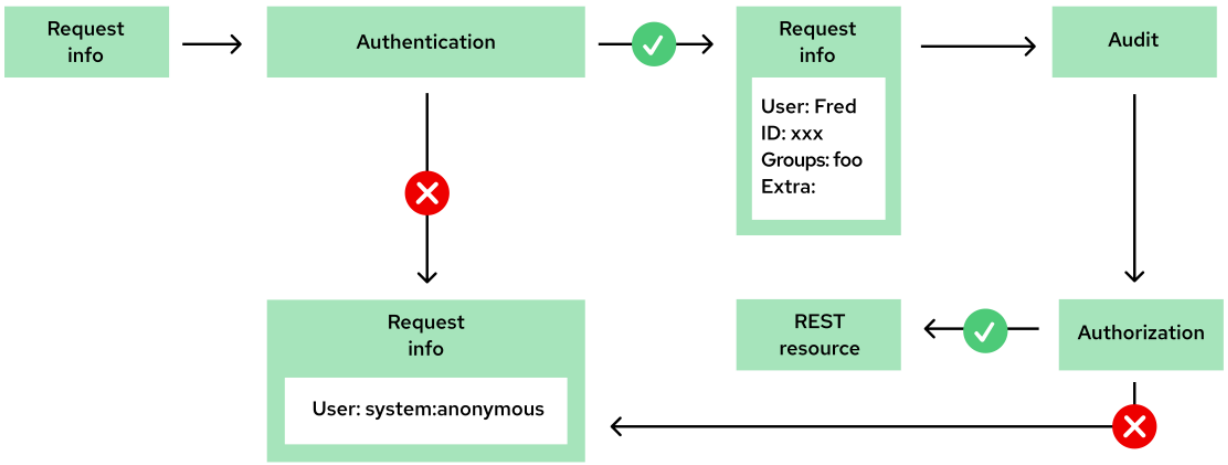
4.2.3. Suffix

Once we have a good understanding of the different attack vectors and possible entry points/misconfiguration that an attacker can take advantage of - we can build a security suite to cover our OpenShift Platform and Infrastructure from top to bottom (under the constraints of what OpenShift provides us with by default) - And this is exactly what this document all about.



5. Platform Security

5.1. Authentication



For users to interact with OpenShift Container Platform, they must first authenticate to the cluster. The authentication layer identifies the user associated with requests to the OpenShift Container Platform API.

The Authentication process can be done in multiple ways in OpenShift, and each methodology can be relevant for a subset of users based on their role in the organization. E.g. Admins can be required for a more secure authentication mechanism because their users can perform more cluster-impactful tasks.

In general, we have two main protocols to authenticate users, each can be implemented in multiple variations;

1. OAuth (access tokens obtained after a proper “login” process)
2. X.509 client certificates (HTTPS + Certificate validation against CA bundle)

5.1.1. OAuth

The OpenShift Container Platform master includes a built-in OAuth server.

A user who desires to obtain a new access token for accessing the platform sends a request (contains the credentials of the user) to the OAuth server, that verifies the user’s identity against an “identity provider” (3rd party “trusted” authority) configured for the cluster.

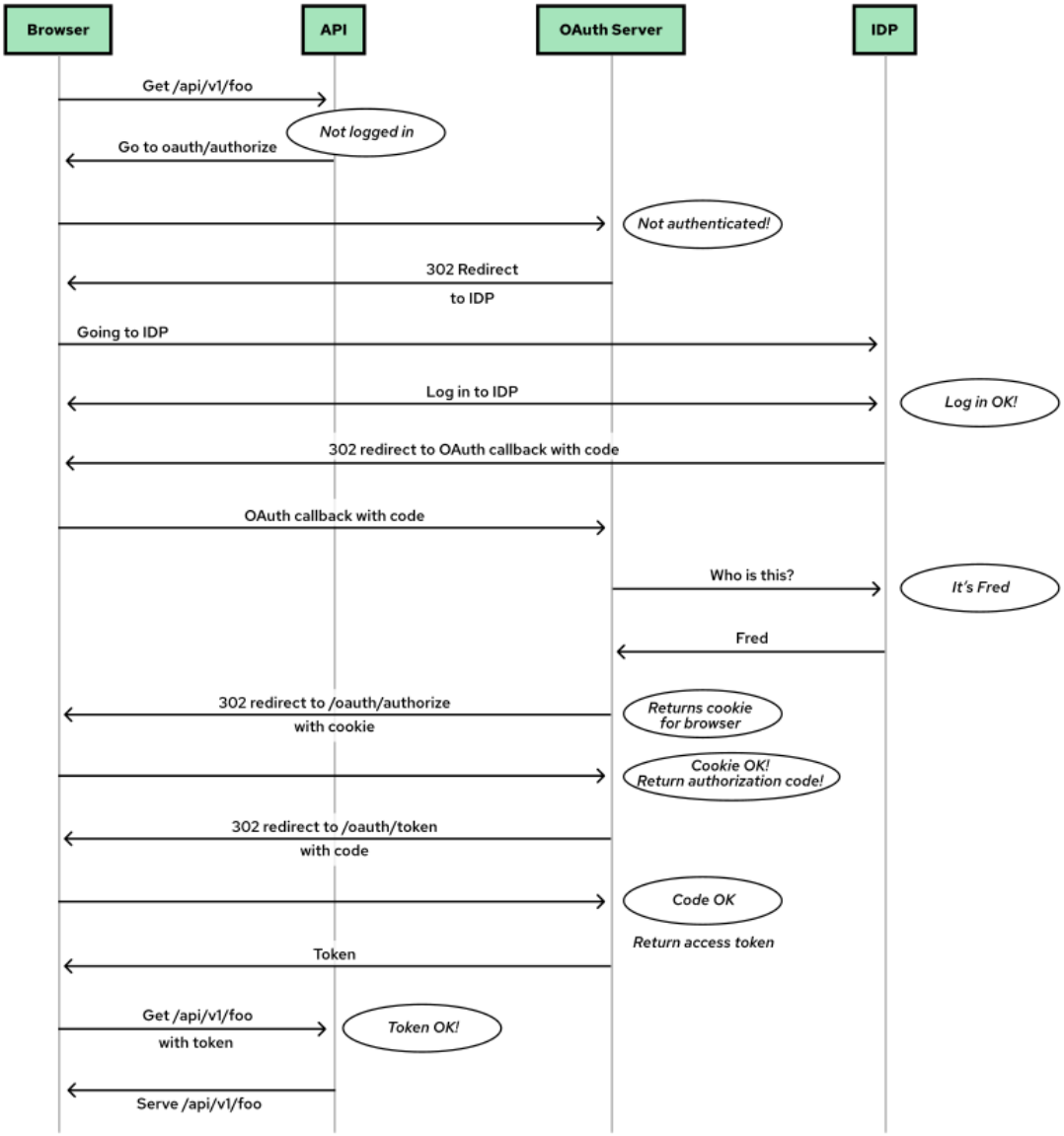
This identity provider can be one of the following:

- htpasswd file
- Keystone (internal DB extension for OCP)
- LDAP
- Basic Authentication
- 3rd party OIDC Authentication Server (Github, GitLab, Google, etc.)
- On-Premise OIDC Server such as RHSSO

You can define multiple identity providers, of the same or different kinds, on the same OAuth custom resource.

5.1.1.1. Best Practices for OAuth

Use OIDC for user authentication to your clusters.



It is far more advanced than Basic Authentication/HTPasswd or any other option.
It also enables us flexibility on the authentication flow; E.g. our OIDC main server (RHSSO for example) can implement the authentication flow process with many protocols to choose from - such as using Kerberos/Client Certificate etc. that would all be covered by the secured OIDC algorithm mechanism.

Furthermore, our OIDC server could also function as another auditing source for login attempts monitoring, and enforce shorter login sessions by reconfiguring the refresh rate policy of the JWT tokens to prevent an attacker from having a long operation time.

By default, only a ***kubeadmin*** user exists on your cluster. To specify an identity provider, you must create a custom resource (CR) that describes that identity provider and add it to the cluster. More on the kubeadmin user in the next section.

An example of such configuration in the OAuth CR object:

1. Identity providers use OpenShift Container Platform **Secret** objects in the **openshift-config namespace** to contain the client secret, client certificates, and keys to perform the user-verification against the provider.

```
# Option #1: From Literal String
$ oc create secret generic <secret_name> \
--from-literal=clientSecret=<secret> -n openshift-config

# Option #1: From File
$ oc create secret generic <secret_name> \
--from-file=/path/to/file -n openshift-config
```

2. Identity providers use OpenShift Container Platform **ConfigMap** objects in the openshift-config namespace to contain the **certificate authority bundle**. These are primarily used to contain certificate bundles needed by the identity provider.

```
$ oc create configmap <ca-config-map-name> --from-file=ca.crt=/path/to/ca -n
openshift-config
```

3. **And finally the CR itself**

```
cat > idp_cr.yml << EOF
apiVersion: config.openshift.io/v1
kind: OAuth
metadata:
  name: cluster
spec:
  identityProviders:
  - name: oidcidp
    mappingMethod: claim
    type: OpenID
    openID:
      clientID: <client id created in the idp for OCP specificly>
      clientSecret:
        name: <previously_created_secret_name>
      ca:
        name: <ca-config-map-name>
      extraScopes:
      - <optional-scope#1>
      - ...
      - ...
      extraAuthorizeParameters:
```

```
        include_granted_scopes: "true"
      claims:
        preferredUsername:
        - preferred_username
        name:
        - name
        email:
        - email
      issuer: https://www.<issuer_url>
EOF

$ oc create -f idp_cr.yml
```

Note!! If you must specify a custom certificate bundle, extra scopes, extra authorization request parameters, or a userInfo URL, use the full OpenID Connect CR.

For additional information on the topic: Link

5.1.1.2. Client Certificate-based authentication for users

Kubernetes provides the option to use client certificates for user authentication. However as there is no way to revoke these certificates when a user leaves an organization or loses their credential, they are not suitable for this purpose.

It should only be used for internal components, and it is configured to work in such a way by default.

It is recommended to proactively monitor the OAuth object to make sure no identity provider works with client certificates.

5.1.1.3. Best practice - extra configuration

The OAuth server generates two tokens for us after a successful login;

Token	Description
Access tokens	Longer-lived tokens that grant access to the API.
Authorize codes	Short-lived tokens whose only use is to be exchanged for an access token.

It is advised to change (shorten) the default “lifespan” of these two [default - 24hrs], to inflict difficulty on an attacker that may obtain one of these - in such a way the attacker will have a harder time keeping a sustained long-duration connection to the platform. For example, if a full-work-day in an organization is considered to be (on average) 9hrs, the duration can be exactly 9hrs.

```
$ oc edit OAuth cluster

# Add the following lines after 'spec:' (including indentation)
# 32400 seconds == 9 hrs
 tokenConfig:
    accessTokenMaxAgeSeconds: 32400
```

It's also advised to configure token inactivity timeout, to revoke unused tokens;

```
$ oc edit OAuth cluster

# Add the following lines after 'spec:' (including indentation)
# 90 minutes == 1.5 hrs
 tokenConfig:
    accessTokenInactivityTimeout: 90m
```

5.1.2. Dedicated Identity Provider for cluster-admins login

It's recommended to separate the login process of the "most privileged" entity (such as cluster admins) in the system to a different, more restricted identity provider's authentication flow, which requires a more "complex" login process (that includes smartcard/OTP, etc.).

This extra effort could be the Delete the kubeadmin user after creating a new identity provider + granting cluster-admin cluster role to relevant usersmain difference between a minor temporary breach in our platform (in case of some project-specific admin user credentials got stolen by an attacker) and a colossal undertaking (in case an attacker compromised a cluster-admin it's game over)

5.1.3. Managing the "kubeadmin" user properly

The kubeadmin is the user we're getting upon finishing installing the cluster initially.

This user is cluster-admin (and statically configured on the platform) by definition because it's also the first (& single) user we have once the cluster is installed properly.

A new cluster admin account should be created and the "kubeadmin" account should be removed.

```
$ oc adm policy add-cluster-role-to-user cluster-admin my-new-admin
```

```
$ oc delete secrets kubeadmin -n kube-system
```

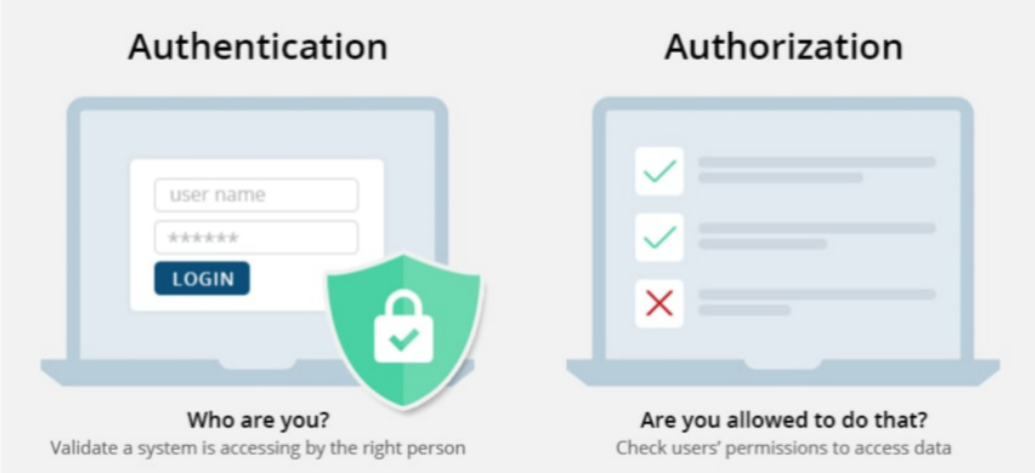
5.1.4. Managing the Kubeconfig file securely

Other than the kubeadmin user, during installation, we also get the “kubeconfig” file. It contains an X.509 client certificate with no expiration date to our cluster.

This file is dedicated to emergencies, unlike the kubeadmin you can’t delete it from the platform itself, and it could be useful for recovery purposes

It is best practice to save it in a dedicated vault (such as HSM) with very limited access to cluster-admins only, auditing & detecting any access to it.

5.2. Authorization



Once a user has been authenticated properly, the system should then determine if a certain request is allowed to be executed or either the user should have a specific set of permissions to perform it.

A user can be assigned to one or more groups, each of which represents a certain set of users. Groups are useful when managing authorization policies to grant permissions to multiple users at once. We can create custom groups or use pre-existing ones such as:

Virtual group	Description
`system:authenticated`	Automatically associated with all authenticated users.
`system:authenticated:oauth`	Automatically associated with all users authenticated with an OAuth access token.
`system:unauthenticated`	Automatically associated with all unauthenticated users.

In OpenShift we also should take into account the entity “Service Account” - which is a “special system user” that is associated with a specific project. It exists to let automated processes perform actions on the OCP securely like any other user, and we can/should limit these entities with an authorization policy like any other user.
E.g. When an application requires the creation/patching/deleting of a sub-service on OpenShift, we would create a service account that will be linked to the main service (with a token linked via Kubernetes secret) to perform the `oc create` functionality that will create the new service.

Note! Service accounts can also be included in groups like any other user.

5.2.1. RBAC

Role-Based Access Control (RBAC) objects to determine whether a user is allowed to perform a given action within a project/the entire cluster.

Restrict access to objects and keys that are stored in the etcd. An entity that has full-access to etcd has cluster-admin permissions.

Some must-glossary on the topic:

Authorization object	Description
Rules	Sets of permitted verbs on a set of objects. For example, whether a user or service account can `create` pods.
Roles	Collections of rules. You can associate, or bind, users and groups to multiple roles.
Bindings	Associations between users and/or groups with a role.

An RBAC "rule" consists of an API resource (Pod/service/...) and a verb (GET/POST/...).
Multiple rules can be gathered to create a "role".
"Subjects" are entities that can be given permissions, for example - users, processes, system accounts, etc. A role contains one or multiple rules, Aa role comprised of the following fields (API resource & verb == rule):

The object associating subjects to roles is called "role binding". A role binding can be valid to a specific project (e.g. user "Joe" is assigned an "administrator" role only in the "Library" project). This kind of binding is called "Local Role Binding" (or just plain "RoleBinding"). A "ClusterRoleBinding" is valid for all projects in the cluster.

Some role definitions can be useful to multiple projects in a cluster. For example - "Project Admin" is a role that is relevant to all projects (although each project has different admins). For the convenience of not redefining the "Project Admin" role for each project, there is a collection of "Cluster Roles" that could be referenced from RoleBindings and ClusterRoleBindings.

Note that a "Cluster Role" does not define the scope of permissions, it can be assigned as a local RoleBinding for project-only permissions, or as a ClusterBinding for cluster-wide permissions.

For more in-depth information on RBAC - Link

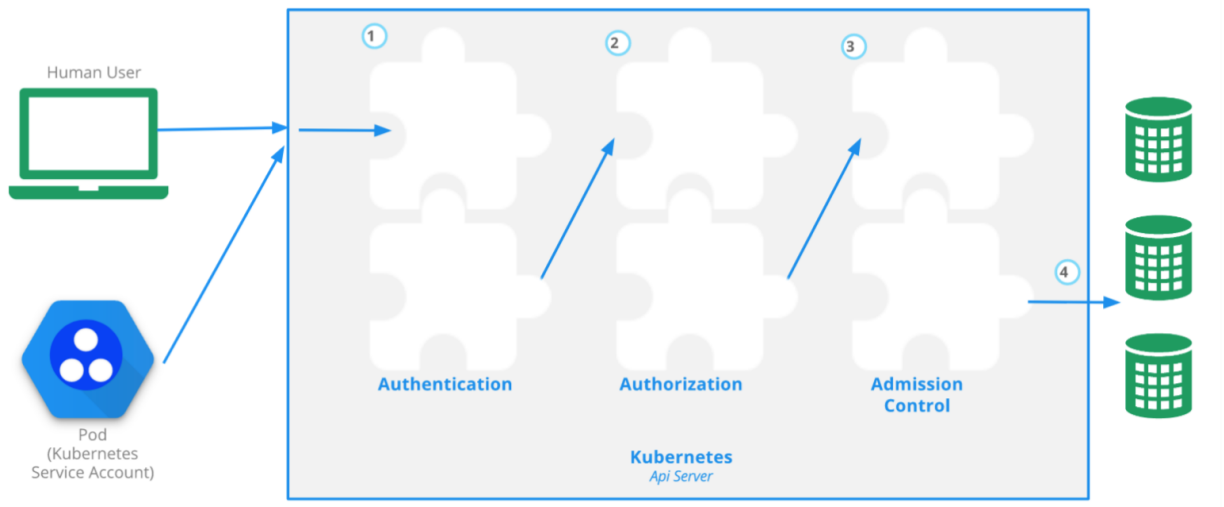
RBAC level	Description
Cluster RBAC	Roles and bindings that are applicable across all projects. *Cluster roles* exist cluster-wide, and *cluster role bindings* can reference only cluster roles.
Local RBAC	Roles and bindings that are scoped to a given project. While *local roles* exist only in a single project, local role bindings can reference **both** cluster and local roles.

The action-validation process is performed by a Kubernetes component which is called **“Admission Controller”,** and it operates based on the RBAC policy and admission plug-ins (built-in and custom plugins). The flow is as follows:

1. Cluster-wide "allow" rules are checked.
2. Locally-bound "allow" rules are checked.

3. Deny by default.

Further information about the admission controller can be found in the following links: 1, 2, 3



Note! Creating RoleBindings / ClusterRoleBindings can be done only by administrators / cluster-administrators respectively.

Default cluster roles

OpenShift Container Platform includes a set of default cluster roles that you can bind to users and groups cluster-wide or locally. You can manually modify the default cluster roles, if required.

Default Cluster Role	Description
`admin`	A project manager. If used in a local binding, an `admin` has rights to view any resource in the project and modify any resource in the project except for quota.
`basic-user`	A user that can get basic information about projects and users.
`cluster-admin`	A super-user that can perform any action in any project. When bound to a user with a local binding, they have full control over quota and every action on every resource in the project.
`cluster-status`	A user that can get basic cluster status information.
`edit`	A user that can modify most objects in a project but does not have the power to view or modify roles or bindings.
`self-provisioner`	A user that can create their own projects.
`view`	A user who cannot make any modifications, but can see most objects in a project. They cannot view or modify roles or bindings.

5.2.1.1. RBAC Best Practices

RBAC In many cases there will be a need for a more specific set of permissions for some entity in the organization - **the advisable way to approach that situation is to duplicate an**

existing restricted Role/ClusterRole and add only the must-required permissions to it, and then obviously attach it to a user/service account and test.

It's also important to mention that attaching the RBAC policy roles to groups is much more correct (from an operative perspective & security-wise) because it's more scalable, and you can make sure that newly-created users are part of an existing group that has restricted RBAC role/clusterole-binding (least privileges perception).

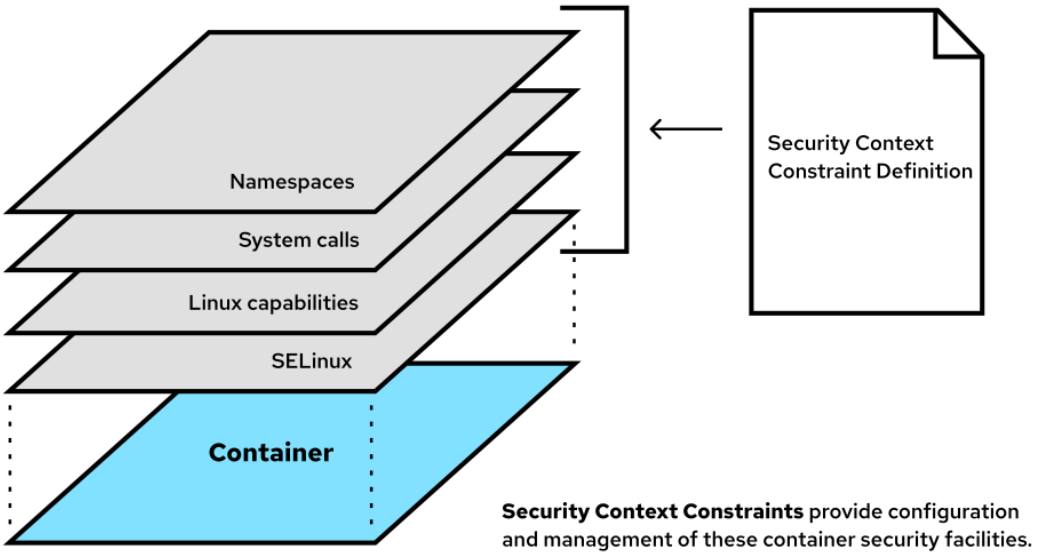
Furthermore, you can sync the groups into OCP using the LDAP sync operator that will do it for you automatically (from a single LDAP source), to get all the benefits of RBAC in the manner that has been mentioned previously. More on that operator can be found in the documentation: https://docs.openshift.com/container-platform/4.6/authentication/ldap-syncing.html

5.2.2. SCC - Security Context Constraints

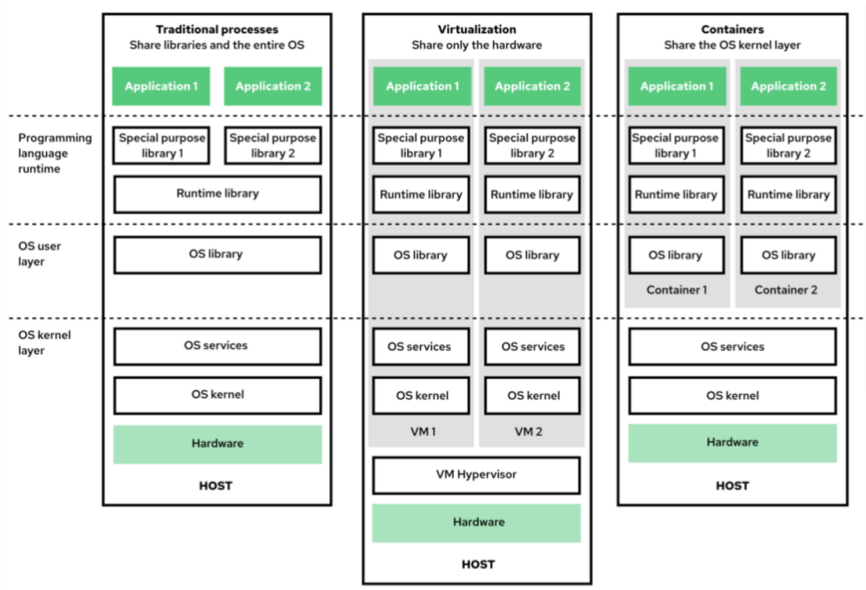
Security Context Constraints are the complementary side of RBAC in the Authorization area in OCP.
RBAC specifies the actions the user can perform on the platform, while the SCC specifies if a certain container/pod is allowed to run on a node in case it has dedicated permissions requirements - mostly related to the Linux & RHCOS infrastructure directly.
Any process that runs on any infrastructure, once that it has been initialized, becomes a potential attack-vector. As such, for the Linux OS, container applications are processed as any other, with the special behavioral property which is the namespace that it's sand-boxed under.

This discipline has tools that are dedicated to making sure that no vulnerable-container is getting exploited - some are responsible for the detection of non-standard behavior and some can prevent such behaviors once they are recognized.

SCC is one such feature that comes out of the box with OpenShift. It makes use of Linux security capabilities (such as Linux capabilities, SElinux Labeling, Seccomps profiles, etc.) and implementing them in a user-friendly way via the OpenShift Platform.



When we run a container (using docker-engine such as CRI-O on OCP 4.x, or Podman/Docker binaries) — the container is treated like any other process by the Host it runs on, with the small distinction which is: a container is an **isolated** (sandboxed) process by Namespaces, SELinux policy, and Cgroups. As such, containers can interact with the host on which they run, much like a “traditional” process, if we would only enable it.



Security context constraints allow administrators to control permissions for pods.

SCCs are objects that define a set of conditions that a pod must run with to be accepted into the system. SCCs allows an administrator to control:

- Whether a pod can run privileged containers.
- The capabilities that a container can request.
- The use of host directories as volumes.
- The SELinux context of the container.
- The container user ID.
- The use of host namespaces and networking.
- The allocation of an FSGroup that owns the pod's volumes.
- The configuration of allowable supplemental groups.
- Whether a container requires the use of a read-only root file system.
- The usage of volume types.
- The configuration of allowable seccomp profiles.

Once we install OpenShift, we are getting with the platform several SCCs out of the box, as an example:
The privileged SCC allows:

- Users to run privileged pods
- Pods to mount host directories as volumes
- Pods to run as any user
- Pods to run with any MCS label
- Pods to use the host's IPC namespace
- Pods to use the host's PID namespace
- Pods to use any FSGroup
- Pods to use any supplemental group
- Pods to use any seccomp profiles
- Pods to request any capabilities

The restricted SCC:

- Ensures that pods cannot run as privileged.
- Ensures that pods cannot mount host directory volumes.
- Requires that a pod run as a user in a pre-allocated range of UIDs.
- Requires that a pod run with a pre-allocated MCS label.
- Allows pods to use any FSGroup.
- Allows pods to use any supplemental group.

On top of that, OpenShift is designed in such a way that no pod will run as privileged by default; It does it (practically) by attaching Restricted SCC policy to any pod by default, and pods that require any elevated privileges (i.e. run with specific UID/GID/MCS Label [SELinux labeling system], add-on Linux capability, access the host's filesystem, change Seccomp profile, mount unique volume types, etc.) need to ask for these permissions from the relevant OCP admin that should be aware enough to decide if it is approved or what other solution can be implemented instead.

Here is an example of such pod's yaml file with the required Linux Capabilities request:

```
apiVersion: v1
kind: Pod
metadata:
  name: security-context-demo-4
spec:
  containers:
  - name: sec-ctx-4
    image: gcr.io/google-samples/node-hello:1.0
    securityContext:
      capabilities:
        add: ["NET_ADMIN", "SYS_TIME"]
```

Without a proper SCC (more elevated than the "restricted" SCC policy) this pod creation request will be prevented by default by the admission controller.

For more detailed information on SCC properties and configuration options, refer to the following links: 1, 2, 3

5.2.2.1. SCC Best Practices

1. **Don't ever grant Privileged SCC to a Service Account**
2. **Don't change the original default SCCs, copy them and deploy the new versions under newly (proactively monitored) SCC**
3. **Monitor for any changes to the default SCCs, especially the 'restricted' SCC. It's a bad habit that managers have to change the default SCC in order to make it easier for them to manage the platform but the side-effect is that over-permissive permissions are allowed by default. Some infrastructure default operators use the**

default SCCs and any change to them may be the cause of some unwanted after-effects.

4. **If you do need some non-restricted SCC, add only the must-have capabilities required, after a security-oriented authority approves this action; Follow the "least-privileges" methodology.**

5. **Do not allow one of the following to be "True": allowHostIPC, allowHostNetwork, allowHostPID, allowHostPorts**

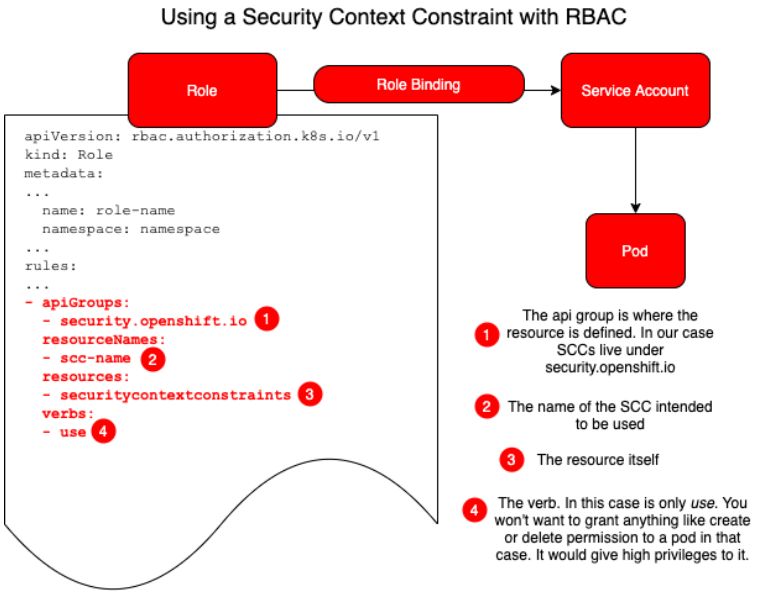
   Any of these will enable an attacker with access to the compromised pod to have access to the host in some way that he/she could elaborate on and expand their hold on our infrastructure

6. **Avoid using RunAsAny in the SCC policies. Instead, if needed, use "MustRunAsRange"/"MustRunAs"**

7. **Avoid changing seccomp profiles in SCC as much as possible.**

5.2.2.2. Integrate SCC with RBAC for easier & automated management

For more automated/organized organizations the preferable method (always) is to work with automation processes as much as possible.



To make our infrastructure-security (using SCCs attached to the Service Accounts in our different projects) in a more generic/modular way — it is recommended to deploy these using

RBAC objects, which are much more “user-friendly” to operate, consistent, sustainable as Yaml files in VSCs (e.g. GitHub, GitLab, etc).

For elaboration on the topic - refer to my medium article: Link

5.3. Managing Service Accounts Properly

5.3.1. Intro

A service account is an OpenShift Container Platform account that allows a component to directly access the API. Service accounts are API objects that exist within each project. Service accounts provide a flexible way to control API access without sharing a regular user’s credentials.

As such, they are in particular targeted by attackers for exploitation. They are not logging in to the cluster as other users by using the OAuth’s chosen identity provider (OIDC for example) but rather with access tokens, generated for each of them by the platform itself.

Furthermore, every pod runs using a service account (and thus under its UID), which means that pods could also mount the service account’s token and use it to authenticate to the API server and perform API calls; It is most common with platform management services but in case of automation services it could be the relevant case with many customers.

An example

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
  serviceAccountName: ngnix-project-x-service-account
```

It is important to mention that these tokens are not rotatable by default by Kubernetes/OpenShift and they do not expire; These properties make the service account token a very attractive target for an attacker that wants stable platform credentials. To mitigate the risk, there is a mechanism that helps us make sure that pods won’t use it directly but either a time-limited token attached to the service account, it is explored in a later section.

Under these circumstances it is quite important to make sure that any such usage of a service account is restricted as possible, and that any misuse of such service account could be easily monitored and lead the forensics team back to the relevant exploited attack vector.

For any new project the platform generates 3 service accounts (same by name for all projects but has different UID and are namespaced);

1. default - By default, if it is not mentioned otherwise, any new deployment is being deployed under this service account.
2. builder - For any action related to the S2i image building processes
3. deployer - For importing images from external registries to the internal OCP registry

5.3.2. Best Practices

5.3.2.1. Separate Service Account for each service

It has already been mentioned that the specific project's "default" service account is the service account that runs all the deployments' pods & containers in a project unless specified otherwise.

It could be problematic if it is being obtained by an attacker because it could be used to exploit the platform or other services that are related to it.

Furthermore, if we have multiple services running under the same service account, and at least one of them is exploitable and is being misused by an attacker to send API calls to the platform - it is audited under this service account, and its source is not explicitly mentioned;

It means that it'll be really difficult to detect what is the attack source - a specific service or even completely external access.

The solution for this issue is quite straightforward:

1. **Generate a dedicated service account for each new deployment/service in our project.**
   This should be further enforced by implementing an admission controller policy that checks if a newly deployed service has a dedicated service account (as shown in the previous example) or whether it relies on the "default" service account (by not mentioning serviceAccountName at all or either requesting specifically the default one). If the latter is true, the admission controller should prevent this deployment from taking place.
2. **Make sure the service account's RBAC & SCC policies are as restricted as possible as discussed in detail in the "SCC" and "RBAC" sections of this document.**

By enforcing these 2 steps we are getting multiple benefits:

1. In case of suspicious behavior - We can audit exactly which service has been compromised.
2. The attack surface is minimized significantly.
3. Granularity - any non-standard service account's permissions (RBAC or SCC or Both) could be generated dedicated to the relevant service account only, without affecting all the others.

5.3.2.2. Restrict Automounting of Service Account Tokens

As previously mentioned, pods could mount the service account's token to access the platform explicitly; It is not so common, and should only be allowed in special cases.

Fortunately, it is easily avoidable by using "automountServiceAccountToken".

It can be in done two ways:

1. Disable all services from using the service account's token; **This is the preferred option in most cases.**

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: example
automountServiceAccountToken: false
```

2. Disable a specific service from using the service account's token;

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  serviceAccountName: example
  automountServiceAccountToken: false
```

5.3.2.3. ExpirationSeconds on ServiceAccountTokens within Pods

The kubelet can also project a service account token into a Pod. You can specify desired properties of the token, such as the audience and the validity duration. These properties are not configurable on the default service account token. The service account token will also become invalid against the API when the Pod or the ServiceAccount is deleted.

The kubelet will request and store the token on behalf of the pod, make the token available to the pod at a configurable file path, and refresh the token as it approaches expiration. The kubelet proactively rotates the token if it is older than 80% of its total TTL, or if the token is older than 24 hours.

The application is responsible for reloading the token when it rotates. Periodic reloading (e.g. once every 5 minutes) is sufficient for most use cases.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - image: nginx
    name: nginx
    volumeMounts:
    - mountPath: /var/run/secrets/tokens
      name: vault-token
  serviceAccountName: example
  volumes:
  - name: vault-token
    projected:
      sources:
      - serviceAccountToken:
          path: vault-token
          expirationSeconds: 7200
          audience: vault
```

Relevant links for deeper explanation: Link1 , Link2

5.4. Etcd protection

5.4.1. Data at rest protection - Etcd Encryption

By default, etcd data is not encrypted in OpenShift Container Platform. You can enable etcd encryption for your cluster to provide an additional layer of data security for sensitive data such as Secrets, ConfigMaps, Routes, OAuth access tokens, OAuth authorize tokens.

This methodology - encrypting data on the storage source that holds it is called “protecting sensitive data at rest”.

Do it by performing the following steps:

```
$ oc edit apiserver
⇒
spec:
  encryption:
    type: aescbc
⇒ save ⇒ verify:

$ oc get openshiftapiserver \
-o=jsonpath='{range
.items[0].status.conditions[?(@.type=="Encrypted")]}{.reason}{"\n"}{.message
}{"\n"}'

$ oc get kubeapiserver \
-o=jsonpath='{range
.items[0].status.conditions[?(@.type=="Encrypted")]}{.reason}{"\n"}{.message
}{"\n"}'
```

Note!! The AES-CBC type means that AES-CBC with PKCS#7 padding and a 32-byte key are used to perform the encryption.

When you enable etcd encryption, encryption keys are created. These keys are rotating weekly.

(Frequent rotation of infrastructure credentials).

5.4.2. Sensitive data transmission in a secured manner

Any communication between the etcd and other cluster components (nodes/masters/apps/operators etc. that requires access to Kubernetes/OpenShift objects) is being done using TLS (both parties have certificates which is being verified for the secure communication to take place, and they both signed by OCP internal-only-accessible CA).

This methodology ensures that only permitted in-cluster components have access to the etcd datastore, and the sensitive data transfer is secured as well.

5.4.3. Backup and monitoring

Usually, on standard clusters, there are at least three instances of etcd, each is getting deployed on a separate master node for redundancy and break-even situations. **Always make sure that all your etcd instances are healthy** because it will have a tremendous effect on your cluster's performance and overall stability.

It is critical to have a valid backup of the etcd at all times, in a remote location, on a dedicated, encrypted storage source.

```
$ oc get pods -n openshift-etcd -o wide ⇒ determine on which nodes the etcds are located
$ oc debug node/<node_name>
sh# chroot /host
sh# /usr/local/bin/cluster-backup.sh /home/core/assets/backup
⇔
Pull/Send it to a secured vault automatically upon change and delete it from the node
```

6. Infrastructure Security

6.1. FIPS Mode - Pre-Installation

Starting with version 4.3, you can install an OpenShift Container Platform cluster that uses FIPS Validated / Modules in Process cryptographic libraries. it's disabled by default.

This process takes place during the installation process and it's very simple to enable that feature - it is been done in the install-config.yaml file that controls the installation configuration; Add the following line to the install-config.yaml file to activate it - `fips: true`

Because FIPS must be enabled before the operating system that your cluster uses boots for the first time, you cannot enable FIPS after you deploy a cluster.

This configuration method at the beginning of the installation also **ensures that new cluster nodes that will be added in the future will have to include that feature or otherwise they will be rejected** by the platform-management operators that were described before, such as MCO, etc.

6.2. Direct access & changes to RHCOS

6.2.1. Monitor for RHCOS Files Changes

For day-to-day management, use only the MCO (Machine config operator - elaborated in the appendix) to make changes on the nodes, and avoid as much as possible from direct access to the nodes (SSH, etc.), except for extreme cases.

Any direct access to the nodes should be monitored and raise an alert, and any changes made by the operator should be allowed to cluster-admins only, using strict RBAC policy, and being audited for any anomalies.

Any new object created in this manner should be audited, and at times, It's advisable to add a compliance check in the admission controller to make sure that newly created objects do not contain too-permissive files permissions (mode) like 777 for example.

```
# This example MachineConfig replaces /etc/chrony.conf
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: example-chrony
spec:
  config:
    ignition:
      version: 2.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-8;base64,<BASE64_DATA>
        filesystem: root
        mode: 0644
        path: /etc/chrony.conf
```

Any change to system files on RHCOS should be monitored, audited, and prevent permissions' change. It could & should be achieved by implementing standard work with the file integrity operator and Compliance Operator (Openscap for OCP).

6.2.2. Direct access to nodes

There is a command available through the platform, that allows access to the nodes over the platform itself.

That command goes as follows:

```
$ oc debug node/<node>
```

By making it the standard option to access out nodes, that command we're getting multiple security achievements at once;

1. Only platform-authenticated parties are even accessible to the `oc` API. (authentication)
2. We can make sure only the relevant entities are using that command by implementing a restricting RBAC policy on our platform (authorization).
3. By using that OCP command we can audit everything about it just like we audit every other command running on our platform such as: who ran it, when, from where, etc. (auditing)

```
$ oc adm node-logs --role master --path=/kube-apiserver/audit.log | grep -i "-debug"
```

4. We can pop an alert for every direct SSH connection to the nodes because with the command in place there's not any reason to use it (excluding the obvious exceptions); By doing so, we're keeping any "random" persona in our network as far as we can from our infrastructure, and can interrupt any such attempt preemptively/prematurely.

During installation, an ssh-login-dedicated user is getting created. It is called “core” and its ssh-key is already copied to all cluster nodes. It is part of the “wheel” group, which means that it has sudo privileges on the nodes. So to expand on the last point from the last paragraph - **monitor for any login made by the core user, and also monitor for any additional members to the “wheel” group or either new entries that are being added to the “sudoers” file on the nodes.**

The second point could be achieved easily by using the File Integrity Operator.

6.3. Resources Exhaustion Prevention

6.3.1. Glossary

- **Pods’ Requests & Limits**
  As a developer, you can set *requests* and *limits* on compute resources at the pod and container levels.
  - *requests* - minimum amount of resources required for the pod/object (service, deployments, etc) to function properly
  - *limits* - the maximum threshold that is requestable by an object/set of objects in a project

- **Resource Quota**
  Defines the max amount of objects’ instances that can be created in a project (by type), as well as the total amount of compute resources that may be consumed by resources in that project. (in terms of CPU, Memory & Storage).

  A quota specifies hard resource usage limits for a project. All attributes of a quota are optional, meaning that **any resource that is not restricted by a quota can be consumed without bounds.**

  If project modifications exceed the quota usage limit, the server denies the action.

Table 1. Compute resources managed by quota

Resource Name	Description
`cpu`	The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. `cpu` and `requests.cpu` are the same value and can be used interchangeably.
`memory`	The sum of memory requests across all pods in a non-terminal state cannot exceed this value. `memory` and `requests.memory` are the same value and can be used interchangeably.
`ephemeral-storage`	The sum of local ephemeral storage requests across all pods in a non-terminal state cannot exceed this value. `ephemeral-storage` and `requests.ephemeral-storage` are the same value and can be used interchangeably. This resource is available only if you enabled the ephemeral storage technology preview. This feature is disabled by default.
`requests.cpu`	The sum of CPU requests across all pods in a non-terminal state cannot exceed this value. `cpu` and `requests.cpu` are the same value and can be used interchangeably.
`requests.memory`	The sum of memory requests across all pods in a non-terminal state cannot exceed this value. `memory` and `requests.memory` are the same value and can be used interchangeably.
`requests.ephemeral-storage`	The sum of ephemeral storage requests across all pods in a non-terminal state cannot exceed this value. `ephemeral-storage` and `requests.ephemeral-storage` are the same value and can be used interchangeably. This resource is available only if you enabled the ephemeral storage technology preview. This feature is disabled by default.
`limits.cpu`	The sum of CPU limits across all pods in a non-terminal state cannot exceed this value.
`limits.memory`	The sum of memory limits across all pods in a non-terminal state cannot exceed this value.
`limits.ephemeral-storage`	The sum of ephemeral storage limits across all pods in a non-terminal state cannot exceed this value. This resource is available only if you enabled the ephemeral storage technology preview. This feature is disabled by default.

Table 2. Storage resources managed by quota

Resource Name	Description
`requests.storage`	The sum of storage requests across all persistent volume claims in any state cannot exceed this value.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the project.
`<storage-class-name>.storageclass.storage.k8s.io/requests.storage`	The sum of storage requests across all persistent volume claims in any state that have a matching storage class, cannot exceed this value.
`<storage-class-name>.storageclass.storage.k8s.io/persistentvolumeclaims`	The total number of persistent volume claims with a matching storage class that can exist in the project.

Note! Table 1 “ephemeral storage” refers to temporary storage types like “emptyDir”, table 2 refers to persistent storage options

Table 3. Object counts managed by quota

Resource Name	Description
`pods`	The total number of pods in a non-terminal state that can exist in the project.
`replicationcontrollers`	The total number of ReplicationControllers that can exist in the project.
`resourcequotas`	The total number of resource quotas that can exist in the project.
`services`	The total number of services that can exist in the project.
`services.loadbalancers`	The total number of services of type `LoadBalancer` that can exist in the project.
`services.nodeports`	The total number of services of type `NodePort` that can exist in the project.
`secrets`	The total number of secrets that can exist in the project.
`configmaps`	The total number of `ConfigMap` objects that can exist in the project.
`persistentvolumeclaims`	The total number of persistent volume claims that can exist in the project.
`openshift.io/imagestreams`	The total number of imagestreams that can exist in the project.

Note! Each property is being enforced separately from the others

```
$ oc create quota dev-quota --hard pod=10,requests.cpu=1,memory=1.5Gi,limits.cpu=2
```

```yaml
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-quota
spec:
  hard:
    pods: "10"
    requests.cpu: "1"
    memory: "1.5Gi"
    limits.cpu: "2"
```

Note! Although a single quota resource can define all of the quotas for a project, a project can also contain multiple quotas

- **Limit Range**

defines the default, minimum, and maximum values for compute resource requests, and the limits for a single pod or container defined inside the project.

```yaml
apiVersion: "v1"
kind: "LimitRange"
metadata:
  name: "dev-limits"
spec:
  limits:
    - type: "Pod"
      max:
        cpu: "500m"
        memory: "750Mi"
      min:
        cpu: "10m"
        memory: "5Mi"
    - type: "PersistentVolumeClaim"
      min:
        storage: "1Gi"
      max:
        storage: "50Gi"
```

```
[user@demo ~]$ oc describe limitrange dev-limits
Name:       dev-limits
Namespace:  schedule-demo
Type                        Resource                Min   Max    Default Request ...
----                        --------                ---   ---    --------------- ...
Pod                         cpu                     10m   500m   -               ...
Pod                         memory                  5Mi   750Mi  -               ...
Container                   memory                  5Mi   750Mi  20Mi            ...
Container                   cpu                     10m   500m   20m             ...
openshift.io/Image          storage                 -     1Gi    -               ...
openshift.io/ImageStream    openshift.io/image-tags -     10     -               ...
openshift.io/ImageStream    openshift.io/images     -     20     -               ...
PersistentVolumeClaim       storage                 1Gi   50Gi   -               ...
```

- **Cluster Quota**
  Similar to quota, but is being defined in a non-namespace scope (cluster-scope) and can be “attached” to multiple namespaces simultaneously.

  Cluster administrators can specify which projects are subject to cluster resource quotas in two ways:

  - Using the *openshift.io/requester* annotation to specify the project owner. All projects with the specified owner are subject to the quota.
  - Using a *selector* property within the cluster quota object. All projects whose labels match the selector are subject to the quota.

```
# Option #1:
$ oc create clusterquota exmple_cq \
--project-annotation-selector openshift.io/requester=qa \
--hard pods=12,secrets=20
# Option #2
$ oc create clusterquota exmple_cq_2 \
--project-label-selector label=qa \
--hard pods=10,services=5
```

6.3.2. The security perspective

Why is it so important? Because any resource that is not restricted by a quota can be consumed without bounds; Additionally, OpenShift & Kubernetes have shared environments by definition, which means that our resources are shared as well and that one exploitable service whose not restricted properly could “starve” our entire cluster.

Furthermore, imposing a quota on the number of Kubernetes resources improves the stability of the OpenShift control plane by avoiding unbounded growth of the Etcd database. Quotas on Kubernetes resources also avoid exhausting other limited software resources, such as IP addresses for services.

By using Quotas and LimitRanges, cluster administrators can set constraints to limit the number of objects or the amount of compute resources that are used in a project. This helps cluster administrators ensure that no projects are using more than is appropriate for the cluster size.

For proper management, it is recommended that Resource request and resource limits will be configured for each container in either deployment or a deployment configuration resource; They should be properly adjusted according to the project’s/cluster’s quota. It could be achieved easily by deploying a standard, compiled LimitRanges object for every project in the cluster.

All resource creation and modification requests are evaluated against each LimitRanges object in the project. If the resource violates any of the enumerated constraints (including the minimum requirements, if it is defined), then the resource is rejected. If the resource does not set an explicit value, and if the constraint supports a default value, then the default value is applied to the resource.

```
# Option 1:
$ oc set resources deployment hello-world-nginx \
--requests cpu=10m,memory=20Mi --limits cpu=80m,memory=100Mi

# Option 2:
edit the deployment/deploymentconfig YAML file to include those properties
```

Create the LimitRanges & Quota in every project as a non-issue, OpenShift provides a "default project template" that could be edited to contain such objects out-of-the-box for each newly created project.

```
# Phase 1: Create, Edit & Deploy the new template
$ oc adm create-bootstrap-project-template -o yaml > template.yaml

# Edit the template.yaml file to include the LimitRanges object, and the appropriate for you
environment quota object/s

$ oc create -f template.yaml -n openshift-config

# Phase 2: Notify the cluster to use the new project by default
$ oc edit project.config.openshift.io/cluster

apiVersion: config.openshift.io/v1
kind: Project
metadata:
  ...
spec:
  projectRequestTemplate:
    name: <template_name>

# Phase 3: Testing
$ oc new-project <project_name>
$ oc get resourcequotas -n <project_name>
$ oc get limitranges -n <project_name>
```

Finally, There's a crucial aspect around the decision who is allowed to create new projects within our platform; Limiting the amount of resources per project is useless unless we are limiting the number of projects a user can create (in case it has "self-provisioning"), or limit the number of users that can provision projects; Either allowing it only to cluster admins (which could be problematic in case of the vast amount of projects required) or to a dedicated group with a controlled amount of users.

By default "self-provisioning" permissions are granted to all authenticated users.

The easiest solution: Remove the default "self-provisioner" permission & Prevent it from being auto-updated, and attach it to the relevant users/custom group only. (Link)

```
$ oc adm policy \
    remove-cluster-role-from-group self-provisioner \
    system:authenticated:oauth
```

```
$ oc patch clusterrolebinding.rbac self-provisioners -p '{ "metadata": { "annotations": {
"rbac.authorization.kubernetes.io/autoupdate": "false" } } }'
```

6.4. Secure Storage Management

6.4.1 Persistent Volumes vs. StorageClass

The OpenShift Container Platform persistent volume framework allows administrators to provision a cluster with persistent storage.
The framework also gives users a way to request those resources without having any knowledge of the underlying infrastructure.

Many storage types are available for use as persistent volumes in the OpenShift Container Platform. All of them can be statically provisioned by an administrator (Creating Persistent Volume Object with details about the storage server itself), but it is also possible and very common to enable dynamic storage provisioning using StorageClass.

The StorageClass resource object describes and classifies storage that can be requested, as well as provides a means for passing parameters for dynamically provisioned storage on demand.

StorageClass objects can also serve as a management mechanism for controlling different levels of storage and access to the storage. Cluster Administrators (cluster-admin) or Storage Administrators (storage-admin) define and create the StorageClass objects that users can request without needing any detailed knowledge about the underlying storage volume sources, unlike persistent volume objects.

Security-wise it’s important to make the distinction between Persistent Volumes and StorageClass when it comes to access by standard users.

A standard user that needs storage for its application/service is also required to know what storage options are accessible (what storage types the administrators attached to the platform), to ask for them using Persistent Volume Claim Object & mount it to its pods.

By exposing Persistent Volume objects directly to the user, the developer can see everything about the storage backend itself and how OpenShift is configured to access it. This is a tremendously problematic approach because it exposes the storage access points to a potential attacker that has standard access to the OpenShift Platform.

Unlike exposing “vanilla” Persistent volumes, StorageClasses are much more user-oriented. Without getting into too much detail, “StorageClass” is a suite solution, built-in a multi-layer

manner; It enables the cluster administrator to display only the StorageClass Object itself without including everything running in the background, such as the sensitive details on connecting to the storage provider itself, among other things.

Needless to say that standard users should not be able to create their Persistent Volumes, it is possible with the right RBAC permissions, but it is recommended not to grant this capability.

A user that can mount into the OpenShift platform and the pods running on top of it whatever he/she wants (which is exactly the situation when a user can create a custom PV from an unknown source, with potentially pre-created malicious data, and bind it to a pod), is a risk to the environment whole environment, and only cluster-admins/storage-admins should be able to perform that action; it is enforceable by RBAC permissions ⇒ “create” method ⇒ Persistent Volume object - make sure only admins have it, and standard does not.

For more information on PVs, StorageClasses and PVCs use the attached links.

6.4.2 Ephemeral Storage Quota

Files in a container are ephemeral. As such, when a container crashes or stops, the data is lost. You can use volumes to persist the data used by the containers in a pod. A volume is a directory, accessible to the Containers in a pod, where data is stored for the life of the pod.
,
Although you can use persistent storage, it’s only relevant when the data is important, and it’s not always the case - i.e. a pod with a container that only performs a mathematical operation and returns a response to the external requester, it does not need this data available again, and therefore “ephemeral” storage is the suitable solution.

“emptyDir” is a crucial Kubernetes volume type that enables pods to run using ephemeral storage only, without the need of declaring & mounting PVCs to a persistent volume. It’s extremely common, and so it can be unexpected to think of it as a potential cause of denial of service situation on an OpenShift node, but it is a valid possibility that should be taken care of;

An emptyDir volume is first created when a Pod is assigned to a node and exists as long as that Pod is running on that node. As the name says, the emptyDir volume is initially empty. All containers in the Pod can read and write the same files in the emptyDir volume. When a Pod is removed from a node for any reason, the data in the emptyDir is deleted permanently.

If we take a look at this feature it may seem unharmful, but when considering the possibility of a pod with multiple containers that keeps on writing and saving files on that emptyDir without rotating, it means that they are consuming disk space from the hosting node.

The scenario of local storage exhaustion is possible, but thankfully there is a simple solution that can prevent this situation from happening entirely.

There are two dedicated quota optional fields that **you should define by default in your project template to mitigate that risk completely:**

Resource Name	Description
`requests.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage requests cannot exceed this value.
`limits.ephemeral-storage`	Across all pods in the namespace, the sum of local ephemeral storage limits cannot exceed this value.
`ephemeral-storage`	Same as `requests.ephemeral-storage`.

More on emptyDir and ephemeral quota options can be found here:
https://kubernetes.io/docs/concepts/storage/volumes/#emptydir
https://v1-19.docs.kubernetes.io/docs/concepts/policy/resource-quotas/

6.4.3 Security Summary

StorageClass is the preferred method of exposing volumes to users for many reasons; Not only that StorageClass is a storage dynamic provisioning solution, which can save the cluster-administrators a lot of unnecessary, manual work - it can also be classified as the preferred solution for security reasons.

- **In your default RBAC policy - disable user access to PVs (specifically "get" method, the "list" is more acceptable because it may be necessary to make sure that storage has been bound to a PVC correctly - and it does not expose extra-unneeded-sensitive data)**
- **Add dynamic storage provisioning solutions with StorageClasses and grant users Read-Only access to use them properly**
- **Remove the option for standard users to create their own PVs (RBAC)**
- **Limit the storage amount that a single project/service can request using default LimitRange and Quotas in a project template as specified in the *"Resource Exhaustion Prevention"* section**
- **Limit usage of ephemeral storage to avoid node storage exhaustion by implementing a default storage quota that includes "*requests.ephemeral-storage*" and "*limits.ephemeral-storage*"**

6.5. Trusted Images Sources

Red Hat OpenShift includes a private registry that provides basic functionality to manage your container images. The Red Hat OpenShift Platform provides a role-based access control (RBAC) mechanism that allows you to manage who can pull and push specific container images, and from/to which external image sources/registries he/she can do it.

6.5.1. Best Practices - External Registry

Use a trusted external registry as the only permitted source of standard base images for the organization. This registry should be maintained, updated, monitored, and protected from tampering using RBAC. This practice reduces the risk of supply chain attacks and enforces the use of secure, updated base images.

Avoid using the internal registry as the main registry for the network.

6.5.2. Best Practices - Verify external image sources

Verifying the external image sources on the cluster is crucial for the platform and infrastructure security. An attacker with access to the platform that can deploy images from wherever he/she wants - would be able to run exploitable containers that would grant him/her a comfortable attack surface for future attacks and lateral movement.

OpenShift provides us with an easy way to specify allowed image sources. it has several steps involved;

1. “Image config” object, editable from within OCP
    a. Property: allowedRegistriesForImport + additionalTrustCA
    b. Property: allowedRegistries - Relevant for Build processes
    c. insecure Registries - Should not exist at all and should be monitored by the cluster-administrators that this property doesn’t exist on the cluster.
    d. For disconnected environments - add “blockedRegistries” property with the URLs of well-known public registries

6.5.2.1. Whitelist of Allowed Registries

```
$ oc edit image.config.openshift.io/cluster
```

```yaml
apiVersion: config.openshift.io/v1
kind: Image
spec:
  allowedRegistriesForImport:
    - domainName: example.com
      insecure: false (Crucial!!!)
  additionalTrustedCA:
    name: myconfigmap
  registrySources:
    allowedRegistries:
    - my.registry.com
    - my.other.registry.com
    - a.public.registry.if.needed
    insecureRegistries:
    - insecure.com (Should not be allowed)
status:
  internalRegistryHostname: image-registry.openshift-image-registry.svc:5000
```

Property: "AllowedRegistriesForImport"

As the name suggests, this property refers to registries that our users (developers, etc.) are going to fetch/import images from (Pull Operations).

Property: "AdditionalTrustCA"

This one is relevant in the context of secured access to the external registries;

For any uniquely signed external registry, that we would like to allow access to—we need to include its certificate bundle in our ConfigMap (it could be named whoever we want, just include the name in the property). This ConfigMap holds the certificates for all of our allowed registries.

```
$ oc create configmap myconfigmap --from-file=<registry_address>=my_registires_ca_bundle.crt
-n openshift-config
```

Property: "AllowedRegistries"

Allow listed for image pull and push actions. All other registries are blocked.

6.5.2.2. Prevent insecureRegistries

Note!! There's an option to add "insecureRegistries" property - it is not recommended to configure any such registries (security-wise) - Monitor for any such configuration in the object.

Another good alternative is to develop a custom admission controller webhook that blocks the deployment of this object if the "insecureRegistries" property exists in the YAML file.

6.5.2.3. Block Specific Registries

Most Relevant for disconnected environments. It is advised to add a blacklist of the well-known public image sources (such as quay.io, registry.access.redhat.com, etc.);

It is mentioned because in case you're allowing those registries, you could end up allowing an attacker to spoof its registry as one of these options and by that deploy everything he/she wants on the platform (under the assumption that he/she already got a user with access).

```
spec:
  registrySources:
    blockedRegistries:
    - untrusted.com
```

6.5.3. Sign images - GPG Keys

Verifying the external registries URLs & certificates is a fine security add-on but to take our security efforts an extra step forward - signing the images themselves is a much more demanding process but is also a crucial one.

This process includes two parts;

Part 1:

1. Build our application image locally OR Pull an existing non-signed-image (e.g. base image for s2i future builds) from a trusted source (quay.io for example) locally

```
$ podman pull...
OR
$ podman build...
```

2. Generate GPG Key

```
$ gpg2 --gen-key
...
Real name: username
Email address: username@domain.com
You selected this USER-ID:
    "username <username@domain.com>"

Change (N)ame, (E)mail, or (O)kay/(Q)uit? O
...
pub   rsa2048 2020-12-05 [SC] [expires: 2022-12-05]
      ID_XXXXXXXXX
uid                      username <username@domain.com>
sub   rsa2048 2020-12-05 [E] [expires: 2022-12-05]

# find ID from last output
$ gpg2 --edit-key ID_XXXXXXXXX trust
  1 = I don't know or won't say
  2 = I do NOT trust
  3 = I trust marginally
  4 = I trust fully
  5 = I trust ultimately
  m = back to the main menu

Your decision? 5

Do you really want to set this key to ultimate trust? (y/N) y
...
$ gpg> quit
```

3. Sign the desired Image
4. Push the signed image to our external & trusted registry (e.g. quay/harbor etc.)

3+4:

```
$ podman push <image> --sign-by username@domain.com
```

5. Expose Sigstore on a separated secured service (e.g. HTTPS) for gpg-key verification by OCP, also export the public key, and copy it for the next steps related to OCP.

```
$ mkdir -p /etc/pki/developers/
$ gpg2 --armor --export username@domain.com > /etc/pki/developers/signer-key.pub
```

Note! It's up to the administrator to choose the way of exposing the SigStore; It can be done simply by copying the public key file to an HTTPS server under /var/www/html.

Note! The exposed SigStore should be protected using file integrity validation mechanisms

- Let's say for the rest of the example it is exposed under https://mywebserver/sigstore/

Part 2:

6. Configure *"imge.config.openshift.io/cluster"* object ⇒ under "allowedRegistriesForImport" - add the URL of our external registry [my.registry]

```
$ oc edit image.config.openshift.io/cluster

# YAML
kind: Image
metadata:
  annotations:
  name: cluster
spec:
  allowedRegistriesForImport:
    - domainName: my.registry
```

7. Create policy.json + my.registry.yaml file that specifies the gpg & sigstore parameters for OCP

```
$ cat > policy.json <<EOF
{
  "default": [
    {
      "type": "insecureAcceptAnything"
    }
  ],
  "transports": {
    "docker": {
      "my.registry": [
        {
            "type": "signedBy",
            "keyType": "GPGKeys",
            "keyPath": "/etc/pki/developers/signer-key.pub"
        }
      ]
    },
    "docker-daemon": {
      "": [
        {
          "type": "insecureAcceptAnything"
        }
      ]
    }
  }
}
EOF

$ cat <<EOF > my.registry.yaml
docker:
     my.registry:
         sigstore: https://mywebserver/sigstore/
EOF
```

Also export some encoded base64 parameters for future use

```
$ export DOCKER_REG=$( cat my.registry.yaml | base64 -w0 )
$ export SIGNER_KEY=$(cat /etc/pki/developers/signer-key.pub | base64 -w0 )
$ export POLICY_CONFIG=$( cat policy.json | base64 -w0 )
```

8. Create & apply a “MachineConfig” object file that is being monitored and acted upon by the MCO (Machine Config Operator)

```
$ cat > worker-custom-registry-trust.yaml <<EOF
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: worker-custom-registry-trust
spec:
  config:
    ...
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-8;base64,${DOCKER_REG}
          verification: {}
        filesystem: root
        mode: 420
        path: /etc/containers/registries.d/my.registry.yaml
      - contents:
          source: data:text/plain;charset=utf-8;base64,${POLICY_CONFIG}
          verification: {}
        filesystem: root
        mode: 420
        path: /etc/containers/policy.json
      - contents:
          source: data:text/plain;charset=utf-8;base64,${SIGNER_KEY}
          verification: {}
        filesystem: root
        mode: 420
        path: /etc/pki/developers/signer-key.pub
  osImageURL: ""
EOF
```

Notice how we used our exported variables to deliver the files' content easily.

```
$ oc apply -f worker-custom-registry-trust.yaml
```

Diagram Part 1:

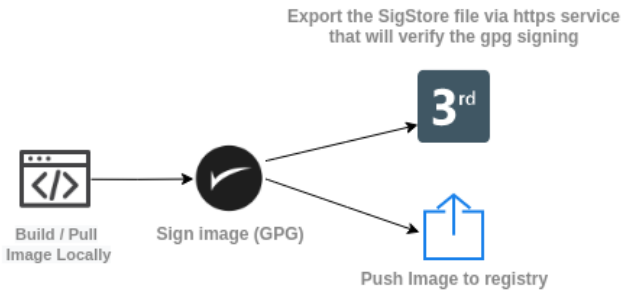


Diagram Part 2:

6.6. Security Consideration with S2i

There's an option in OCP to build & deploy applications directly from within the platform. This approach is intended for developers to "skip" the standard local build process, and it is very common with multiple customers - and even more, since Tekton got into the picture as a cloud-native CI/CD pipeline tool.

But from a security perspective, it can be a tricky issue to solve. How do we ensure that each image built in such a way is being scanned before it gets saved in our internal registry? (it's the default behavior of S2i)

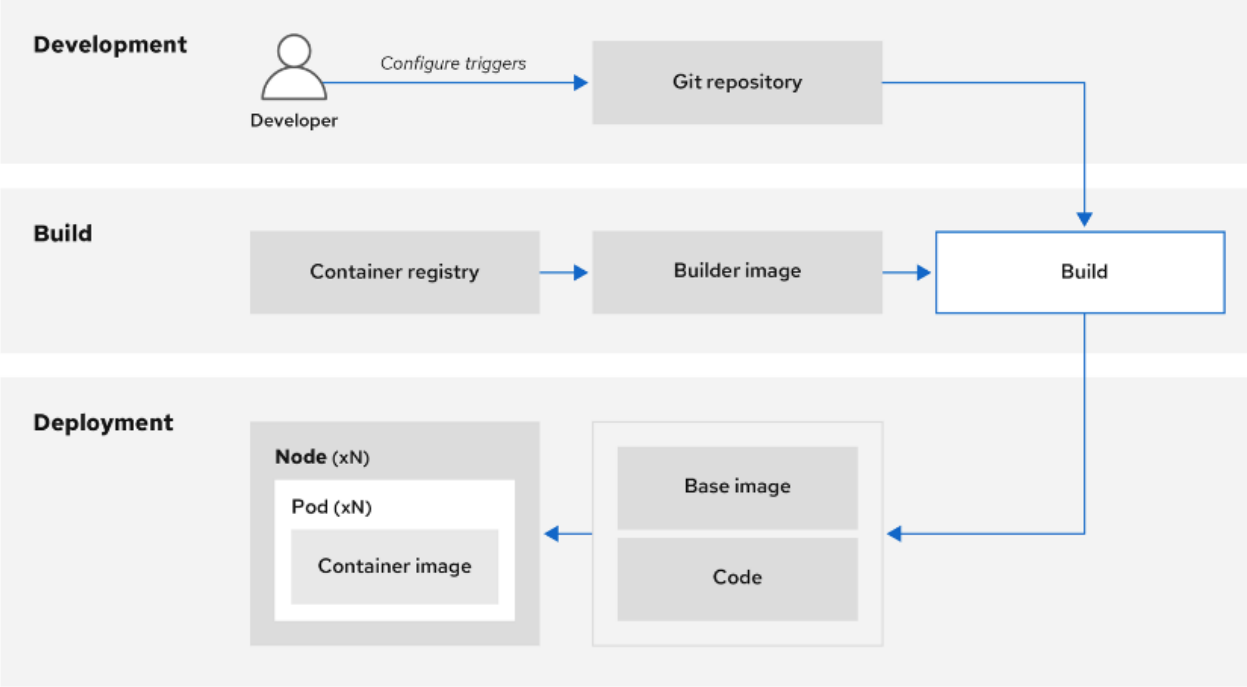
4 Main S2i options:

1. Source build - directly pulling the code from VSC and build it using an existing base-image within the internal registry
2. Docker-based upon Dockerfile build
3. JenkinsPipeline
4. Custom

Furthermore, **Builds in OpenShift Container Platform are run in privileged containers. Depending on the build strategy used, if you have privileges, you can run builds to escalate their permissions on the cluster and host nodes. And as a security measure, it limits who can run builds and the strategy that is used for those builds. Custom builds are inherently less safe than source builds, because they can execute any code within a privileged container, and are disabled by default.**

By default, all users that can create builds are granted permission to use the docker and **Source-to-image** (S2I) **build strategies**. Users with cluster administrator privileges can enable the custom build strategy, as referenced in the restricting build strategies to a user globally section.

You can control who can build and which build strategies they can use by using an authorization policy. Each build strategy has a corresponding build subresource. A user must have permission to create a build and permission to create on the build strategy subresource to create builds using that strategy. Default roles are provided that grant the create permission on the build strategy subresource.



OpenShift administrators can disable that option completely on the platform, thus the developers will be required to perform the build process externally - it will end with a signed image in an external registry where the image will be scanned, and only then with a strict image-source policy it will be pulled by a dedicated service account into the platform.

It can be easily achieved by deleting the following clusterrolebinding:

```
$ oc adm policy remove-cluster-role-from-group system:build-strategy-docker
system:authenticated
$ oc adm policy remove-cluster-role-from-group system:build-strategy-source
system:authenticated
$ oc adm policy remove-cluster-role-from-group system:build-strategy-custom
system:authenticated
$ oc adm policy remove-cluster-role-from-group system:build-strategy-jenkinspipline
system:authenticated
```

Also, for each existing standard role, remove the non-required strategy from the ClusterRole:

```
$ oc edit clusterrole admin
```

```
kind: ClusterRole
metadata:
  name: admin
...
rules:
- resources:
  < all strategies are deleted >
  ...
...
```

https://docs.openshift.com/container-platform/4.6/builds/securing-builds-by-strategy.html

6.7. Observability / Visibility - Logging, Monitoring & Auditing

OpenShift is getting installed with logging, monitoring, and auditing built-in mechanisms, and it also can send alerts when it's configured to detect abnormalities.

But there is a joint issue with these default mechanisms which is - the grand amount of alerts/logs are not security oriented by design, which means it's very limited for proper attack detection/investigations. It's non-trivial to recognize a security-helpful log and isolate/tag it as relevant from the main bulk of platform/infrastructure-error-logs pool.

There are multiple tools designed to do just that - take advantage of the default logging/auditing/monitoring mechanisms, fetch relevant data from the Kubernetes and OpenShift engines, make the security-oriented correlations and send a proper alert that holds the relevant data that points at specific attack attempt or suspicious behavior in the environment.

It is highly recommended to integrate with such a 3rd party monitoring system.

3rd party observability products are out of the scope of that document, but it is an important topic and there are measures that one could take with the existing "vanilla" OpenShift Platform to gain some form of visibility.

6.7.1. Basic Audit logs gathering & analysis

You can view logs for the OpenShift Container Platform API server or the Kubernetes API server for each master node.

```
# OpenShift Logs
$ oc adm node-logs --role=master --path=openshift-apiserver/
$ oc adm node-logs <node-name> --path=openshift-apiserver/<log-name>

# Kubernetes Logs
$ oc adm node-logs --role=master --path=kube-apiserver/
$ oc adm node-logs <node-name> --path=kube-apiserver/<log-name>
```

Output example [redundant] for a single log;

Several things to notice:

- username, uid, and groups of the user which performed the API call to the OCP API server
- description of the operation (creation of ClusterRoleBinding of the "Cluster Admin" role to

a Certain Service account

- final admission controller's decision regarding the operation (= "allow" in that case)

```
{
  "kind": "Event",
...
  "verb": "update",
  "user": {
    "username":
"system:serviceaccount:openshift-kube-controller-manager:localhost-recovery-client",
    "uid": "dd4997e3-d565-4e37-80f8-7fc122ccd785",
    "groups": [
      "system:serviceaccounts",
      "system:serviceaccounts:openshift-kube-controller-manager",
      "system:authenticated"
    ]
  },
...
  "annotations": {
    "authorization.k8s.io/decision": "allow",
    "authorization.k8s.io/reason": "RBAC: allowed by ClusterRoleBinding
\"system:openshift:operator:kube-controller-manager-recovery\" of ClusterRole
\"cluster-admin\" to ServiceAccount
\"localhost-recovery-client/openshift-kube-controller-manager\""
  }
}
```

The default log policy profile Logs only metadata for reading and writing requests; does not log request bodies. It can be configured by changing the policy profile to "WriteRequestBodies" or "AllRequestBodies" the specific added data on each can be found here: https://docs.openshift.com/container-platform/4.6/security/audit-log-policy-config.html#about-audit-log-profiles_audit-log-policy-config

It is recommended to change to at least "WriteRequestBodies" to get a better state of our environment & operations and to be able to troubleshoot events (cyber or of other types) properly.

6.7.2. Cluster logging

As a cluster administrator, you can deploy cluster logging (Installation using Operator) to aggregate all the logs from your OpenShift Container Platform cluster, such as node system

audit logs, application container logs, and infrastructure logs. Cluster logging aggregates these logs from throughout your cluster and stores them in a default log store. You can use the Kibana web console to visualize log data.

Cluster logging aggregates the following types of logs:

- **application** - Container logs generated by user applications running in the cluster
- **infrastructure** - Logs generated by infrastructure components running in the cluster and OpenShift Container Platform nodes, such as journal logs. Infrastructure components are pods that run in the openshift*, kube*, or default projects.
- **audit** - Logs generated by the node audit system (auditd), which are stored in the /var/log/audit/audit.log file, and the audit logs from the Kubernetes apiserver and the OpenShift apiserver.

The major components of cluster logging are:

- **collection** - This is the component that collects logs from the cluster, formats them, and forwards them to the log store. The current implementation is Fluentd. By default, the log collector uses the following sources:
  - journald for all system logs
  - /var/log/containers/*.log for all container logs
- **log store** - This is where the logs are stored. The default implementation is Elasticsearch. You can use the default Elasticsearch log store or forward logs to external log stores. The default log store is optimized and tested for short-term storage.
- **visualization** - This is the UI component you can use to view logs, graphs, charts, and so forth. The current implementation is Kibana.

The ClusterLogging CR defines a complete cluster logging environment that includes all the components of the logging stack to collect, store and visualize logs.

By default, the OpenShift Container Platform uses Elasticsearch (ES) to store log data. Optionally, you can use the log forwarding features to forward logs to external log stores using Fluentd protocols, syslog protocols, or the OpenShift Container Platform Log Forwarding API. See the relevant documentation here:
https://docs.openshift.com/container-platform/4.6/logging/cluster-logging-external.html

Make sure all redirected logs are getting sent securely only (configurable in the "ClusterLogForwarder" CR).

6.7.3. Security-relevant logs to monitor

Multiple logs could be useful when it comes to security;

All of the following options (combined) are important for minimum security visibility, and complementary products such as Service Mesh, 3Scale, etc. are crucial for a complete, proper observability cover.

As mentioned earlier, this entire section is an overview only, there are multiple products dedicated to deal with this issue properly, but they also should be fine-tuned and handle at least the following disciplines.

Day-to-Day Platform & Infrastructure Management related logs:

- Everything considering the cluster state and the logging tools' health status. Examples:
  - https://docs.openshift.com/container-platform/4.6/logging/troubleshooting/cluster-logging-alerts.html
  - https://docs.openshift.com/container-platform/4.6/nodes/nodes/nodes-nodes-machine-config-daemon-metrics.html

- Quotas / LimitRanges related failure (e.g. request too much storage than the approved amount)

- Platform & infrastructure events. Examples:
  - negative Node Events (e.g. FailedMount)
  - Suspicious container events (e.g. BackOff because image source is non-approved one/image isn't signed with GPG)

- Infrastructure & Platform Operators CRs changes as specified in the upper sections

Networking Logs

- Every SSH direct access to the nodes

- NetworkPolicy connections rejections
  - North-South communication
    - ➢ internal-to-external attempts (egress)
    - ➢ external-to-internal attempts (ingress)
  - East-West communication
    - ➢ intra-cluster (between namespaces) communication
    - ➢ inter-namespace (between services in the same namespace) communication

- Non-standard & not-approved networking configuration attempts such as serviceExternalIP and NodePort

Invalid Permissions Logs

- Every RBAC / SCC modification/creation/deletion related audit logs (e.g. creating a new RoleBinding/ClusterRoleBinding for a user/group with elevated privileges) - with high severity when it comes to cluster-roles.
- Any API calls that failed due to RBAC/SCC policy permissions (Authorization policy ⇒ Admission Controller)

OAuth logs

- successful login attempts + timestamp + user identity
- metrics-based alerts such as:

Prometheus metric	rule should include	oc / web ui login attempts
`openshift_auth_basic_password_count`	if metric value (in X seconds) >= max_count	oc login
`openshift_auth_basic_password_count_result`	`if result==error` (in X seconds) `  i++` `if i>= max_count` `  alert`	web ui
`openshift_auth_form_password_count`	if metric value (in X seconds) >= max_count	oc login
`openshift_auth_form_password_count_result`	`if result==error` (in X seconds) `  i++` `if i>= max_count` `  alert`	web ui

- multiple failed login attempts to the platform during a short period

Application non-standard behavior based on known CVEs

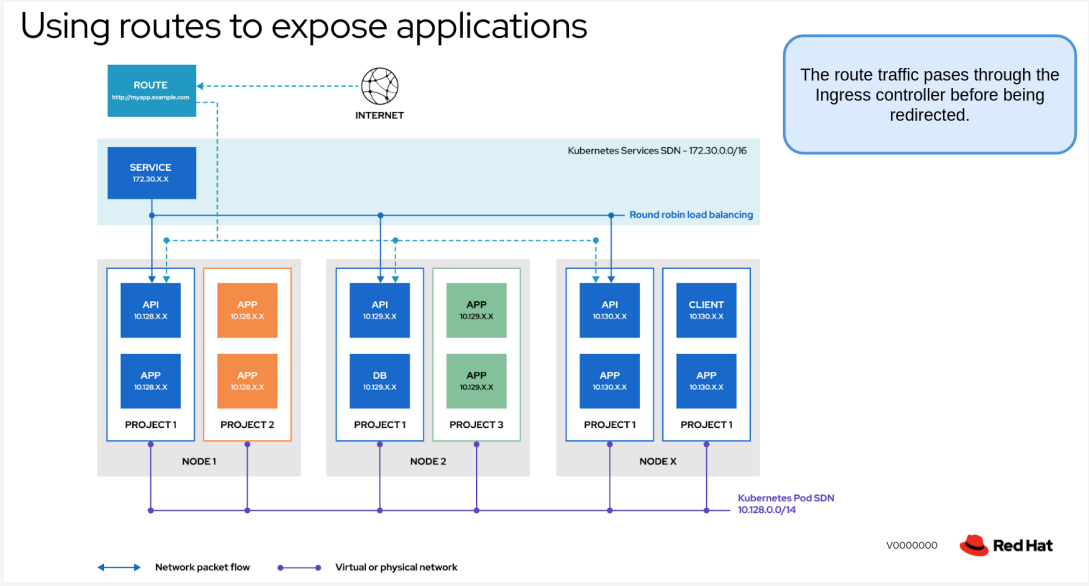
6.8. Networking and Certificates

6.8.1. Routes, types, and network policies

6.8.1.1. Standard OpenShift Routes

Routes provide ingress (External) traffic into services in the cluster. Routes are enhanced alternative objects of OCP for the Kubernetes ingress objects and they are the standard way to expose services in OCP environments. Routes provide advanced features that may not be supported by Kubernetes ingress controllers through a standard interface, such as TLS re-encryption, TLS passthrough, and split traffic for blue-green deployments, etc.

OpenShift provides multiple types of routes that can be configured with relative-ease with pre-built security capabilities, designed to provide several solution options to applications;

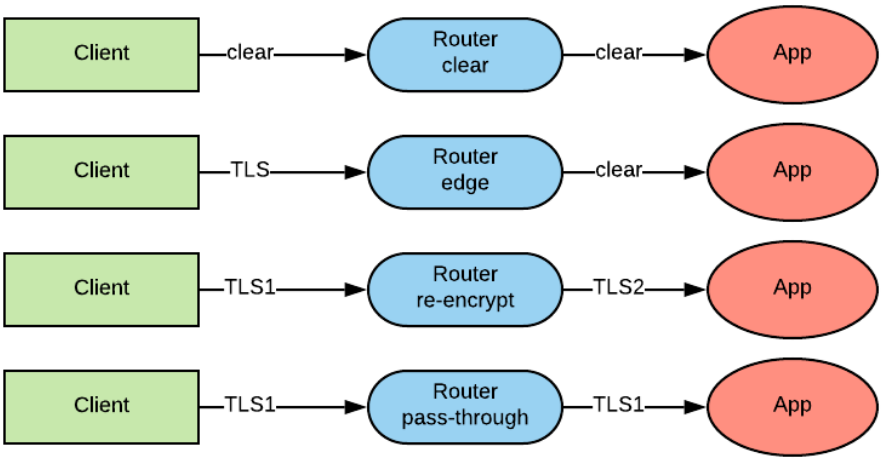


The three most commonly used & secured-acceptably are:

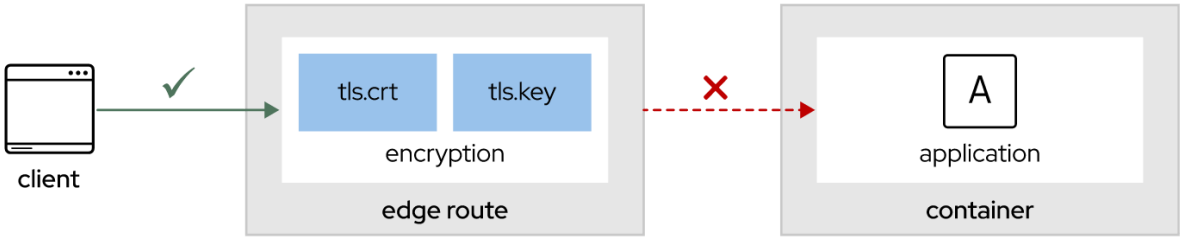
1. Edge route
2. Passthrough route
3. Re-Encryption route

There's also obviously the non-encrypted-at-all option (which is just exposing route without specifying its type) - **this option is not acceptable** and there's no reason to use it as we will see in the next sections.

Block any attempt to expose service with a "regular" route; It can be done by writing an admission-controller policy that will prevent yamls of type=route that does not contain the keyword "tls" from being created.



6.8.1.1.1. Edge Route



Probably the most common option of them all. And **it's a good overall option for external communication standards for our cluster.**

With edge termination, TLS termination occurs at the router, before the traffic is routed to the pods. The router serves the TLS certificates, so you must configure them into the route (or if you don't - the OpenShift platform will generate one for you, based on the internal CA's certificate (it can be signed by the organizational CA and will be explained in a later section); The OpenShift platform is also responsible to rotate the certificate when it expires automatically.

Because TLS is terminated at the router, connections from the router to the endpoints over the internal network are not encrypted.

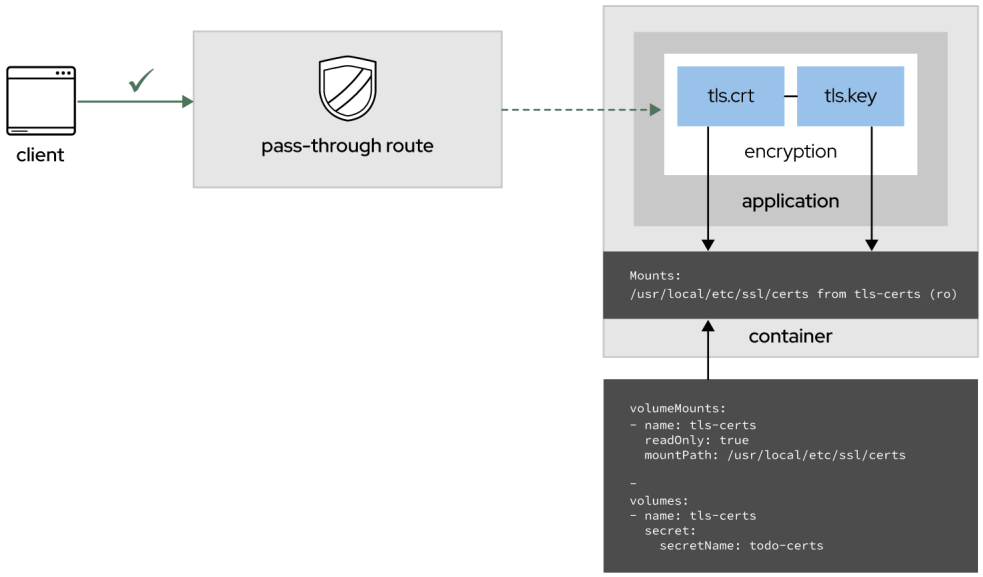
It's not considered unsecured tho, but it does raise the relevant question - how could we and should we even bother with controlling traffic inside the cluster? because after the traffic passed the route, it can be misused like any other traffic. Both will be answered in later sections.

```
# Option 1:
$ oc create route edge --service <app_service_name> [--hostname <subdomain>] --cert <tls.crt>
```

```
--key <tls.key> --ca-cert <ca.crt>

# Option 2:
$ oc create route edge --service <app_service_name>
```

6.8.1.1.2. Passthrough Route



With passthrough termination, encrypted traffic is sent straight to the destination pod without the router providing TLS termination. In this mode, the application is responsible for serving certificates for the traffic.

It's a good solution for applications that migrated to a microservice-oriented architecture and already has a tls-responsible component, or for applications with dedicated encryption protocol/algorithm/mechanism as derived from the data-sensitivity-level.

By using passthrough you're making sure that the communication across the entire channel between the client and the application is encrypted.

```
$ oc create route passthrough --service <app_service_name> [--hostname <subdomain>]
```

6.8.1.1.3. Re-Encrypt Route

Re-encryption is a variation on edge termination, whereby the router terminates TLS with a certificate, and then re-encrypts its connection to the endpoint, which might have a different certificate (notice the *--dest-ca-cert* parameter). Therefore, **the full path of the connection is**

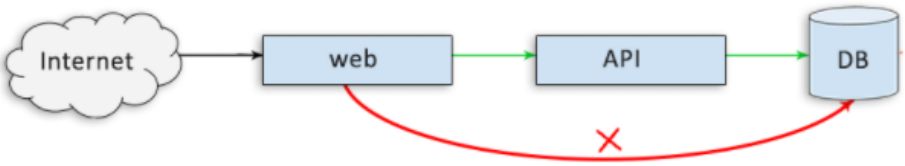
encrypted, even over the internal network. The router uses health checks to determine the authenticity of the host.

```
# Option 1:
$ oc create route reencrypt --service <app_service_name> --dest-ca-cert <destca.crt>
[--hostname <subdomain>] --cert <tls.crt> --key <tls.key> --ca-cert <ca.crt>

# Option 2:
$ oc create route reencrypt --service <app_service_name> --dest-ca-cert <destca.crt>
[--hostname <subdomain>]
```

6.8.1.2. Cross-Namespace communication - Inter-cluster communication - Network Policies

It is mission-critical (security-wise) to prevent communication between services in openshift that weren't meant to connect at all in the first place; This is relevant for cross-namespace communication and in-namespace communication as well.



Network policy prevents unauthorized service-to-service communication

As far as security is concerned, with regards to containers - every container is a sovereign entity that has access to its sensitive information, and each can be compromised by an attacker, and become a potential "backdoor"/"relevant attack vector" into your environment/project. This perception is called "Zero-trust perimeter" which has been defined as a consequence of the trend to move to microservices-driven-applications.

The most intuitive to tackle this problem is networkpolicy objects.
Networkpolicy objects are cloud-native objects that can grant access to specific communication channels, required for our application to function properly.

By default, when a project is getting created, it does not have any such networkpolicies in place and all connections (external/internal, etc.) are allowed.

Important! Once we apply even a single networkpolicy object onto our project, even if it is empty, the behavior of Kubernetes "flips" and all connections are disabled by default unless stated otherwise in a networkpolicy object that enables the communication.

OpenShift/Kubernetes networkpolicy enforcement mechanism flow works like that:

1. A new package arrives at the ingress controller router

2. The platform checks what's the destination service of the package
3. The platform looks for a network policy on the service's project that approves that this package is acceptable.
4. If no such policy is found, the packet is being dropped.

As a rule of thumb it's best advised to make sure every new project on our cluster has these 2 networkpolicy objects right from the start:

1. Block all communication (Non-deletable/editable by the projects' standard users - can be achieved by removing the "delete & update & edit " capabilities on networkpolicy objects in the RBAC policy).

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: deny-by-default
spec:
  podSelector:
  ingress: []
```

2. Allow external communication via Ingress routers (edge/passthrough/re-encrypt) **to our internal frontend services only**
    a. Option 1: Edge routes - internal comm on an unencrypted port

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress-and-TCP-8080-to-frontend-only
spec:
  ingress: # Source
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress # OCP Router
    - ports: # Specific Protocol
      - protocol: TCP
        port: 8080 # Edge routes only
  podSelector: # The Destination Service
    matchLabels:
      app: frontend
  policyTypes:
  - Ingress
```

    b. Option 2: Passthrough / Reencrypt routes - internal communication on encrypted port

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-from-openshift-ingress-and-TCP-8080-to-frontend-only
spec:
  ingress: # Source
  - from:
    - namespaceSelector:
        matchLabels:
          network.openshift.io/policy-group: ingress # OCP Router
    - ports: # Specific Protocol
      - protocol: TCP
        port: 443 # Passthrough/reencrypt routes only
  podSelector: # The Destination Service
    matchLabels:
      app: frontend
  policyTypes:
  - Ingress
```

And finally, each organization should consider its approach to in-namespace & cross-namespace communication;

- allow all communication within a specific namespace (not-recommended as explained before)

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-same-namespace
spec:
  podSelector:
  ingress:
  - from:
    - podSelector: {}
```

- allow only specific ports to specific services, from specific sources (can be customized)

```yaml
kind: NetworkPolicy
apiVersion: networking.k8s.io/v1
metadata:
  name: allow-27107-to-mongo
spec:
  podSelector:
    matchLabels:
      app: mongodb
  ingress:
  - from:
    - podSelector:
        matchLabels:
          app: app
    ports:
    - protocol: TCP
      port: 27017
```

Make a white-list of common applications (MongoDB, PostgreSQL, etc) and their standard ports (27017/TCP, 5432/TCP respectively) and for internal-applications services as well, and allow connections to them only & monitor any anomalies proactively from the central-dedicated security-monitoring service of the organization (under the jurisdiction of the SOC teams).

As can be seen in the example below, this query fetches the networkpolicy objects from the entire cluster, parsing them, and presenting them - it's an easy way to see that my application (MongoDB in that case) is accessible via routes only, on port 27017/TCP which is the standard for that service.

Query:

```
curl -s -k -H "Authorization: Bearer $TOKEN" \
-H 'Accept: application/json' \
https://$ENDPOINT/apis/networking.k8s.io/v1/networkpolicies | jq -r \
'"Type:" + .items[].spec.policyTypes[0] +
", name: " + .items[].metadata.name +
", namespace: " + .items[].metadata.namespace +
", From labeled: " + .items[].spec.ingress[].from[].podSelector.matchLabels.app +
", Port: " + (.items[].spec.ingress[].ports[].port|tostring) +
", Protocol:" + .items[].spec.ingress[].ports[].protocol+
", To labeled: "+ .items[].spec.podSelector.matchLabels.app +
"\n"' | sort -u
```

Result:

```
Type:Ingress, name: allow-27107, namespace: test, From labeled: app, Port: 27017, Protocol:TCP, To labeled: mongodb
Type:Ingress, name: allow-27107, namespace: www, From labeled: app, Port: 27017, Protocol:TCP, To labeled: mongodb
```

Needless to say that this parsing method is not a production-ready example but it is an example of the general more-important approach of proactive monitoring of our policies compliance.

Note! Make sure that this procedure is being done by a dedicated-read-only service account with ClusterRoleBinding that has RBAC permissions to fetch networkpolicy objects only, as derived from the “least privileges" perception.

6.8.1.3. Egress Communication Policies

In-cluster services may be required to initiate external connections for many different reasons. To prevent the exploited container from becoming a pivoting point for an attacker, we can enforce the exact connections that each container can initiate outside of the cluster and thus tightly narrowing the area of effect of compromised services.

Egress networkpolicies are fairly common and are easy to create usings-native objects. E.g:

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-egress
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

```yaml
---
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all-egress-within-namespace
spec:
  podSelector: {}
  egress:
  - {}
  policyTypes:
  - Egress
```

Unlike ingress policies, which are mandatory because every cluster within an organization is bound to have sensitive resources that we want to control access to, but egress is not so obvious, and is really dependent on the specification of each application, on the specification of external components that it has to communicate with if it even has any. **The best practice is to**

deploy the deny-all-egress policy + allow-all-egress-within-namespace as a first step and then customize the egress policies based on each application's needs.

6.8.1.4. non-Standard OpenShift Routes

Now we got to the picantry, but it is important to mention these options and prevent access to them as prematurely as possible.

6.8.1.4.1. NodePort

NodePort exposes an application's service API object from each cluster worker.
A NodePort exposes the service on a static port on the node's IP address. NodePorts are in the 30000 to 32767 range by default, which means a NodePort is unlikely to match a service's intended port.
This feature can be relevant for development environments where programmers would access their service directly for debugging purposes etc.

But, and it's a serious clarification - **in production environments it is essential to block this feature**, otherwise, we are enabling direct access to the clusters, which is the exact opposite of what we did earlier when we talked about the "oc debug" cmd. We want to make a standard way to access our applications and it should be via standard, encrypted routes only, and NodePort just jeopardizes that effort.

Blocking NodePort

```
$ oc create quota --hard=services.nodeports=0 nodeport
```

It should be mandatory on every project within our cluster, made by the cluster-admin and non-deletable by any other user.

6.8.1.4.2. ExternalIP

As a cluster administrator, you can designate an IP address block that is external to the cluster that can send traffic to services in the cluster

For non-cloud environments, the OpenShift Container Platform supports the assignment of the external IP address a Service object spec.externalIPs field through the ExternalIP facility. This exposes an add virtual IP address, assigned to the service, that can be outside the service network defined for the cluster. A service configured with an external IP functions similarly to a service with type=NodePort, allowing you to direct traffic to a local node for load balancing.

It can be required for accessing non-http/s based applications like Postgresql by external entities etc.
This solution is very much the same as NodePort - you're enabling direct access to nodes, but this time you enable it on a dedicated virtual IP rather than on a dedicated static high-port.

Block ExternalIP

Example policy objects

The examples that follow demonstrate several different policy configurations.

- In the following example, the policy prevents OpenShift Container Platform from creating any Service with an external IP address specified:

Example policy to reject any value specified for Service `spec.externalIPs[]`

```
apiVersion: config.openshift.io/v1
kind: Network
metadata:
  name: cluster
spec:
  externalIP:
    policy: {}
  ...
```

By preventing from externalIPs from being generated for the cluster, you ultimately disable this feature completely. It can be done by cluster-admin only.

6.8.1.5. OpenShift Networking Best Practices - Summary

All of the following suggestions are negotiable because networking requirements could be flexible and very different between applications, but they should be the ground base for the default state of the cluster.

- Disable ServiceExternalIP
- Disable NodePort
- Default Ingress Policy (North-South)
    - Disable all external access to a project, other than via securely exposed routes (edge, passthrough,re-encrypt)
        - Disable access to this default policy using RBAC
- Enable specific networkpolicies between projects' microservices (East-West)
- Proactively query the platform using a secure & read-only service account, to look for abnormalities in standard protocols' port numbers
- Deploy network capabilities enhancing products such as Service Mesh for more flexibility, network-observability, and better-configured network access to your services.

Note! For any out-of-standard service that requires a dedicated networking solution such as using a special CNI-plugin, etc. - make sure that it is monitored (security-wise) properly to prevent breaches.

6.8.2. Secure External Access Points

6.8.2.1. Sign the Ingress Controller Certificate with External CA

The Ingress Controller is the main component that exposes in-cluster entities to outside/external communication.

To increase security for external access points, custom certificates from an external CA can and should be installed for the public hostnames.

```
# config map that includes only the root CA certificate used to sign the wildcard
certificate
$ oc create configmap custom-ca \
     --from-file=ca-bundle.crt=</path/to/example-ca.crt> \
     -n openshift-config

# familiar the proxy with the new configmap
$ oc patch proxy/cluster \
     --type=merge \
     --patch='{"spec":{"trustedCA":{"name":"custom-ca"}}}'

# secret that contains the wildcard certificate chain and key (it needs to sign
newly exposed services with the same certificate and key)
$ oc create secret tls <secret> \
     --cert=</path/to/cert.crt> \
     --key=</path/to/cert.key> \
     -n openshift-ingress
# familiar the ingress controller with the new secret
$ oc patch ingresscontroller.operator default \
     --type=merge -p \
     '{"spec":{"defaultCertificate": {"name": "<secret>"}}}' \
     -n openshift-ingress-operator
```

The following documentation provides a precise “How-To” guide to replace the default ingress certificate with a custom one, signed by an external CA.

https://docs.openshift.com/container-platform/4.6/security/certificates/replacing-default-ingress-certificate.html

6.8.2.2. Replace Default API Server Certificate

The default API server certificate is issued by an internal OpenShift Container Platform cluster CA. Clients outside of the cluster will not be able to verify the API server’s certificate by default.

You can add one or more alternative certificates that the API server will return based on the fully qualified domain name (FQDN) requested by the client, for example when a reverse proxy or load balancer is used.

```
$ oc create secret tls <secret> \
     --cert=</path/to/cert.crt> \
     --key=</path/to/cert.key> \
     -n openshift-config

$ oc patch apiserver cluster \
     --type=merge -p \
     '{"spec":{"servingCerts": {"namedCertificates":
     [{"names": ["<FQDN>"],
     "servingCertificate": {"name": "<secret>"}}]}}}'
```

The following documentation provides a precise “How-To” guide to replace the default API Server Certificate with a custom one, signed by an external CA.

https://docs.openshift.com/container-platform/4.6/security/certificates/api-server.html#api-server-certificates

6.8.3. Internal CAs wildcard Certificates - Security Overview and Considerations

Strong certificate use within the platform is critical to modern application security. The only way for public key infrastructure (PKI) to scale for a container orchestration platform is by increasing the use and reach of automation. OpenShift provides integrated management of X.509 certificates for internal cluster components. **Containerized applications are responsible for managing their certificates signed by organizational CAs or may make use of the OpenShift Service CA if they wish.**

The platform includes multiple certificate authorities (CAs) providing independent chains of trust, The certificates generated by each CA are used to identify a particular OpenShift platform component to another OpenShift platform component.

The internal infrastructure CA certificates are self-signed. While this process might be perceived as bad practice by some security or PKI teams, any risk here is minimal. **The only clients that implicitly trust these certificates are other components within the cluster.**

CA bundles are used when more than one communication path needs to be authenticated.

- **Communication between the API server and the kubelet is secured by the kubelet serving CA.**

- **Communication with etcd is secured by the etcd serving CA.**
- **Authentication to the kubeconfig is managed by the admin-kubeconfig-client CA.**

The OpenShift CAs are managed by the cluster and are only used within the cluster. This means that :

- Each cluster CA can only issue certificates for its purpose within its cluster.
- CAs for one OpenShift cluster cannot be used for a different OpenShift cluster, thus avoiding cross-cluster interference.
- Cluster CAs cannot be used by an external CA that the cluster does not control.

Long-term certificates are a point of vulnerability. **OpenShift automatically manages the rotation of certificates generated by the internal CAs.**

With OpenShift 4, it is not possible to replace certificates for internal platform components with certificates from a different CA.

It’s designed like that after the following points:

- Orchestrating a roll-out of externally provided certificates introduces the risk of cluster downtime due to mistakes in certificates or delays in installing or updating certificates
- **The cluster would require access to the CA's private key,** which is not advised because:
    - It could automatically provision certificates for various cluster components
    - **An individual or service with access to the private key on the cluster could provision certificates for the company's other services**

A platform compromise would then put the company's entire infrastructure at risk

7. Best Practices Suggestions - Summary

Platform	
Authentication & Special Users Management	
1	OAuth - OpenID Connect (OIDC) identity provider for standard users login
2	OAuth - Configure OpenID Connect with more restricted authentication flow for admins
3	OAuth - Edit the default lifespan of the access & authorize tokens
4	Delete the ***kubeadmin*** user after creating a new identity provider + grant cluster-admin cluster role to relevant users
5	Store ***kubeconfig*** file in a secured vault
Authorization	
6	RBAC - Create restricted as possible roles based on the groups in the organization (e.g. “read-only” for SOC, “edit” for developers - for specific objects only, full control to cluster admins, etc.)
7	Sync Users to the platform with the LDAP sync operator and associate them with the relevant group
8	SCC 1. Don’t ever grant Privileged SCC to a Service Account 2. Don’t change the original default SCCs, copy them and deploy the new versions under newly (proactively monitored) SCC 3. Monitor for any changes to the default SCCs, especially the ‘restricted’ SCC. It’s a bad habit that managers have to change the default SCC to make it easier for them to manage the platform but the side-effect is that over-permissive permissions are allowed by default. 4. If you do need some non-restricted SCC, add only the must-have capabilities required, after a security-oriented authority approves this action; Follow the “least-privileges” methodology. 5. Do not allow one of the following to be “True” - allowHostIPC, allowHostNetwork, allowHostPID, allowHostPorts 6. Avoid using RunAsAny in the SCC policies. Instead, if needed, use “MustRunAsRange”/”MustRunAs" 7. Avoid changing seccomp profiles in SCC as much as possible.

9	Integrate SCC with RBAC for easier & automated management
Service Accounts	
10	Generate a dedicated service account for each new deployment/service in a project.
11	Implement an admission controller policy that checks if a newly deployed service has a dedicated mentioning of a specific service account
12	Make sure the service account's RBAC & SCC policies are as restricted as possible
13	Restrict Automounting of Service Account Tokens 1. Disable all services from using the service account token. 2. Disable a specific service from using the service account's token.
14	Configure ExpirationSeconds on ServiceAccountTokens within Pods
etcd	
15	Encrypt etcd
16	Backup etcd periodically & monitor health status consistently
Networking & Certificates	
17	Make sure that only secured routes are allowed on the cluster, either by proactive monitoring & alert mechanism or either admission controller policy ● Prevent services exposure without encryption ● Make sure per-project that only frontend services are exposed outside
18	Deploy network capabilities enhancing products such as Service Mesh for more flexibility, network-observability, and better-configured network access to your services.
19	● Define a Default Ingress Policy (North-South) ○ Disable all external access to a project, other than via securely exposed routes (edge, passthrough,re-encrypt) ■ Disable access to this default policy using RBAC ● Enable specific networkpolicies between projects' microservices (East-West) ● Proactively query the platform using a secure & read-only service account, to look for abnormalities in standard protocols' port numbers
Infrastructure	
General	
20	Enable FIPS mode

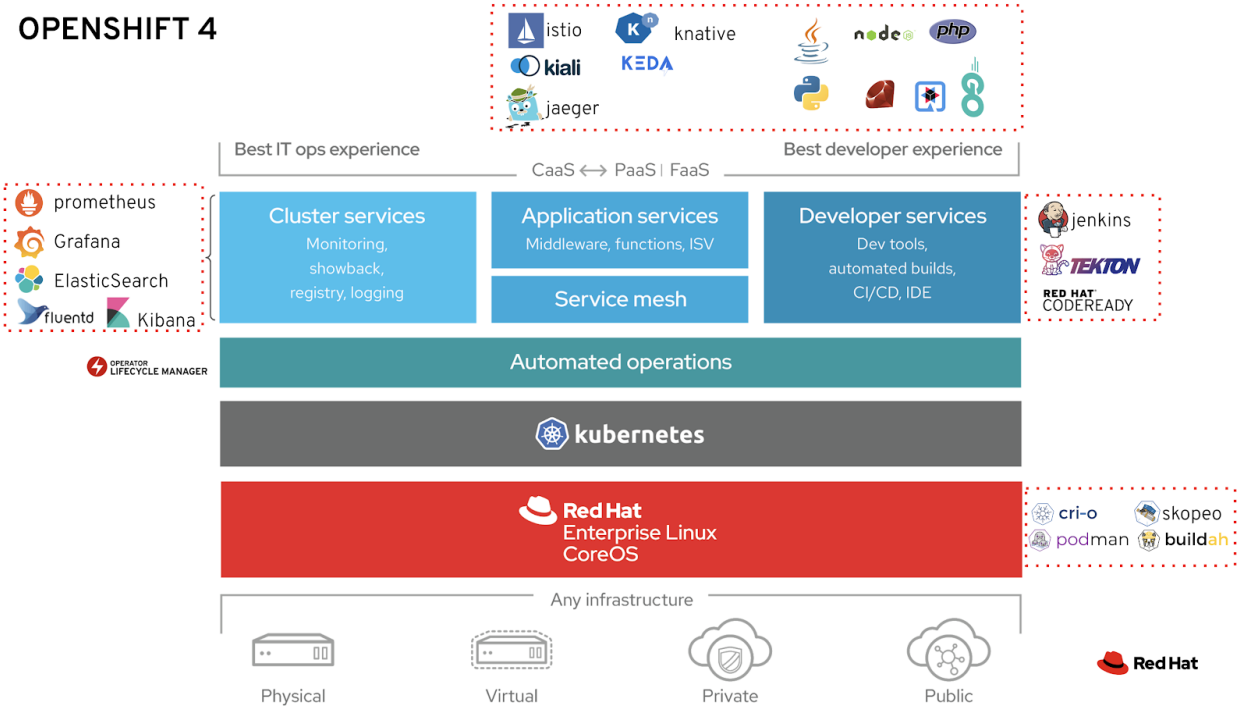
21	S2i - make a decision within the organization whether S2i is approved on the platform
Direct Access to nodes & Making changes to files on RHCOS	
22	Monitor for RHCOS Files Changes
23	Direct access to nodes • Audit any `oc debug node` command usage • Prevent any direct ssh connection to nodes unless from the bastion using core user, and audit any such connection as well • Raise an alert for any direct access attempts to the nodes
Resource Exhaustion	
24	Define resource request and resource limits for any container in a deployment / deploymentconfig
25	Define default project template that includes default quota & limitranges
26	Remove the default “self-provisioner” permission & Prevent it from being auto-updated, and attach it to the relevant users/custom group only.
Trusted Image Sources & Image signing Verification	
27	Use External Registry (with scanning capabilities) for storing & importing images Avoid using the internal registry as the main registry for the network.
28	Verify external image sources, using the following features: • allowed registries (in case s2i is approved in the organization) • Whitelist of Allowed Registries (Pull Only) • Prevent insecureRegistries • Block Specific Registries (in case of private networks - block is known public registries URLs)
29	Validate GPG signature for images before pulling
Storage	
30	Define default RBAC policy - disable user access to PVs (specifically “get” method, the “list” is more acceptable because it may be necessary to make sure that storage has been bound to a PVC correctly - and it does not expose extra-unneeded-sensitive data)
31	Add dynamic storage provisioning solutions with StorageClasses and grant users Read-Only access to use them properly
32	Remove the option for standard users to create their own PVs (RBAC)

33	Limit the **Persistent storage** volume that a single project/service can request using default LimitRange and Quotas in a project template as specified in the *"Resource Exhaustion Prevention"* section
34	Limit the **ephemeral storage** volume that a single project/service can request using "`requests.ephemeral-storage`" and "`limits.ephemeral-storage`" in a project template as specified in the *"Resource Exhaustion Prevention"* section
Monitoring & Observability	
35	Raise log level to at least "WriteRequestsBodies"
36	Use the ClusterLogForwarder feature for log aggregation in a secure manner into an organization-central log store
37	Implement alerts for all anomalies from the log list guidelines specified
38	Integrate complemental 3rd party security monitoring & observability system
Networking & Certificates	
39	Disable ServiceExternalIP
40	Disable NodePort
41	Secure External Access Points
Appendix - Infra Operators	
42	Machine Config Operator - MCO • Prevent any CR object creation & editing with over-privileged file mode (e.g. 777) - Can be done with admission controller policy
43	API Server Operator • Implement a restricted RBAC policy for all of your users in the cluster, based on groups & separation to projects. (This topic is more detailed in the RBAC section in the document). • Monitor the auditing logs for any non-approved operations, and raise an alert accordingly (this topic is more detailed in the monitoring & auditing section in the document) • Sign the API servers using the organizational CA • Protect the API Server by using a complementary solution with an API-management product (e.g. 3Scale) and/or WAF, etc.

8. Appendix

8.1. OpenShift 4 Major Changes

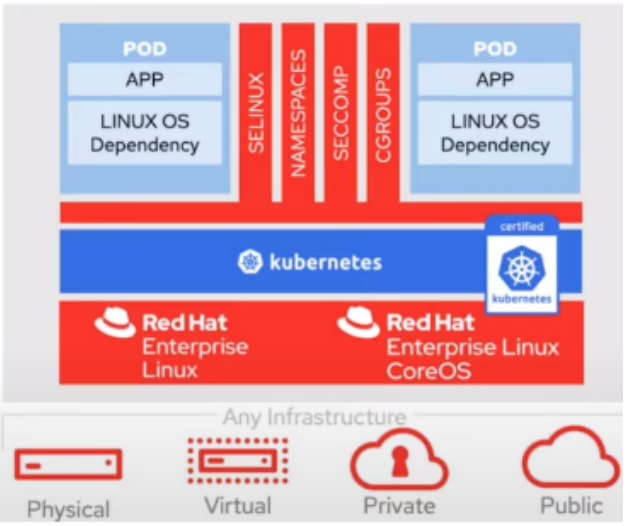
8.1.1. Diagrams

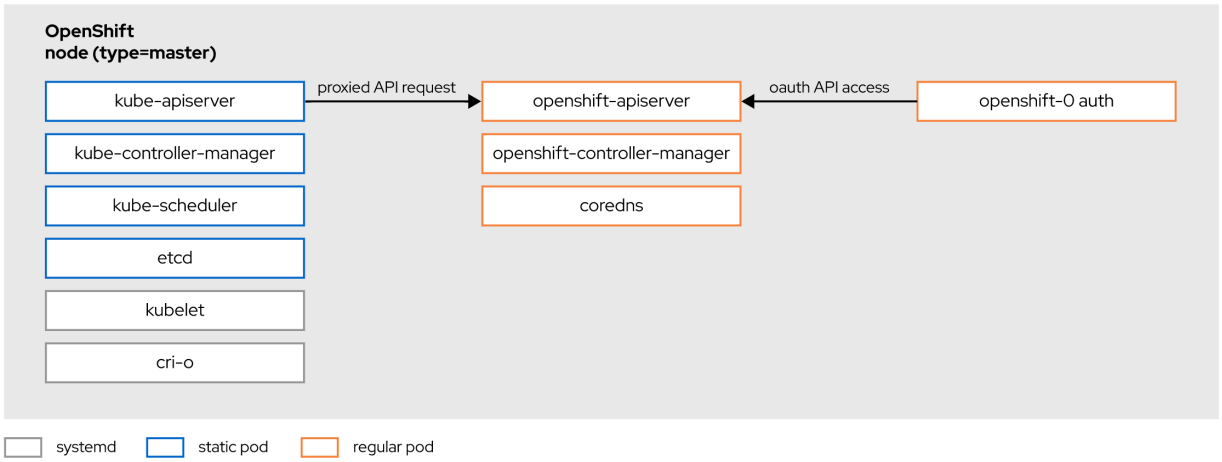


Red Hat

Security features include

- Secure host and container runtime
- Compliance & Hardening
- Runtime Security Policies
- IAM & RBAC
- Project namespaces
- Network Policies
- Integrated & extensible secrets management
- Service Mesh
- Logging, Monitoring, Metrics





8.2. RHCOS

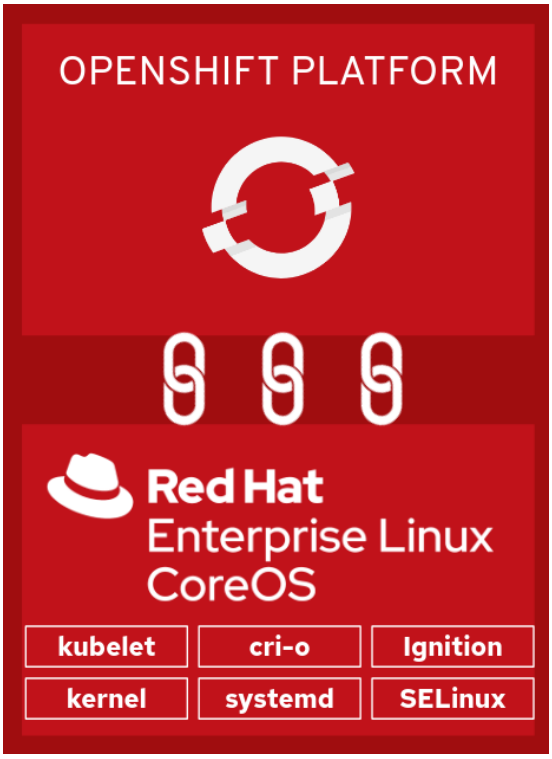
8.2.1. Intro

RHCOS is a single-purpose OS, meaning it is made to do one task -- to run containers. RHCOS is a restricted, container-oriented OS version of the RHEL operating system - it was hardened and stripped from all unnecessary binaries and capabilities and has to default restricted permissions for running containers. Besides, it uses CRI-O containers runtime that has been designed to have as small-attack surface as possible. Also, it is immutable - meaning it cannot be changed during runtime. These features make RHCOS much more resilient from a security perspective - it has less attack surface and it is more complicated for malicious persistence.

Although in OCP 4.x you can technically create compute machines (AKA “worker machines”) that use "plain" RHEL as their operating system, **it’s not recommended due to many security-driven built-in features that RHCOS implements to minimize the potential attack surface.**

RHCOS is supported only as a component of the OpenShift Container Platform.

Note! There are also much more security capabilities for the infrastructure and the platform, but they are not new for OCP 4.x; They will be mentioned later in this document, in the "Infrastructure Security" section."



8.2.2. Day-to-Day management of RHCOS

When you set up your RHCOS machines, you can modify only a few system settings. This design decision is called "**Controlled immutability**" and it's a key feature of RHCOS. This controlled immutability allows OpenShift Container Platform to store the latest state of RHCOS systems in the cluster so it is always able to create additional machines and perform updates based on the latest RHCOS configurations.

After the first boot, RHCOS systems are managed by the Machine Config Operator (MCO) that runs in the OpenShift Container Platform cluster. It operates by implementing Kubernetes-native-objects such as DaemonSets that will be deployed on all nodes/specifically labeled nodes in the cluster and will be consistent on all of them.

It is designed in such a way that you minimize the attack surface on your nodes by preventing direct access to the nodes (unless there's a difficult bug that requires the cluster-admin to debug the nodes).

With such a design we can implement more advanced monitoring and auditing automatic processes that look for any changes/access that has been done directly on the nodes.

Furthermore, the MCO operator constantly monitors the nodes' state at all times for anomalies and if any of them are found (MCO) will automatically bring it (the node) back to the desired original state - which makes it very difficult for an attacker to get a proper persistence hold on a machine if he managed to exploit its way there.

It also deploys all custom-made configurations on newly added nodes automatically so consistency is preserved for all machines.

8.2.3. Nodes patch-management

In this methodology of management, RHCOS updates are delivered via container images and are part of the whole OpenShift platform update process.

This simplifies the process of keeping the nodes in the cluster up-to-date and compiled.

8.2.4. Day-2 Compliance

On top of all the security benefits we get by using OCP 4.x with RHCOS, we can enhance the visibility and enforcement of compliance natively in OpenShift using the new Red Hat Compliance Operator.

The Security Content Automation Protocol (SCAP) is a method for using specific standards to enable automated vulnerability management, measurement, and policy compliance evaluation of systems deployed in an organization.

One such implementation is OpenSCAP - an open-source project that can check compliance against several security standards, including NIST, PCI-DSS, and others; It's an easy way to verify, validate & suggest remediation configuration options for our RHEL-based infrastructure.

Red Hat took it one step further and created the "Compliance Operator" - which is Openscap as an operator, adjusted for OCP, and RHCOS specifically.

There will be a separate dedicated document for the Compliance Operator soon after the release of this document.

Meanwhile, readers can find this procedure documented in the following links:

1. https://www.openshift.com/blog/rhel-coreos-compliance-scanning-in-openshift-4?extIdCarryOver=true&sc_cid=7013a0000026H0hAAE
2. https://www.openshift.com/blog/how-does-compliance-operator-work-for-openshift-part-1
3. https://www.openshift.com/blog/how-does-compliance-operator-work-for-openshift-part-2

8.3. Operators

8.3.1. Operators - Overview and main concepts

One of the major differences between OCP 3.x and OCP 4.x is the introduction of Operators.

An operator is a method of packaging, deploying, and managing a Kubernetes-native application. A Kubernetes-native application is an application that is both deployed on Kubernetes and managed using the Kubernetes APIs with the kubectl tooling. Operators create application-specific custom controllers which allow the platform to use Kubernetes to manage Kubernetes.

Although Kubernetes excels at managing applications, it does not specify or manage platform-level requirements or deployment processes. Neither does it include all the elements necessary for an enterprise platform, such as monitoring and logging. Powerful and flexible platform management tools and processes are important benefits provided by the OpenShift Container Platform.

OpenShift uses operators to manage the Kubernetes cluster components. Each operator manages a specific area of cluster functionality, such as cluster-wide application logging, management of the Kubernetes control plane, or the machine provisioning system.

OpenShift Operators play a critical role in both the initial start-up and the on-going maintenance of those components. Specific operators are charged with performing upgrades for each node, as well as on particular low-level Kubernetes components.

Operators create application-specific custom controllers which allow the platform to use Kubernetes to manage Kubernetes.

OpenShift ensures that the platform components are deployed and managed as declared by the operator. When configuration changes are supported, those changes are made through the operator for the individual component. **If an unsupported configuration change is made, the operator will reset the component back to the supported configuration** (leveraging the declarative nature of Kubernetes). This allows operators to manage configuration drift.

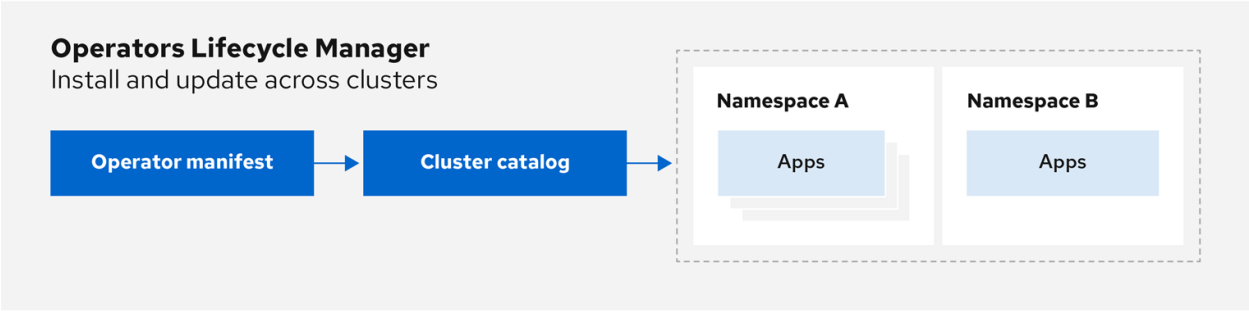
OCP Operators (which manage the infra and crucial platform functions) are deployed on "OpenShift-" namespaces; **They are accessible by default to cluster admins only** and they can provide access permissions to other parties. **It should be managed carefully and with "least privileges" perception in mind** - which means - strict RBAC policy (will be described later) and restricted SCC policy (will be described later) + specific added-capabilities which are specifically required by the operators' service accounts.

8.3.2. OLM - Operator Lifecycle Manager

Operator Lifecycle Manager (OLM) helps users install, update, and manage the lifecycle of Kubernetes native applications (Operators) and their associated services running across their OpenShift Container Platform clusters.

It has a few important CRs (Custom Resource);

`CatalogSource` - A catalog source represents a store of metadata that OLM can query to discover and install Operators and their dependencies. I**t's very important to make sure that only allowed and trusted images' registries are configured as legitimate CatalogSources**, exactly as we should be treating standard image-registries that we pull images non-operator-related into a cluster.



It's important to mention that we can pull & deploy operators' images, not via the OLM but manually, from a regular registry. So to cover that angle as well, **a valid suggestion is to**

proactively monitor every-running-container & image-stream on our cluster and verify it has a legitimate registry as a source.

It is also recommended to scan the available operators in our cluster frequently (specifically the dependency list of each of them), to find unpatched and even vulnerable images that our operator may depend on, and delete them accordingly, so it won't be deployable by developers on our cluster. Make sure you have an updated version before actually deleting the existing operator.

On restricted networks, it's also advised to disable all the default CatalogSources to prevent registry-spoofing attack vectors. (I.E. an attacker that deploys a server with a fake DNS record, one of the default sources for example, and then he/she pulls the malicious image with the facade of a legitimate operator into the cluster).

```
$ oc patch OperatorHub cluster --type json -p '[{"op": "add", "path":
"/spec/disableAllDefaultSources", "value": true}]'
```

8.3.3. RHCOS infrastructure management Operators

All of the following Operators are crucial for the proper functioning of the cluster; **They all should be monitored for any change in their status and audited for any manual change that the cluster admin requires to do by using them**. Furthermore, **only cluster admins should have access to those operators' namespaces.**

8.3.3.1. Cluster Version Operator (CVO)

The OpenShift Container Platform update service is the hosted service that provides over-the-air updates to both OpenShift Container Platform and Red Hat Enterprise Linux CoreOS (RHCOS). The Cluster Version Operator (CVO) in your cluster checks with the OpenShift Container Platform update service to see the valid updates and update paths based on current component versions.

The CVO installs other operators onto a cluster. It is responsible for applying the manifests each operator uses (without any parameterization) and for ensuring an order that installation and updates follow.

The following set of commands allows a cluster administrator to verify the proper versioning of the images in a specific release as part of an upgrade/installation process.

```
$ oc image extract quay.io/openshift-release-dev/ocp-release:4.5.1-x86_64 --path
/:/tmp/release
$ ls /tmp/release/release-manifests
```

```
$ cat /tmp/release/release-manifests/release-metadata
$ $ cat /tmp/release/release-manifests/image-references
```

The `ClusterVersion` is a custom resource object which holds the current version of the cluster (and all the required dependencies it comprises, such as the images for the platform-management operators, etc.). This object is used by the administrator to declare their target cluster state, which the cluster-version operator (CVO) then works to transition the cluster to that target state.

The `ClusterOperator` is a custom resource object which holds the current state of an operator. This object is used by operators to convey their state to the rest of the cluster.

The CVO sweeps the release image and applies it to the cluster. On upgrade, the CVO uses “clusteroperators” to confirm successful upgrades. Cluster-admins make use of these resources to check the status of their clusters.

```
$ oc get clusterversion/version -o jsonpath='{.status.desired.image}{"\n"}'
$ oc get clusteroperator
```

```yaml
apiVersion: config.openshift.io/v1
kind: ClusterOperator
metadata:
  name: my-cluster-operator
spec: {}
status:
  versions:
    - name: operator
      # The string "0.0.1-snapshot" is substituted in the manifests when the payload is built
      version: "0.0.1-snapshot"
```

“status” can be - `Progressing, Available, Degraded` and **it should be monitored at all times by the cluster admins team.**

Under “`ClusterVersion`” CR you can also declare “unmanaged” components which will not be automatically updated by the CVO to the original version if they have been changed manually. ⇒ **In a properly secured environment, there shouldn’t be any of these - and the infra/platform-security team should monitor for such configuration + look for abnormalities in image versioning against the image-references of that specific version.**

```
$ oc get -o json clusterversion version | jq .spec.overrides
```

```
$ oc get -o json clusterversion version | jq .spec.overrides
[
  {
    "kind": "APIService",
    "name": "v1alpha1.packages.apps.redhat.com",
    "unmanaged": true
  }
]
```

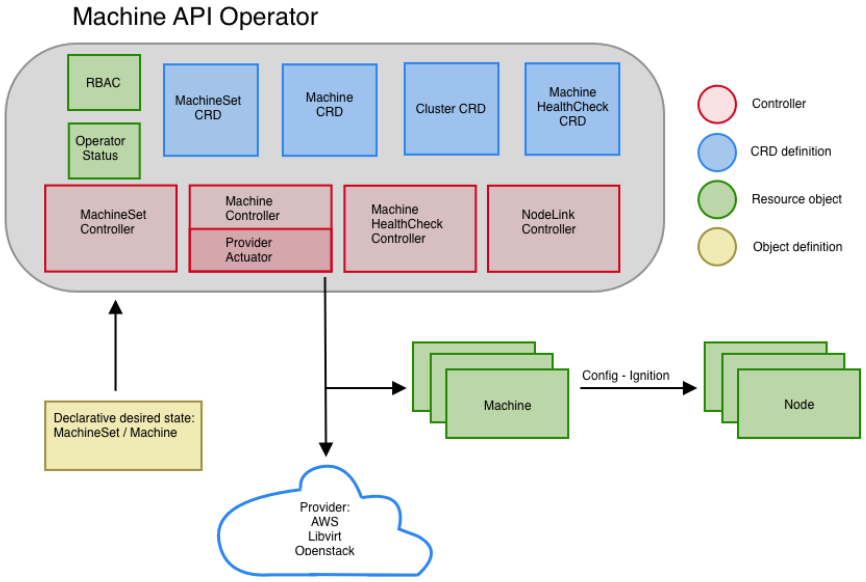
For more information regarding metrics, installation order of different operators, etc. - It is advised to look at the official OpenShift Github project for the CVO - https://github.com/openshift/cluster-version-operator

With regards to upgrades - it's recommended to upgrade the cluster's version for each new stable release to enjoy the benefit of installing the included security-patches.

```
$ oc adm upgrade ⇒ display available versions
$ oc adm upgrade --to-latest=true // --to=<version>
```

8.3.3.2. Machin API Operator (MAO)

The Machine API Operator manages the lifecycle of Cluster's Machines by using specific purpose CRDs, controllers, and RBAC objects that extend the Kubernetes API. This allows us to convey the desired state of machines in a cluster in a declarative fashion.



It's responsible for managing the machines which are part of the cluster;

It does it by introducing & managing three new CRDs to the Kubernetes cluster (objects) - `Machine, MachineSet, MachineHeathCheck.`

It's a very powerful entity in our clusters because it's responsible for the entire lifecycle of each machine in our cluster from the moment it's being added to the cluster, up until it gets deleted from the cluster;

```
$ oc get machines -A -o jsonpath='{range
.items[*]}{@.status.nodeRef.name}{"\t"}{@.status.providerStatus.instanceState}{"\n"}' |
grep -v running

$ oc get nodes -o jsonpath='{range .items[*]}{"\n"}{.metadata.name}{"\t"}{range
.spec.taints[*]}{.key}{" "}' | grep unreachable

$ oc get nodes -l <node_name> | grep "NotReady"
```

When we talk about clusters hosted under an external cloud provider/auto-provision infrastructure tools such as OSP - this is the operator which sends the provision requests to the provider to provide our cluster with more/fewer machines, based on our clients' consumption requirements.

When troubleshooting a Master/Control Plane Machine, you must familiarize yourself with determining the health of the etcd members.

```
$ oc get etcd -o jsonpath='{range
.items[0].status.conditions[?(@.type=="EtcdMembersAvailable")]}{.message}{"\n"}'

$ oc get pods -n openshift-etcd | grep etcd
```

Master/Control Plane Machines are not currently managed by MachineSets, so always ensure you have a backup copy of a master Machine object before deleting it so that you may recreate it easily.

8.3.3.3. Machine Config Operator (MCO)

MCO is the only authority allowed to access the cluster's nodes; It includes: change/push configuration files (such as conf for Kubelet/CRI-O etc.) and run upgrades on the RHCOS OS (rpm-ostree upgrade).
It does it by generating a MachineConfig Object (Custom CRD) for each node and keeping track of its status. It keeps the configuration state consistent on all nodes.

You can also decide to change the configuration to specifically-labeled machines only (like in the example below where it's specified "role: worker").

```yaml
# This example MachineConfig replaces /etc/chrony.conf
apiVersion: machineconfiguration.openshift.io/v1
kind: MachineConfig
metadata:
  labels:
    machineconfiguration.openshift.io/role: worker
  name: example-chrony
spec:
  config:
    ignition:
      version: 2.2.0
    storage:
      files:
      - contents:
          source: data:text/plain;charset=utf-8;base64,<BASE64_DATA>
        filesystem: root
        mode: 0644
        path: /etc/chrony.conf
```

The `machineconfigpool` CR objects track updates on a group of nodes and can be used for monitoring purposes

Make sure to prevent any such CR object creation & editing with over-privileged file mode (e.g. 777) - Can be done with admission controller policy

```
$ oc describe clusteroperator/machine-config
$ oc describe machineconfigpool
```

8.3.3.4. Api Server Operator - OpenShift and Kubernetes

This twin pair of operators are responsible for the OpenShift API Server & Kubernetes API Server - these are the components that receive the external requests via the oc/kubectl binary ⇔ REST API calls for objects fetching/updating/deleting/adding, objects that OpenShift & Kubernetes provides. These servers are accessing the etcd for pulling/pushing/manipulating the data in the users' requests.

Each operator updates its dedicated server and it makes sure the server is up and running at all times.

The API server is served using HTTPS (port 6443/TCP only! non-changeable) with auth (OAuth server - will be specified in a later chapter in that document) & authz (RBAC & SCC - same).

The API Server is also using default OCP-internal-CA cert from the ingress controller - but it can be signed by the Organizational CA for enhanced security because it's a component that communicates with out-of-cluster entities frequently.

Security Suggestions:

1. Make sure to implement a restricted RBAC policy for all of your users in the cluster, based on groups & separation to projects. (This topic is more detailed in the RBAC section in the document).
2. Monitor the auditing logs for any non-approved operations, and raise an alert accordingly (this topic is more detailed in the monitoring & auditing section in the document)
3. Sign the API servers using the organizational CA

- Other solutions may include protecting the API Server by using an integrated solution with an API-management product (e.g. 3Scale) or either WAF, etc.

8.3.3.5. Ingress Operator Operator

Ingress Controller & Wild-Card DNS -
The Ingress Operator is responsible for the proper functioning of the ingress controller - the main OCP router (HAProxy-based), which enables external access to services that run on the OpenShift Container Platform.

The ingress router is signed by the internal-CA by default, and its cert is issued for the sub-domain wildcard network: “apps.subdomain” so it could expose the internal services as if under the same “subdomain” network. (e.g. apps.openshift.example.com). It also exposes the Web UI service and the API server from the previous section, using standard sub-domain hostnames

Best Practice: sign the ingress Controller by the organizational CA for enhanced security

8.3.3.6. Cluster Image Registry Operator

The Cluster Image Registry Operator manages a singleton instance of the OpenShift Container Platform registry. It manages all configurations of the registry, including creating storage.

The image.config.openshift.io/cluster resource can contain a reference to a ConfigMap that contains additional certificate authorities to be trusted during image registry access, such as in pulling images from external registries to the internal one for application deployment, etc.
There is a dedicated section to expand on that topic later in that document - “Trusted Images Sources”.

The registry is not exposed outside of the cluster at the time of installation. **It’s very important to keep it that way**, because we want to prevent a situation that some malicious entity pushes exploitable images to take advantage of and get onto our cluster’s infrastructure, or even reconnaissance tools that are built on top of containers, etc. **In case direct registry access is required it can be achieved from the cluster’s nodes using podman by the cluster admins only, and only after authenticating to the nodes with SSH-keys and not username & passwords.**

8.3.3.7. Cluster Monitoring Operator

The Cluster Monitoring Operator manages and updates the Prometheus-based cluster monitoring stack deployed on top of OpenShift. **Prometheus collects useful information that relates to the cluster's resource-consumption and usage, which can be gathered by security information management to find abnormalities. It's also the best practice that Red Hat suggests for a holistic cluster-monitoring cover.**

Prometheus does not only allow monitoring of the cluster's infrastructure but also applications components' resource consumption. **It is advised to make a standard monitoring policy for the most common components that run in many applications (such as DB for example), based on the default projects' quotas, and apply it automatically with each new component that we deploy on our cluster, for maximum observability in our cluster.**

It's also possible to configure alerts directly from Prometheus to notify the administrators about real-time events.

Each environment has its own limitations, needs, and resources so I'm not elaborating on that point so it'll be impossible to make a generic "to-do list" for monitoring, but every organization should implement that methodology and line of thought in its environments.

8.4. Networking Security Add-ons

There's a whole section of add-on solutions that are not being presented because it's not included by default in OpenShift and requires a separated set of skills and resources to manage, but it is worth mentioning because these products come from the Red Hat portfolio as well, they are both completely open-source, and are very easy to deploy on OpenShift environments with Operators;

8.4.1. 3Scale - API Management

API Management tool grants us a scalable, homogeneous security platform that enables us to secure many of our organization's APIs in one place, and enforce our security compliance to our entire exposed APIs with enough flexibility to make custom adjustments for each unique API based on our needs.

It monitors the requests to the API, enables/disables access to sensitive information to different users based on roles, verifies each source that tries to access the service, controls the rate of the calls to the API, enforces security policies customizable for every exposed API, etc. Just to name a few:

Feature	Description
Access control	Centrally sets up and manages policies and application plans for all APIs
Security	Authenticates and authorizes using API keys, user keys, or OAuth tokens
Rate limits	Manage and control flow of access to API resources
Developer Portal	Allows sign-up, API access, and API documentation for consumers
ActiveDocs live documentation	Based on Swagger framework, provides a way to document APIs and include in the Developer Portal
Analytics	Monitor and set alerts on traffic flow and produce reports
Dashboard	Provides quick, central visibility of all traffic, plans, and customer sign-up flows for the platform

And there are much more like built-in security policies, custom policies features, integration with RHSSO for secured OpenID Connect authentication, authorization, etc.

More information about 3Scale can be found: 1 , 2 , 3 , 4

8.4.2 Service Mesh

Based on the Istio upstream Open Source project, Red Hat Service Mesh is designed to grant us in-cluster and external-cluster communication network security capabilities and flexibility in an enhanced way that the standard networkpolicies and secured routes just can't deliver.

The term service mesh is used to describe the network of microservices that make up such applications and the interactions between them. As a service mesh grows in size and complexity, it can become harder to understand and manage. Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring. A service mesh also often has more complex operational requirements, like A/B testing, canary rollouts, rate limiting, access control, and end-to-end authentication.

Red Hat OpenShift Service Mesh adds a transparent layer on existing distributed applications without requiring any changes to the service code. You add Red Hat OpenShift Service Mesh support to services by deploying a special sidecar proxy throughout your environment that intercepts all network communication between microservices. You configure and manage the service mesh using the control-plane features.

For more information on Service Mesh - 1 , 2