Introduction to Red Hat OpenShift security context constraints

A process running in Linux has access to certain functions that the same process running in a Linux container does not. This is a major aspect of how the container isolates the process--it prevents the process from performing functions that can affect the processes in other containers running in the same Linux kernel. Most applications running in Red Hat OpenShift work fine with these limits on their processes, but some applications need to be able to perform some of these protected functions.

An application deployed to OpenShift can run in a container that allows access to specific protected functions. The application's security context (SC) specifies the permissions the application needs, and the cluster's security context constraints (SCC) specify the permissions the cluster will allow. An SC with an SCC enables an application to request access while limiting the access the cluster will grant.

This article is part 1 of a two-part series on security context constraints (SCC). This one introduces SCCs and gives an overview of how SCCs solve the access problem. Part 2, "Allow containers to access protected Linux functions in Red Hat OpenShift using security context constraints," digs into the details of how to implement and administer SCCs. It assumes you have a general understanding of how to deploy an application to an OpenShift cluster and how the cluster manages a workload.

Deploying a secure pod

By default, OpenShift prevents the containers running in a cluster from accessing protected functions. These functions--Linux features such as shared file systems, root access, and some core capabilities such as the `KILL` command--can affect other containers running in the same Linux kernel, so the cluster limits access to them. Applications can still use these functions, but they need the cluster's permission.

Even a rogue or hacked application cannot grant itself access to protected functionality. The access is configured not by the application (which could be compromised) but rather by the pod that creates the application container and by the cluster that runs it. The application can only access the functions that the pod requests and that the cluster approves.

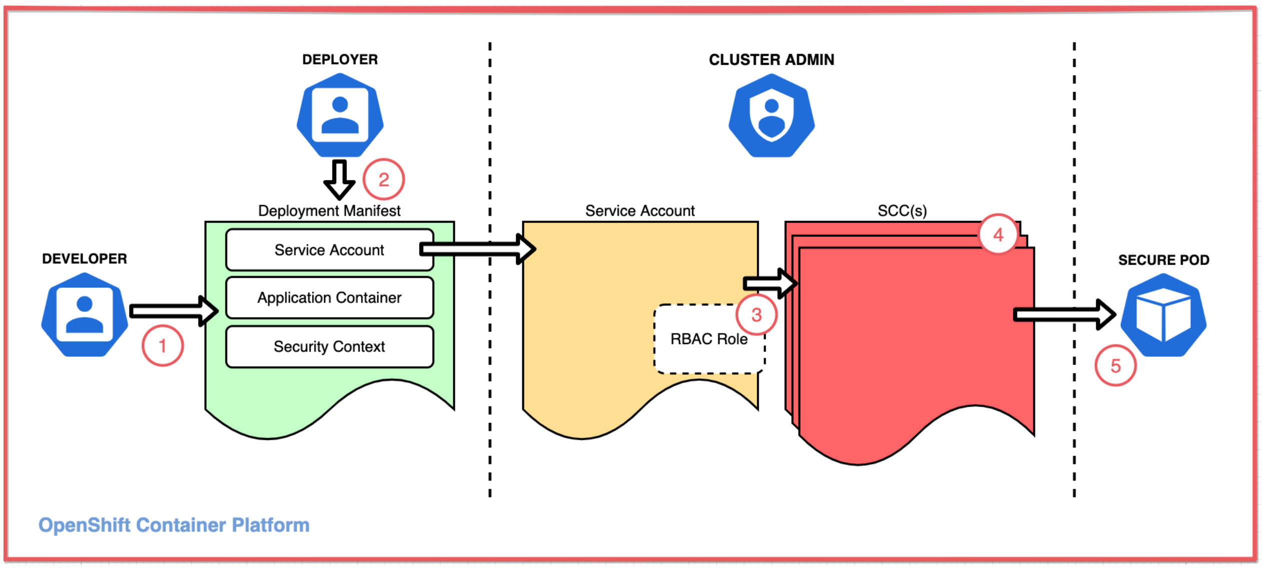
A pod specifies the protected functions its application needs to perform. Of course, the cluster shouldn't allow just any application to request any access it wants, or containers would provide much less isolation. To enforce security, the cluster limits the access pods can enable, allowing some pods to enable access that others cannot.

When the pod creates the application's container, it configures the container to allow the access it specifies. If the application tries to perform a protected function, Linux will block it unless the pod has configured the container to allow access to the function.

An application's access to protected functions is an agreement between three personas:

- A **developer** who writes an application that performs protected functions
- A **deployer** who writes the deployment manifest that must specify the access the application requires
- An **administrator** who decides whether to grant the deployment the access it specifies

This diagram illustrates the components and process that allow an application to access functions:



1. A developer writes an application
2. A deployer creates a **deployment manifest** to deploy the application with a pod spec that configures:
   - A **security context** (for the pod and/or for each container) that specifies the access needed by the application, thereby requesting it
   - A **service account** to grant the requested access
3. An administrator assigns a **security context constraint** to the service account that grants the requested access, thereby allowing the pod to configure Linux as specified
   - The SCC can be assigned directly to the service account or indirectly via an RBAC role or group
4. The SCC may be one of OpenShift's predefined SCCs or may be a custom SCC
5. If the SCC grants the access, the admission process allows the pod to deploy and the pod configures Linux as specified

> **NOTE**: An OpenShift service account is a special type of user account that can be used programmatically without using a regular user's credentials.

A cluster only starts a pod if the SCC grants the permissions requested in the security context. When pod starts, it configures the container as described in the pod's security context.

Now that we know the personas involved, and the general process that they follow, let's explain the components they use in more detail.

Specifying pod security

A pod configures a container's access with permissions requested in the pod's security context and approved by the cluster's security context constraints:

A *security context* (SC), defined in a pod, enables a deployer to specify a container's permissions to access protected functions. When the pod creates the container, it configures the container to allow these permissions and block all others. The cluster will only deploy the pod if the permissions it requests are allowed by a corresponding SCC.

A *security context constraints* (SCC), defined in a cluster, enables an administrator to control permissions for pods, permissions that manage containers' access to protected Linux functions. Similarly to how role-based access control (RBAC) manages users' access to a cluster's resources, an SCC manages pods' access to Linux functions. By default, a pod is assigned an SCC called `restricted` that blocks access to protected functions. For an application to access protected functions, the cluster must make an SCC that allows it available to the pod.

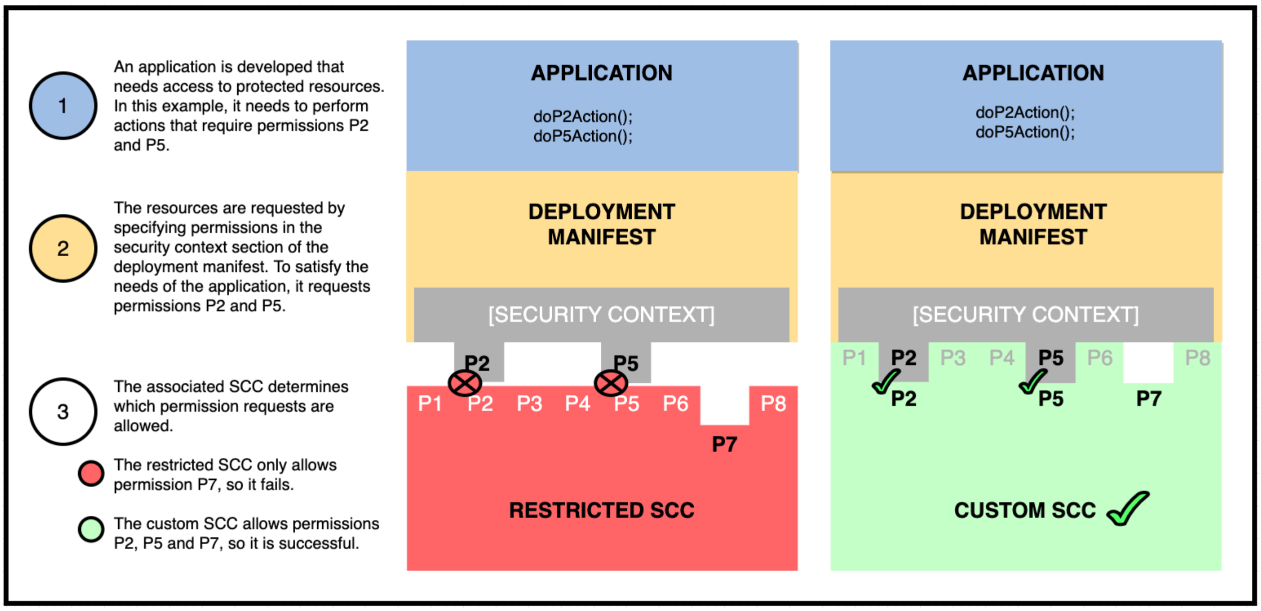
> **NOTE**: SCC enforcement is implemented using SELinux and AppArmor, security modules included in the kernel of all Red Hat Linux distributions. The nodes in an OpenShift v4 cluster can only run on a Red Hat Linux distribution, guaranteeing that the kernel will include those modules.

How a pod requests additional access

While an SCC grants access to protected functions, each pod that wants to use that access must request it. To request access to the functions its application needs, a pod specifies those permissions in the security context field of the pod manifest. The manifest also specifies the service account that the pod expects will be able to grant this access. When the manifest is deployed, the cluster associates the pod with the service account, which is associated with the SCC. For the cluster to deploy the pod, the SCC must grant the permissions that the pod requests.

One way to envision this relationship is to think of the SCC as a lock protecting Linux functions and the manifest being the key. The pod is allowed to deploy only if the key fits.

This diagram illustrates that relationship:


1
An application is developed that needs access to protected resources. In this example, it needs to perform actions that require permissions P2 and P5.
2
The resources are requested by specifying permissions in the security context section of the deployment manifest. To satisfy the needs of the application, it requests permissions P2 and P5.
3
The associated SCC determines which permission requests are allowed.
The restricted SCC only allows permission P7, so it fails.
The custom SCC allows permissions P2, P5 and P7, so it is successful.
APPLICATION
doP2Action();
doP5Action();
DEPLOYMENT MANIFEST
[SECURITY CONTEXT]
P2
P5
P1 P2 P3 P4 P5 P6 P8
P7
RESTRICTED SCC
APPLICATION
doP2Action();
doP5Action();
DEPLOYMENT MANIFEST
[SECURITY CONTEXT]
P1 P2 P3 P4 P5 P6 P8
P2
P5
P7
CUSTOM SCC

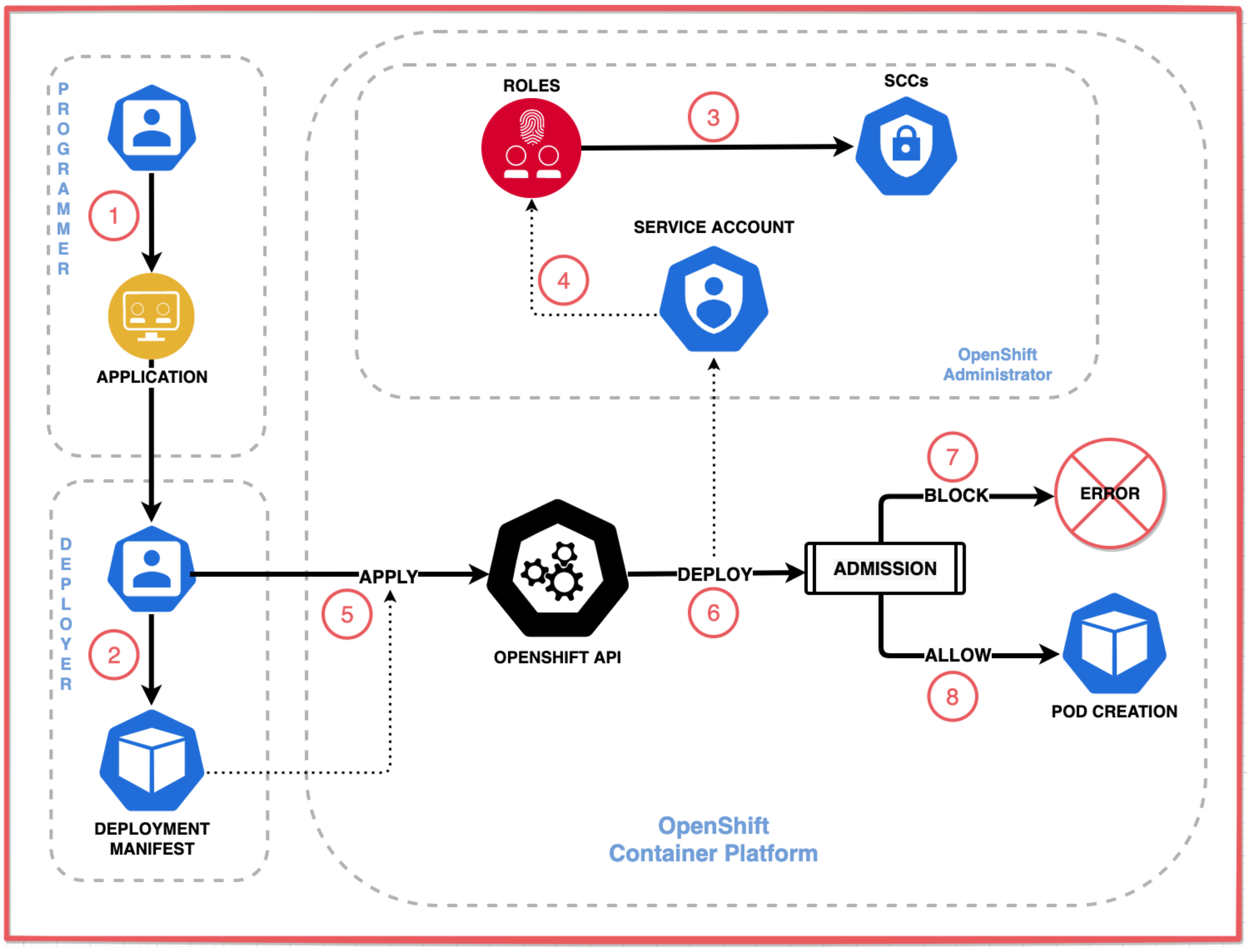
The diagram shows a deployment that will be blocked by the SCC on the left and will be allowed by the SCC on the right. In both examples, the pod specifies in its security context that it needs access to two permissions arbitrarily labeled P2 and P5. In the first example, a very restrictive SCC does not grant the access the manifest requests in its security context, so the cluster will refuse to deploy the pod. In the second example, a custom SCC does grant the access the manifest requests in its security context, so the cluster will proceed with deploying the pod.

> **IMPORTANT**: As a best-practice, each custom SCC should grant as little access as possible. If protected functions are like rooms in a hotel, only allow the users access to the rooms they need. While an SCC can be used to allow a pod to run as a privileged user or the root user, this should be granted to pods very sparingly.

The big picture

To better understand how SCCs control access on an OpenShift cluster, lets walk through a deployment scenario to show how the cluster's admission process uses the deployment manifest, service account, and SCC together to determine whether to deploy a pod.

This diagram illustrates the deployment process:



1. The **developer** implements an application or service that requires access to protected functions, and delivers the application to the deployer.
2. The **deployer** creates a deployment manifest for the application. The manifest specifies a security context and a service account.
3. The **administrator** creates a role and assigns it an SCC.
4. The **administrator** creates a service account and binds it to the role.
5. The **deployer** applies the deployment manifest, thereby deploying the application.
6. OpenShift processes the deployment manifest and attempts to deploy the pod. The deployment process will determine which SCC to use based on the sevice account specified in the manifest. The **admission process** compares the security context of the manifest against the SCC and decides whether to block or allow the pod to deploy.
7. **BLOCK**: Some requested permissions are not granted, so the deployment fails.
8. **ALLOW**: All requested permissions are granted, so the deployment creates the pod, configures the container as described by the pod's security context, and runs the application in that container.

If the pod is denied the requested permissions, the administrator will need to:

- Determine if the additional requests made in the manifest are in fact needed.
- If so, assign the requested permissions to the SCC or select an SCC that already has the requested permissions.

Continue learning

To dig deeper into the details of how SCCs work, check out part 2, "Allow containers to access protected Linux functions in Red Hat OpenShift using security context constraints" of this article series.

To get hands-on experience using SCCs, check out the accompanying tutorial, "Use security context constraints to restrict and empower your OpenShift workloads."