Red Hat OpenShift Pipelines 1.14

Securing OpenShift Pipelines

Security features of OpenShift Pipelines

Last Updated: 2024-07-24

Red Hat OpenShift Pipelines 1.14 Securing OpenShift Pipelines

Security features of OpenShift Pipelines

Abstract

This document provides information about security features of OpenShift Pipelines.

Table of Contents

CHAPTER 1. USING TEKTON CHAINS FOR OPENSHIFT PIPELINES SUPPLY CHAIN SECURITY 4
1.1. KEY FEATURES 4
1.2. CONFIGURING TEKTON CHAINS 4
1.2.1. Supported parameters for Tekton Chains configuration 4
1.2.1.1. Supported parameters for task run artifacts 4
1.2.1.2. Supported parameters for pipeline run artifacts 5
1.2.1.3. Supported parameters for OCI artifacts 6
1.2.1.4. Supported parameters for KMS signers 6
1.2.1.5. Supported parameters for storage 7
1.3. SECRETS FOR SIGNING DATA IN TEKTON CHAINS 11
1.3.1. Signing using cosign 11
1.3.2. Signing using skopeo 11
1.3.3. Resolving the "secret already exists" error 13
1.4. AUTHENTICATING TO AN OCI REGISTRY 13
1.5. CREATING AND VERIFYING TASK RUN SIGNATURES WITHOUT ANY ADDITIONAL AUTHENTICATION 14
1.5.1. Additional resources 16
1.6. USING TEKTON CHAINS TO SIGN AND VERIFY IMAGE AND PROVENANCE 16
1.7. ADDITIONAL RESOURCES 19

CHAPTER 2. SETTING UP OPENSHIFT PIPELINES IN THE WEB CONSOLE TO VIEW SOFTWARE SUPPLY CHAIN SECURITY ELEMENTS 20
2.1. SETTING UP OPENSHIFT PIPELINES TO VIEW PROJECT VULNERABILITIES 20
2.2. SETTING UP OPENSHIFT PIPELINES TO DOWNLOAD OR VIEW SBOMS 24
2.2.1. Viewing an SBOM in the web UI 26
2.2.2. Downloading an SBOM in the CLI 26
2.2.3. Reading the SBOM 27
2.3. ADDITIONAL RESOURCES 27

CHAPTER 3. CONFIGURING THE SECURITY CONTEXT FOR PODS 28
3.1. CONFIGURING THE DEFAULT AND MAXIMUM SCC FOR PODS THAT OPENSHIFT PIPELINES CREATES 28
3.2. CONFIGURING THE SCC FOR PODS IN A NAMESPACE 29
3.3. RUNNING PIPELINE RUN AND TASK RUN BY USING A CUSTOM SCC AND A CUSTOM SERVICE ACCOUNT 29
3.4. ADDITIONAL RESOURCES 32

CHAPTER 4. SECURING WEBHOOKS WITH EVENT LISTENERS 33
4.1. PROVIDING SECURE CONNECTION WITH OPENSHIFT ROUTES 33
4.2. CREATING A SAMPLE EVENTLISTENER RESOURCE USING A SECURE HTTPS CONNECTION 34

CHAPTER 5. AUTHENTICATING PIPELINES WITH REPOSITORIES USING SECRETS 35
5.1. PREREQUISITES 35
5.2. PROVIDING SECRETS USING SERVICE ACCOUNTS 35
5.2.1. Types and annotation of secrets for service accounts 35
5.2.1.1. Git authentication secrets 35
5.2.1.2. Container registry authentication secrets 36
5.2.2. Configuring basic authentication for Git using a service account 37
5.2.3. Configuring SSH authentication for Git using a service account 39
5.2.4. Configuring container registry authentication using a service account 41
5.2.5. Additional considerations for authentication using service accounts 43
5.2.5.1. SSH Git authentication in tasks 43
5.2.5.2. Use of secrets as a non-root user 43
5.3. PROVIDING SECRETS USING WORKSPACES 44

5.3.1. Configuring SSH authentication for Git using workspaces 44
5.3.2. Configuring container registry authentication using workspaces 46
5.3.3. Limiting a secret to particular steps using workspaces 48

CHAPTER 6. BUILDING OF CONTAINER IMAGES USING BUILDAH AS A NON-ROOT USER 50
6.1. CONFIGURING CUSTOM SERVICE ACCOUNT AND SECURITY CONTEXT CONSTRAINT 50
6.2. CONFIGURING BUILDAH TO USE BUILD USER 52
6.3. STARTING A TASK RUN WITH CUSTOM CONFIG MAP, OR A PIPELINE RUN 54
6.4. LIMITATIONS OF UNPRIVILEGED BUILDS 56

CHAPTER 1. USING TEKTON CHAINS FOR OPENSHIFT PIPELINES SUPPLY CHAIN SECURITY

Tekton Chains is a Kubernetes Custom Resource Definition (CRD) controller. You can use it to manage the supply chain security of the tasks and pipelines created using Red Hat OpenShift Pipelines.

By default, Tekton Chains observes all task run executions in your OpenShift Container Platform cluster. When the task runs complete, Tekton Chains takes a snapshot of the task runs. It then converts the snapshot to one or more standard payload formats, and finally signs and stores all artifacts.

To capture information about task runs, Tekton Chains uses **Result** objects. When the objects are unavailable, Tekton Chains the URLs and qualified digests of the OCI images.

1.1. KEY FEATURES

- You can sign task runs, task run results, and OCI registry images with cryptographic keys that are generated by tools such as **cosign** and **skopeo**.
- You can use attestation formats such as **in-toto**.
- You can securely store signatures and signed artifacts using OCI repository as a storage backend.

1.2. CONFIGURING TEKTON CHAINS

The Red Hat OpenShift Pipelines Operator installs Tekton Chains by default. You can configure Tekton Chains by modifying the **TektonConfig** custom resource; the Operator automatically applies the changes that you make in this custom resource.

To edit the custom resource, use the following command:

```
$ oc edit TektonConfig config
```

The custom resource includes a **chain:** array. You can add any supported configuration parameters to this array, as shown in the following example:

```
apiVersion: operator.tekton.dev/v1alpha1
kind: TektonConfig
metadata:
  name: config
spec:
  addon: {}
  chain:
    artifacts.taskrun.format: tekton
  config: {}
```

1.2.1. Supported parameters for Tekton Chains configuration

Cluster administrators can use various supported parameter keys and values to configure specifications about task runs, OCI images, and storage.

1.2.1.1. Supported parameters for task run artifacts

Table 1.1. Chains configuration: Supported parameters for task run artifacts

Key	Description	Supported values	Default value
artifacts.taskrun.format	The format for storing task run payloads.	**in-toto**, **slsa/v1**	**in-toto**
artifacts.taskrun.storage	The storage backend for task run signatures. You can specify multiple backends as a comma-separated list, such as **“tekton,oci”**. To disable storing task run artifacts, provide an empty string “”.	**tekton**, **oci**, **gcs**, **docdb**, **grafeas**	**tekton**
artifacts.taskrun.signer	The signature backend for signing task run payloads.	**x509**,**kms**	**x509**

NOTE

slsa/v1 is an alias of **in-toto** for backwards compatibility.

1.2.1.2. Supported parameters for pipeline run artifacts

Table 1.2. Chains configuration: Supported parameters for pipeline run artifacts

Parameter	Description	Supported values	Default value
artifacts.pipelinerun.format	The format for storing pipeline run payloads.	**in-toto**, **slsa/v1**	**in-toto**
artifacts.pipelinerun.storage	The storage backend for storing pipeline run signatures. You can specify multiple backends as a comma-separated list, such as **“tekton,oci”**. To disable storing pipeline run artifacts, provide an empty string “”.	**tekton**, **oci**, **gcs**, **docdb**, **grafeas**	**tekton**
artifacts.pipelinerun.signer	The signature backend for signing pipeline run payloads.	**x509**, **kms**	**x509**

Parameter	Description	Supported values	Default value
artifacts.pipelinerun.enable-deep-inspection	When this parameter is **true**, Tekton Chains records the results of the child task runs of a pipeline run. When this parameter is **false**, Tekton Chains records the results of the pipeline run, but not of its child task runs.	**"true", "false"**	**"false"**

NOTE

- **slsa/v1** is an alias of **in-toto** for backwards compatibility.
- For the **grafeas** storage backend, only Container Analysis is supported. You can not configure the **grafeas** server address in the current version of Tekton Chains.

1.2.1.3. Supported parameters for OCI artifacts

Table 1.3. Chains configuration: Supported parameters for OCI artifacts

Parameter	Description	Supported values	Default value
artifacts.oci.format	The format for storing OCI payloads.	**simplesigning**	**simplesigning**
artifacts.oci.storage	The storage backend for storing OCI signatures. You can specify multiple backends as a comma-separated list, such as **“oci,tekton”**. To disable storing OCI artifacts, provide an empty string “”.	**tekton**, **oci**, **gcs**, **docdb**, **grafeas**	**oci**
artifacts.oci.signer	The signature backend for signing OCI payloads.	**x509**, **kms**	**x509**

1.2.1.4. Supported parameters for KMS signers

Table 1.4. Chains configuration: Supported parameters for KMS signers

Parameter	Description	Supported values	Default value
signers.kms.kmsref	The URI reference to a KMS service to use in **kms** signers.	Supported schemes: **gcpkms:**//, **awskms:**//, **azurekms:**//, **hashivault:**//. See KMS Support in the Sigstore documentation for more details.	

1.2.1.5. Supported parameters for storage

Table 1.5. Chains configuration: Supported parameters for storage

Parameter	Description	Supported values	Default value
storage.gcs.bucket	The GCS bucket for storage		
storage.oci.repository	The OCI repository for storing OCI signatures and attestation.	If you configure one of the artifact storage backends to **oci** and do not define this key, Tekton Chains stores the attestation alongside the stored OCI artifact itself. If you define this key, the attestation is not stored alongside the OCI artifact and is instead stored in the designated location. See the cosign documentation for additional information.	
builder.id	The builder ID to set for in-toto attestations		**https://tekton.dev/chains/v2**

Parameter	Description	Supported values	Default value
builddefinition.buildtype	The build type for in-toto attestation. When this parameter is **https://tekton.dev/chains/v2/slsa**, Tekton Chains records in-toto attestations in strict conformance with the SLSA v1.0 specification. When this parameter is **https://tekton.dev/chains/v2/slsa-tekton**, Tekton Chains records in-toto attestations with additional information, such as the labels and annotations in each **TaskRun** and **PipelineRun** object, and also adds each task in a **PipelineRun** object under **resolvedDependencies**.	**https://tekton.dev/chains/v2/slsa**,**https://tekton.dev/chains/v2/slsa-tekton**	**https://tekton.dev/chains/v2/slsa**

If you enable the **docdb** storage method is for any artifacts, configure docstore storage options. For more information about the go-cloud docstore URI format, see the docstore package documentation. Red Hat OpenShift Pipelines supports the following docstore services:

- **firestore**
- **dynamodb**

Table 1.6. Chains configuration: Supported parameters for**docstore** storage

Parameter	Description	Supported values	Default value
storage.docdb.url	The go-cloud URI reference to a **docstore** collection. Used if the **docdb** storage method is enabled for any artifacts.	**firestore://projects/[PROJECT]/databases/(default)/documents/[COLLECTION]?name_field=name**	

If you enable the **grafeas** storage method for any artifacts, configure Grafeas storage options. For more information about Grafeas notes and occurrences, see Grafeas concepts.

To create occurrences, Red Hat OpenShift Pipelines must first create notes that are used to link occurrences. Red Hat OpenShift Pipelines creates two types of occurrences: **ATTESTATION** Occurrence and **BUILD** Occurrence.

Red Hat OpenShift Pipelines uses the configurable **noteid** as the prefix of the note name. It appends the suffix **-simplesigning** for the **ATTESTATION** note and the suffix **-intoto** for the **BUILD** note. If the **noteid** field is not configured, Red Hat OpenShift Pipelines uses **tekton-<NAMESPACE>** as the prefix.

Table 1.7. Chains configuration: Supported parameters for Grafeas storage

Parameter	Description	Supported values	Default value
storage.grafeas.projectid	The OpenShift Container Platform project in which the Grafeas server for storing occurrences is located.		
storage.grafeas.noteid	Optional: the prefix to use for the name of all created notes.	A string without spaces.	
storage.grafeas.notehint	Optional: the **human_readable_name** field for the Grafeas **ATTESTATION** note.		**This attestation note was generated by Tekton Chains**

Optionally, you can enable additional uploads of binary transparency attestations.

Table 1.8. Chains configuration: Supported parameters for transparency attestation storage

Parameter	Description	Supported values	Default value
transparency.enabled	Enable or disable automatic binary transparency uploads.	**true**, **false**, **manual**	**false**
transparency.url	The URL for uploading binary transparency attestations, if enabled.		**https://rekor.sigstore.dev**

NOTE

If you set **transparency.enabled** to **manual**, only task runs and pipeline runs with the following annotation are uploaded to the transparency log:

```
chains.tekton.dev/transparency-upload: "true"
```

If you configure the **x509** signature backend, you can optionally enable keyless signing with Fulcio.

Table 1.9. Chains configuration: Supported parameters for**x509** keyless signing with Fulcio

Parameter	Description	Supported values	Default value
signers.x509.fulcio.enabled	Enable or disable requesting automatic certificates from Fulcio.	**true**, **false**	**false**
signers.x509.fulcio.address	The Fulcio address for requesting certificates, if enabled.		**https://v1.fulcio.sigstore.dev**
signers.x509.fulcio.issuer	The expected OIDC issuer.		**https://oauth2.sigstore.dev/auth**
signers.x509.fulcio.provider	The provider from which to request the ID Token.	**google**, **spiffe**, **github**, **filesystem**	Red Hat OpenShift Pipelines attempts to use every provider
signers.x509.identity.token.file	Path to the file containing the ID Token.		
signers.x509.tuf.mirror.url	The URL for the TUF server. **$TUF_URL/root.json** must be present.		**https://sigstore-tuf-root.storage.googleapis.com**

If you configure the **kms** signature backend, set the KMS configuration, including OIDC and Spire, as necessary.

Table 1.10. Chains configuration: Supported parameters for KMS signing

Parameter	Description	Supported values	Default value
signers.kms.auth.address	URI of the KMS server (the value of **VAULT_ADDR**).		
signers.kms.auth.token	Authentication token for the KMS server (the value of **VAULT_TOKEN**).		
signers.kms.auth.oidc.path	The path for OIDC authentication (for example, **jwt** for Vault).		
signers.kms.auth.oidc.role	The role for OIDC authentication.		

Parameter	Description	Supported values	Default value
signers.kms.auth.spire.sock	The URI of the Spire socket for the KMS token (for example, **unix:///tmp/spire-agent/public/api.sock**).		
signers.kms.auth.spire.audience	The audience for requesting a SVID from Spire.		

1.3. SECRETS FOR SIGNING DATA IN TEKTON CHAINS

Cluster administrators can generate a key pair and use Tekton Chains to sign artifacts using a Kubernetes secret. For Tekton Chains to work, a private key and a password for encrypted keys must exist as part of the **signing-secrets** secret in the **openshift-pipelines** namespace.

Currently, Tekton Chains supports the **x509** and **cosign** signature schemes.

NOTE

Use only one of the supported signature schemes.

To use the **x509** signing scheme with Tekton Chains, store the **x509.pem** private key of the **ed25519** or **ecdsa** type in the **signing-secrets** Kubernetes secret.

1.3.1. Signing using cosign

You can use the **cosign** signing scheme with Tekton Chains using the **cosign** tool.

Prerequisites

- You installed the cosign tool.

Procedure

1. Generate the **cosign.key** and **cosign.pub** key pairs by running the following command:

   ```
   $ cosign generate-key-pair k8s://openshift-pipelines/signing-secrets
   ```

   Cosign prompts you for a password and then creates a Kubernetes secret.

2. Store the encrypted **cosign.key** private key and the **cosign.password** decryption password in the **signing-secrets** Kubernetes secret. Ensure that the private key is stored as an encrypted PEM file of the **ENCRYPTED COSIGN PRIVATE KEY** type.

1.3.2. Signing using skopeo

You can generate keys using the **skopeo** tool and use them in the **cosign** signing scheme with Tekton Chains.

Prerequisites

- You installed the skopeo tool.

Procedure

1. Generate a public/private key pair by running the following command:

   ```
   $ skopeo generate-sigstore-key --output-prefix <mykey> (1)
   ```

   (1) Replace **<mykey>** with a key name of your choice.

   Skopeo prompts you for a passphrase for the private key and then creates the key files named **<mykey>.private** and **<mykey>.pub**.

2. Encode the **<mykey>.pub** file using the **base64** tool by running the following command:

   ```
   $ base64 -w 0 <mykey>.pub > b64.pub
   ```

3. Encode the **<mykey>.private** file using the **base64** tool by running the following command:

   ```
   $ base64 -w 0 <mykey>.private > b64.private
   ```

4. Encode the passhprase using the **base64** tool by running the following command:

   ```
   $ echo -n '<passphrase>' | base64 -w 0 > b64.passphrase (1)
   ```

   (1) Replace **<passphrase>** with the passphrase that you used for the key pair.

5. Create the **signing-secrets** secret in the **openshift-pipelines** namespace by running the following command:

   ```
   $ oc create secret generic signing-secrets -n openshift-pipelines
   ```

6. Edit the **signing-secrets** secret by running the following command:

   ```
   $ oc edit secret -n openshift-pipelines signing-secrets
   ```

   Add the encoded keys in the data of the secret in the following way:

   ```
   apiVersion: v1
   data:
     cosign.key: <Encoded <mykey>.private> (1)
     cosign.password: <Encoded passphrase> (2)
     cosign.pub: <Encoded <mykey>.pub> (3)
   immutable: true
   kind: Secret
   metadata:
   ```

```
  name: signing-secrets
# ...
type: Opaque
```

1 Replace **<Encoded <mykey>.private>** with the content of the **b64.private** file.

2 Replace **<Encoded passphrase>** with the content of the **b64.passphrase** file.

3 Replace **<Encoded <mykey>.pub>** with the content of the **b64.pub** file.

1.3.3. Resolving the "secret already exists" error

If the **signing-secret** secret is already populated, the command to create this secret might output the following error message:

```
Error from server (AlreadyExists): secrets "signing-secrets" already exists
```

You can resolve this error by deleting the secret.

Procedure

1. Delete the **signing-secret** secret by running the following command:

```
$ oc delete secret signing-secrets -n openshift-pipelines
```

2. Re-create the key pairs and store them in the secret using your preferred signing scheme.

1.4. AUTHENTICATING TO AN OCI REGISTRY

Before pushing signatures to an OCI registry, cluster administrators must configure Tekton Chains to authenticate with the registry. The Tekton Chains controller uses the same service account under which the task runs execute. To set up a service account with the necessary credentials for pushing signatures to an OCI registry, perform the following steps:

Procedure

1. Set the namespace and name of the Kubernetes service account.

```
$ export NAMESPACE=<namespace> 1
$ export SERVICE_ACCOUNT_NAME=<service_account> 2
```

1 The namespace associated with the service account.

2 The name of the service account.

2. Create a Kubernetes secret.

```
$ oc create secret registry-credentials \
  --from-file=.dockerconfigjson \ 1
  --type=kubernetes.io/dockerconfigjson \
  -n $NAMESPACE
```

1 Substitute with the path to your Docker config file. Default path is ~/**.docker/config.json**.

3. Give the service account access to the secret.

```
$ oc patch serviceaccount $SERVICE_ACCOUNT_NAME \
  -p "{\"imagePullSecrets\": [{\"name\": \"registry-credentials\"}]}" -n $NAMESPACE
```

If you patch the default **pipeline** service account that Red Hat OpenShift Pipelines assigns to all task runs, the Red Hat OpenShift Pipelines Operator will override the service account. As a best practice, you can perform the following steps:

a. Create a separate service account to assign to user's task runs.

```
$ oc create serviceaccount <service_account_name>
```

b. Associate the service account to the task runs by setting the value of the **serviceaccountname** field in the task run template.

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: build-push-task-run-2
spec:
  taskRunTemplate:
    serviceAccountName: build-bot 1
  taskRef:
    name: build-push
...
```

1 Substitute with the name of the newly created service account.

1.5. CREATING AND VERIFYING TASK RUN SIGNATURES WITHOUT ANY ADDITIONAL AUTHENTICATION

To verify signatures of task runs using Tekton Chains with any additional authentication, perform the following tasks:

- Create an encrypted x509 key pair and save it as a Kubernetes secret.
- Configure the Tekton Chains backend storage.
- Create a task run, sign it, and store the signature and the payload as annotations on the task run itself.
- Retrieve the signature and payload from the signed task run.
- Verify the signature of the task run.

Prerequisites

Ensure that the following components are installed on the cluster:

- Red Hat OpenShift Pipelines Operator

- Tekton Chains
- Cosign

Procedure

1. Create an encrypted x509 key pair and save it as a Kubernetes secret. For more information about creating a key pair and saving it as a secret, see "Signing secrets in Tekton Chains".

2. In the Tekton Chains configuration, disable the OCI storage, and set the task run storage and format to **tekton**. In the **TektonConfig** custom resource set the following values:

```
apiVersion: operator.tekton.dev/v1alpha1
kind: TektonConfig
metadata:
  name: config
spec:
# ...
    chain:
      artifacts.oci.storage: ""
      artifacts.taskrun.format: tekton
      artifacts.taskrun.storage: tekton
# ...
```

   For more information about configuring Tekton Chains using the **TektonConfig** custom resource, see "Configuring Tekton Chains".

3. To restart the Tekton Chains controller to ensure that the modified configuration is applied, enter the following command:

```
$ oc delete po -n openshift-pipelines -l app=tekton-chains-controller
```

4. Create a task run by entering the following command:

```
$ oc create -f
https://raw.githubusercontent.com/tektoncd/chains/main/examples/taskruns/task-output-
image.yaml (1)
```

   (1) Replace the example URI with the URI or file path pointing to your task run.

   Example output

```
taskrun.tekton.dev/build-push-run-output-image-qbjvh created
```

5. Check the status of the steps by entering the following command. Wait until the process finishes.

```
$ tkn tr describe --last
```

   Example output

```
[...truncated output...]
NAME                   STATUS
```

```
 ∙ create-dir-builtimage-9467f   Completed
 ∙ git-source-sourcerepo-p2sk8   Completed
 ∙ build-and-push                Completed
 ∙ echo                          Completed
 ∙ image-digest-exporter-xlkn7   Completed
```

6. To retrieve the signature from the object stored as **base64** encoded annotations, enter the following commands:

```
$ tkn tr describe --last -o jsonpath="{.metadata.annotations.chains\.tekton\.dev/signature-taskrun-$TASKRUN_UID}" | base64 -d > sig
```

```
$ export TASKRUN_UID=$(tkn tr describe --last -o  jsonpath='{.metadata.uid}')
```

7. To verify the signature using the public key that you created, enter the following command:

```
$ cosign verify-blob-attestation --insecure-ignore-tlog --key path/to/cosign.pub --signature sig --type slsaprovenance --check-claims=false /dev/null (1)
```

(1) Replace **path/to/cosign.pub** with the path name of the public key file.

Example output

```
Verified OK
```

1.5.1. Additional resources

- Section 1.3, “Secrets for signing data in Tekton Chains”
- Section 1.2, “Configuring Tekton Chains”

1.6. USING TEKTON CHAINS TO SIGN AND VERIFY IMAGE AND PROVENANCE

Cluster administrators can use Tekton Chains to sign and verify images and provenances, by performing the following tasks:

- Create an encrypted x509 key pair and save it as a Kubernetes secret.
- Set up authentication for the OCI registry to store images, image signatures, and signed image attestations.
- Configure Tekton Chains to generate and sign provenance.
- Create an image with Kaniko in a task run.
- Verify the signed image and the signed provenance.

Prerequisites

Ensure that the following are installed on the cluster:

- Red Hat OpenShift Pipelines Operator
- Tekton Chains
- Cosign
- Rekor
- jq

Procedure

1. Create an encrypted x509 key pair and save it as a Kubernetes secret:

   ```
   $ cosign generate-key-pair k8s://openshift-pipelines/signing-secrets
   ```

   Provide a password when prompted. Cosign stores the resulting private key as part of the **signing-secrets** Kubernetes secret in the **openshift-pipelines** namespace, and writes the public key to the **cosign.pub** local file.

2. Configure authentication for the image registry.

   a. To configure the Tekton Chains controller for pushing signature to an OCI registry, use the credentials associated with the service account of the task run. For detailed information, see the "Authenticating to an OCI registry" section.

   b. To configure authentication for a Kaniko task that builds and pushes image to the registry, create a Kubernetes secret of the docker **config.json** file containing the required credentials.

   ```
   $ oc create secret generic <docker_config_secret_name> \ 1
     --from-file <path_to_config.json> 2
   ```

   1 Substitute with the name of the docker config secret.

   2 Substitute with the path to docker **config.json** file.

3. Configure Tekton Chains by setting the **artifacts.taskrun.format**, **artifacts.taskrun.storage**, and **transparency.enabled** parameters in the **chains-config** object:

   ```
   $ oc patch configmap chains-config -n openshift-pipelines -p='{"data":
   {"artifacts.taskrun.format": "in-toto"}}'

   $ oc patch configmap chains-config -n openshift-pipelines -p='{"data":
   {"artifacts.taskrun.storage": "oci"}}'

   $ oc patch configmap chains-config -n openshift-pipelines -p='{"data":
   {"transparency.enabled": "true"}}'
   ```

4. Start the Kaniko task.

   a. Apply the Kaniko task to the cluster.

   ```
   $ oc apply -f examples/kaniko/kaniko.yaml 1
   ```

1 Substitute with the URI or file path to your Kaniko task.

b. Set the appropriate environment variables.

```
$ export REGISTRY=<url_of_registry> 1

$ export DOCKERCONFIG_SECRET_NAME=
<name_of_the_secret_in_docker_config_json> 2
```

1 Substitute with the URL of the registry where you want to push the image.

2 Substitute with the name of the secret in the docker **config.json** file.

c. Start the Kaniko task.

```
$ tkn task start --param IMAGE=$REGISTRY/kaniko-chains --use-param-defaults --
workspace name=source,emptyDir="" --workspace
name=dockerconfig,secret=$DOCKERCONFIG_SECRET_NAME kaniko-chains
```

Observe the logs of this task until all steps are complete. On successful authentication, the final image will be pushed to **$REGISTRY/kaniko-chains**.

5. Wait for a minute to allow Tekton Chains to generate the provenance and sign it, and then check the availability of the **chains.tekton.dev/signed=true** annotation on the task run.

```
$ oc get tr <task_run_name> \ 1
-o json | jq -r .metadata.annotations

{
  "chains.tekton.dev/signed": "true",
  ...
}
```

1 Substitute with the name of the task run.

6. Verify the image and the attestation.

```
$ cosign verify --key cosign.pub $REGISTRY/kaniko-chains

$ cosign verify-attestation --key cosign.pub $REGISTRY/kaniko-chains
```

7. Find the provenance for the image in Rekor.

a. Get the digest of the $REGISTRY/kaniko-chains image. You can search for it ing the task run, or pull the image to extract the digest.

b. Search Rekor to find all entries that match the **sha256** digest of the image.

```
$ rekor-cli search --sha <image_digest> 1

<uuid_1> 2
```

```
<uuid_2> (3)
...
```

(1) Substitute with the **sha256** digest of the image.

(2) The first matching universally unique identifier (UUID).

(3) The second matching UUID.

The search result displays UUIDs of the matching entries. One of those UUIDs holds the attestation.

c. Check the attestation.

```
$ rekor-cli get --uuid <uuid> --format json | jq -r .Attestation | base64 --decode | jq
```

1.7. ADDITIONAL RESOURCES

- Installing OpenShift Pipelines

CHAPTER 2. SETTING UP OPENSHIFT PIPELINES IN THE WEB CONSOLE TO VIEW SOFTWARE SUPPLY CHAIN SECURITY ELEMENTS

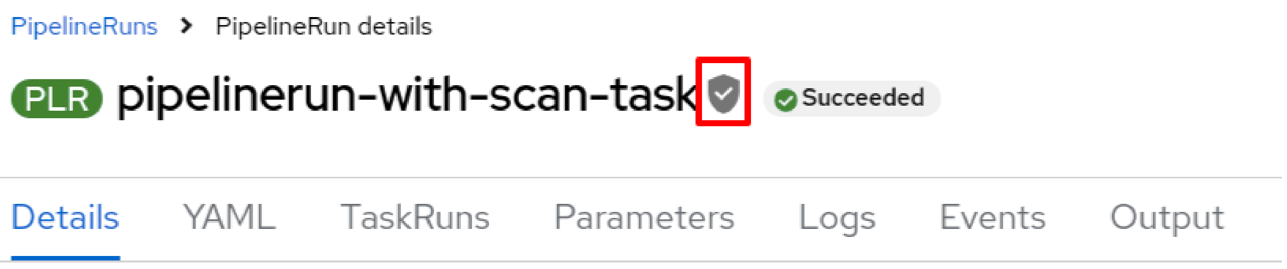
Use the **Developer** or **Administrator** perspective to create or modify a pipeline and view key Software Supply Chain Security elements within a project.

Set up OpenShift Pipelines to view:

- **Project vulnerabilities**: Visual representation of identified vulnerabilities within a project.
- **Software Bill of Materials (SBOMs)**: Download or view detailed listing of PipelineRun components.

Additionally, PipelineRuns that meet Tekton Chains requirement displays signed badges next to their names. This badge indicates that the pipeline run execution results are cryptographically signed and stored securely, for example within an OCI image.

Figure 2.1. The signed badge

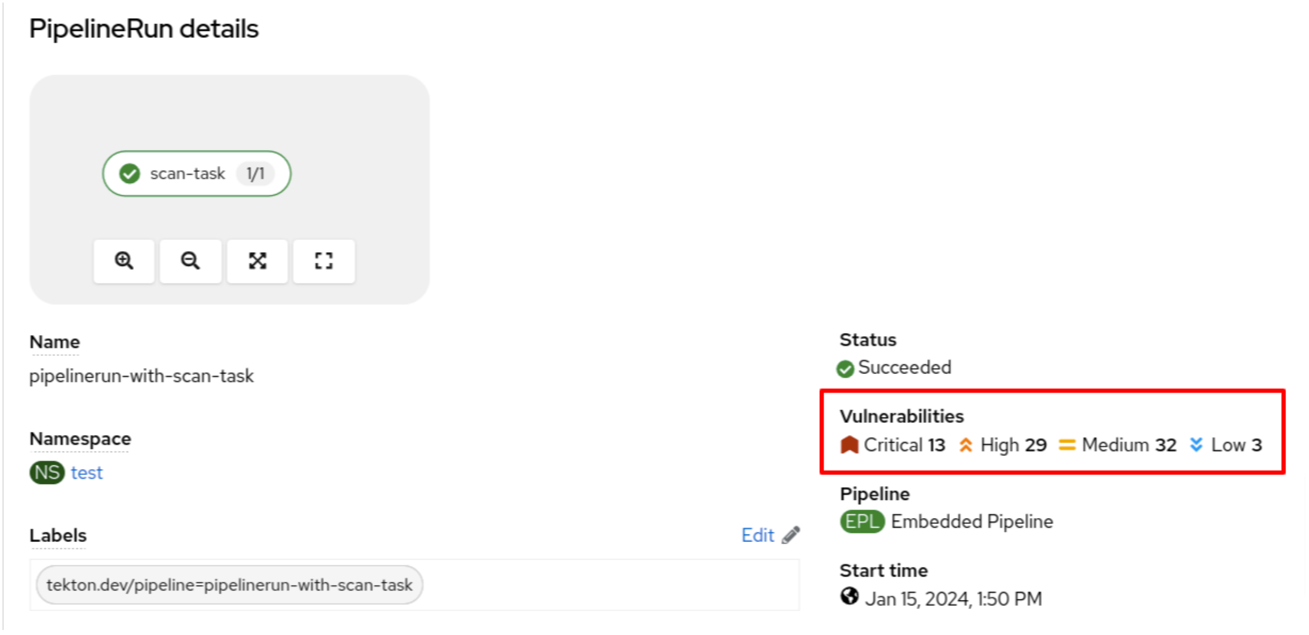


The PipelineRun displays the signed badge next to its name only if you have configured Tekton Chains. For information on configuring Tekton Chains, see Using Tekton Chains for OpenShift Pipelines supply chain security.

2.1. SETTING UP OPENSHIFT PIPELINES TO VIEW PROJECT VULNERABILITIES

The PipelineRun details page provides a visual representation of identified vulnerabilities, categorized by the severity (critical, high, medium, and low). This streamlined view facilitates prioritization and remediation efforts.

Figure 2.2. Viewing vulnerabilities on the **PipelineRun** details page


PipelineRun details

scan-task 1/1

Name
pipelinerun-with-scan-task

Namespace
NS test

Labels
tekton.dev/pipeline=pipelinerun-with-scan-task

Status
Succeeded

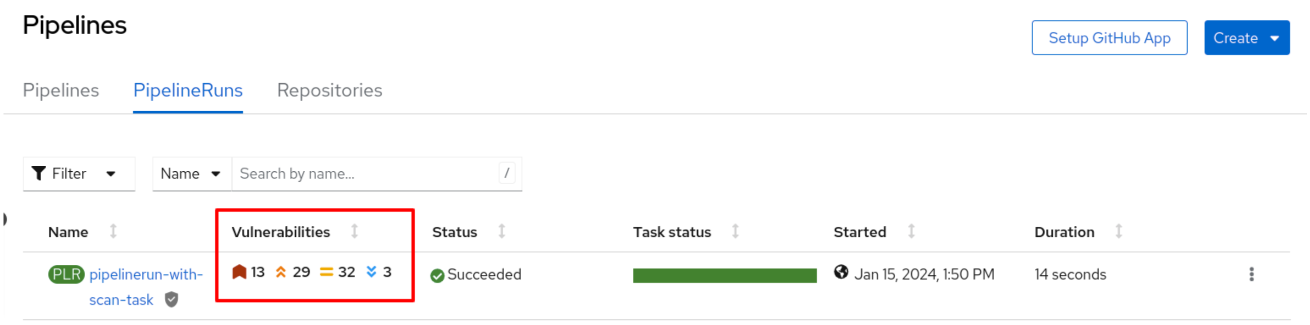
Vulnerabilities
Critical 13 High 29 Medium 32 Low 3

Pipeline
EPL Embedded Pipeline

Start time
Jan 15, 2024, 1:50 PM

You can also review the vulnerabilities in the **Vulnerabilities** column in the pipeline run list view page.

Figure 2.3. Viewing vulnerabilities on the **PipelineRun** list view


Pipelines

Name	Vulnerabilities	Status	Task status	Started	Duration
PLR pipelinerun-with-scan-task	13 29 32 3	Succeeded		Jan 15, 2024, 1:50 PM	14 seconds

NOTE

Visual representation of identified vulnerabilities is available starting from the OpenShift Container Platform version 4.15 release.

Prerequisites

- You have logged in to the web console .
- You have the appropriate roles and permissions in a project to create applications and other workloads in OpenShift Container Platform.
- You have an existing vulnerability scan task.

Procedures

1. In the **Developer** or **Administrator** perspective, switch to the relevant project where you want a visual representation of vulnerabilities.
2. Update your existing vulnerability scan task to ensure that it stores the output in the .json file and then extracts the vulnerability summary in the following format:

```
# The format to extract vulnerability summary (adjust the jq command for different JSON
structures).
jq -rce \
  '{vulnerabilities:{
    critical: (.result.summary.CRITICAL),
    high: (.result.summary.IMPORTANT),
    medium: (.result.summary.MODERATE),
    low: (.result.summary.LOW)
    }}' scan_output.json | tee $(results.SCAN_OUTPUT.path)
```

NOTE

You might need to adjust the jq command for different JSON structures.

a. (Optional) If you do not have a vulnerability scan task, create one in the following format:
Example vulnerability scan task using Roxctl

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: vulnerability-scan (1)
  annotations:
    task.output.location: results (2)
    task.results.format: application/json
    task.results.key: SCAN_OUTPUT (3)
spec:
  params:
    - description: Image to be scanned
      name: image
      type: string
  results:
    - description: CVE result format (4)
      name: SCAN_OUTPUT
  steps:
    - name: roxctl
      image: 'quay.io/lrangine/crda-maven:11.0' (5)
      env:
        - name: ROX_CENTRAL_ENDPOINT
          valueFrom:
            secretKeyRef:
              key: rox_central_endpoint (6)
              name: roxsecrets
        - name: ROX_API_TOKEN
          valueFrom:
            secretKeyRef:
              key: rox_api_token (7)
              name: roxsecrets

      name: roxctl-scan
      script: | (8)
        #!/bin/sh
        curl -k -L -H "Authorization: Bearer $ROX_API_TOKEN"
https://$ROX_CENTRAL_ENDPOINT/api/cli/download/roxctl-linux --output ./roxctl
```

```
        chmod +x ./roxctl
        ./roxctl image scan --insecure-skip-tls-verify -e $ROX_CENTRAL_ENDPOINT --
image $(params.image) --output json  > roxctl_output.json
        jq -rce \
        "{vulnerabilities:{
        critical: (.result.summary.CRITICAL),
        high: (.result.summary.IMPORTANT),
        medium: (.result.summary.MODERATE),
        low: (.result.summary.LOW)
        }}" roxctl_output.json | tee $(results.SCAN_OUTPUT.path)
```

1. The name of your task.
2. The location for storing the task outputs.
3. The naming convention of the scan task result. A valid naming convention must end with the **SCAN_OUTPUT** string. For example, SCAN_OUTPUT, MY_CUSTOM_SCAN_OUTPUT, or ACS_SCAN_OUTPUT.
4. The description of the result.
5. The location of the container image to run the scan tool.
6. The **rox_central_endpoint** key obtained from Advanced Cluster Security for Kubernetes (ACS).
7. The **rox_api_token** obtained from ACS.
8. The shell script performs the vulnerability scanning and sets the scan output in the Task run results.

NOTE

This is an example configuration. Modify the values according to your specific scanning tool to set results in the expected format.

3. Update an appropriate Pipeline to add vulnerabilities specifications in the following format:

```
...
spec:
  results:
    - description: The common vulnerabilities and exposures (CVE) result
      name: SCAN_OUTPUT
      value: $(tasks.vulnerability-scan.results.SCAN_OUTPUT)
```

Verification

- Navigate to the **PipelineRun** details page and review the **Vulnerabilities** row for a visual representation of identified vulnerabilities.
- Alternatively, you can navigate to the **PipelineRun** list view page, and review the **Vulnerabilities** column.

Additional resources

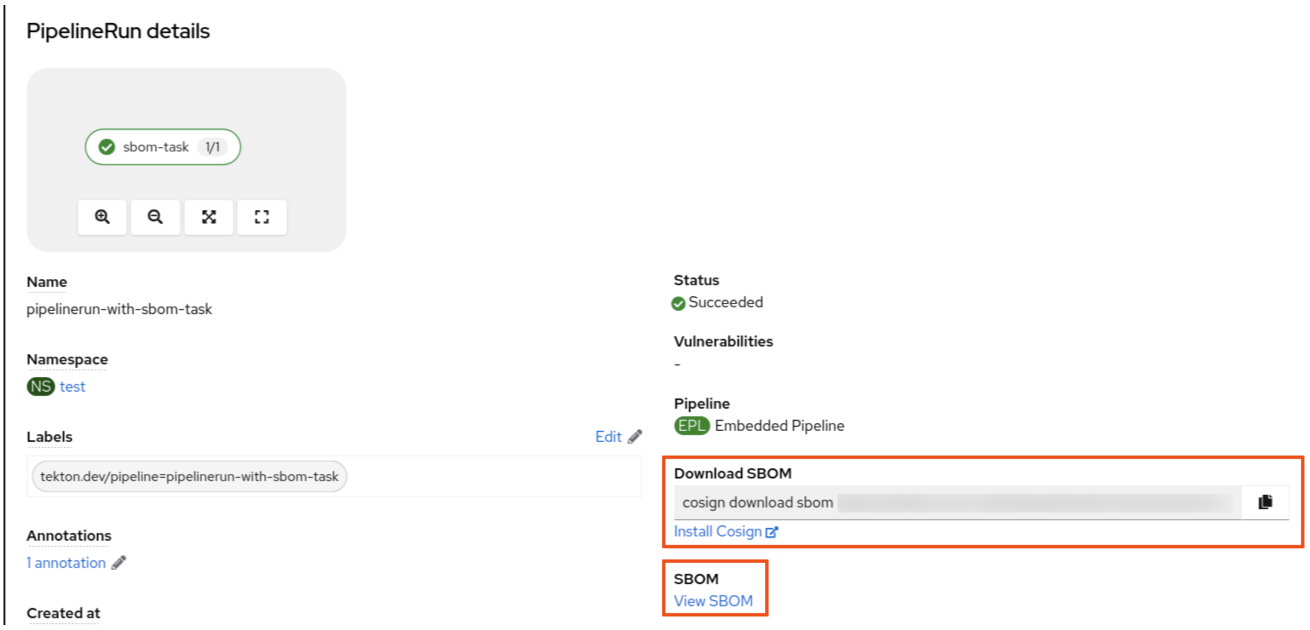
- Review Red Hat Trusted Application Pipeline (RHTAP) for a customizable end-to-end solution for building applications
- Getting Started with Red Hat Trusted Application Pipeline

2.2. SETTING UP OPENSHIFT PIPELINES TO DOWNLOAD OR VIEW SBOMS

The **PipelineRun** details page provides an option to download or view Software Bill of Materials (SBOMs), enhancing transparency and control within your supply chain. SBOMs lists all the software libraries that a component uses. Those libraries can enable specific functionality or facilitate development.

You can use an SBOM to better understand the composition of your software, identify vulnerabilities, and assess the potential impact of any security issues that might arise.

Figure 2.4. Options to download or view SBOMs



PipelineRun details

Name
pipelinerun-with-sbom-task

Namespace
NS test

Labels
Edit
tekton.dev/pipeline=pipelinerun-with-sbom-task

Annotations
1 annotation

Created at

Status
Succeeded

Vulnerabilities
-

Pipeline
EPL Embedded Pipeline

Download SBOM
cosign download sbom [redacted]
Install Cosign

SBOM
View SBOM

Prerequisites

- You have logged in to the web console .
- You have the appropriate roles and permissions in a project to create applications and other workloads in OpenShift Container Platform.

Procedure

1. In the **Developer** or **Administrator** perspective, switch to the relevant project where you want a visual representation of SBOMs.
2. Add a task in the following format to view or download the SBOM information:

   Example SBOM task

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: sbom-task (1)
  annotations:
    task.output.location: results (2)
    task.results.format: application/text
    task.results.key: LINK_TO_SBOM (3)
    task.results.type: external-link (4)
spec:
  results:
    - description: Contains the SBOM link (5)
      name: LINK_TO_SBOM
  steps:
    - name: print-sbom-results
      image: quay.io/image (6)
      script: | (7)
        #!/bin/sh
        syft version
        syft quay.io/<username>/quarkus-demo:v2 --output cyclonedx-json=sbom-image.json
        echo 'BEGIN SBOM'
        cat sbom-image.json
        echo 'END SBOM'
        echo 'quay.io/user/workloads/<namespace>/node-express/node-express:build-8e536-1692702836' | tee $(results.LINK_TO_SBOM.path) (8)
```

1. The name of your task.
2. The location for storing the task outputs.
3. The SBOM task result name. Do not change the name of the SBOM result task.
4. (Optional) Set to open the SBOM in a new tab.
5. The description of the result.
6. The image that generates the SBOM.
7. The script that generates the SBOM image.
8. The SBOM image along with the path name.

3. Update the Pipeline to reference the newly created SBOM task.

```
...
spec:
  tasks:
    - name: sbom-task
      taskRef:
        name: sbom-task (1)
  results:
    - name: IMAGE_URL (2)
      description: url
      value: <oci_image_registry_url> (3)
```

1. The same name as created in Step 2.

2. The name of the result.
3. The OCI image repository URL which contains the **.sbom** images.

4. Rerun the affected OpenShift Pipeline.

2.2.1. Viewing an SBOM in the web UI

Prerequisites

- You have set up OpenShift Pipelines to download or view SBOMs.

Procedure

1. Navigate to the Activity → PipelineRuns tab.
2. For the project whose SBOM you want to view, select its most recent pipeline run.
3. On the **PipelineRun** details page, select **View SBOM**.
   a. You can use your web browser to immediately search the SBOM for terms that indicate vulnerabilities in your software supply chain. For example, try searching for **log4j**.
   b. You can select **Download** to download the SBOM, or **Expand** to view it full-screen.

2.2.2. Downloading an SBOM in the CLI

Prerequisites

- You have installed the Cosign CLI tool.
- You have set up OpenShift Pipelines to download or view SBOMs.

Procedure

1. Open terminal, log in to **Developer** or **Administrator** perspective, and then switch to the relevant project.
2. From the OpenShift web console, copy the **download sbom** command and run it on your terminal.

   Example cosign command

   ```
   $ cosign download sbom quay.io/<workspace>/user-workload@sha256
   ```

   a. (Optional) To view the full SBOM in a searchable format, run the following command to redirect the output:

      Example cosign command

      ```
      $ cosign download sbom quay.io/<workspace>/user-workload@sha256 > sbom.txt
      ```

2.2.3. Reading the SBOM

In the SBOM, as the following sample excerpt shows, you can see four characteristics of each library that a project uses:

- Its author or publisher
- Its name
- Its version
- Its licenses

This information helps you verify that individual libraries are safely-sourced, updated, and compliant.

Example SBOM

```
{
    "bomFormat": "CycloneDX",
    "specVersion": "1.4",
    "serialNumber": "urn:uuid:89146fc4-342f-496b-9cc9-07a6a1554220",
    "version": 1,
    "metadata": {
        ...
    },
    "components": [
        {
            "bom-ref": "pkg:pypi/flask@2.1.0?package-id=d6ad7ed5aac04a8",
            "type": "library",
            "author": "Armin Ronacher <armin.ronacher@active-4.com>",
            "name": "Flask",
            "version": "2.1.0",
            "licenses": [
                {
                    "license": {
                        "id": "BSD-3-Clause"
                    }
                }
            ],
            "cpe": "cpe:2.3:a:armin-ronacher:python-Flask:2.1.0:*:*:*:*:*:*:*",
            "purl": "pkg:pypi/Flask@2.1.0",
            "properties": [
                {
                    "name": "syft:package:foundBy",
                    "value": "python-package-cataloger"
                    ...
```

2.3. ADDITIONAL RESOURCES

- Working with Red Hat OpenShift Pipelines in the web console

CHAPTER 3. CONFIGURING THE SECURITY CONTEXT FOR PODS

The default service account for pods that OpenShift Pipelines starts is **pipeline**. The security context constraint (SCC) associated with the **pipeline** service account is **pipelines-scc**. The **pipelines-scc** SCC is based the **anyuid** SCC, with minor differences as defined in the following YAML specification:

Example pipelines-scc.yaml snippet

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
# ...
allowedCapabilities:
  - SETFCAP
# ...
fsGroup:
  type: MustRunAs
# ...
```

In addition, the **Buildah** cluster task, shipped as part of OpenShift Pipelines, uses **vfs** as the default storage driver.

You can configure the security context for pods that OpenShift Pipelines creates for pipeline runs and task runs. You can make the following changes:

- Change the default and maximum SCC for all pods
- Change the default SCC for pods created for pipeline runs and task runs in a particular namespace
- Configure a particular pipeline run or task run to use a custom SCC and service account

NOTE

The simplest way to run **buildah** that ensures all images can build is to run it as root in a pod with the **privileged** SCC. For instructions about running **buildah** with more restrictive security settings, see Building of container images using Buildah as a non-root user.

3.1. CONFIGURING THE DEFAULT AND MAXIMUM SCC FOR PODS THAT OPENSHIFT PIPELINES CREATES

You can configure the default security context constraint (SCC) for all pods that OpenShift Pipelines creates for task runs and pipeline runs. You can also configure the maximum SCC, which is the least restrictive SCC that can be configured for these pods in any namespace.

Procedure

- Edit the **TektonConfig** custom resource (CR) by entering the following command:

  ```
  $ oc edit TektonConfig config
  ```

  Set the default and maximum SCC in the spec, as in the following example:

```
apiVersion: operator.tekton.dev/v1alpha1
kind: TektonConfig
metadata:
  name: config
spec:
# ...
  platforms:
    openshift:
      scc:
        default: "restricted-v2" 1
        maxAllowed: "privileged" 2
```

1. **spec.platforms.openshift.scc.default** specifies the default SCC that OpenShift Pipelines attaches to the service account (SA) used for workloads, which is, by default, the **pipeline** SA. This SCC is used for all pipeline run and task run pods.

2. **spec.platforms.openshift.scc.maxAllowed** specifies the least restrictive SCC that you can configure for pipeline run and task run pods in any namespace. This setting does not apply when you configure a custom SA and SCC in a particular pipeline run or task run.

Additional resources

- Changing the default service account for OpenShift Pipelines

3.2. CONFIGURING THE SCC FOR PODS IN A NAMESPACE

You can configure the security context constraint (SCC) for all pods that OpenShift Pipelines creates for pipeline runs and task runs that you create in a particular namespace. This SCC must not be less restrictive than the maximum SCC that you configured using the **TektonConfig** CR, in the **spec.platforms.openshift.scc.maxAllowed** spec.

Procedure

- Set the **operator.tekton.dev/scc** annotation for the namespace to the name of the SCC.

  Example namespace annotation for configuring the SCC for OpenShift Pipelines pods

```
apiVersion: v1
kind: Namespace
metadata:
  name: test-namespace
  annotations:
    operator.tekton.dev/scc: nonroot
```

3.3. RUNNING PIPELINE RUN AND TASK RUN BY USING A CUSTOM SCC AND A CUSTOM SERVICE ACCOUNT

When using the **pipelines-scc** security context constraint (SCC) associated with the default **pipelines** service account, the pipeline run and task run pods might face timeouts. This happens because in the default **pipelines-scc** SCC, the **fsGroup.type** parameter is set to **MustRunAs**.

NOTE

For more information about pod timeouts, see BZ#1995779.

To avoid pod timeouts, you can create a custom SCC with the **fsGroup.type** parameter set to **RunAsAny**, and associate it with a custom service account.

NOTE

As a best practice, use a custom SCC and a custom service account for pipeline runs and task runs. This approach allows greater flexibility and does not break the runs when the defaults are modified during an upgrade.

Procedure

1. Define a custom SCC with the **fsGroup.type** parameter set to **RunAsAny**:

   Example: Custom SCC

```
apiVersion: security.openshift.io/v1
kind: SecurityContextConstraints
metadata:
  annotations:
    kubernetes.io/description: my-scc is a close replica of anyuid scc. pipelines-scc has
fsGroup - RunAsAny.
  name: my-scc
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true
allowPrivilegedContainer: false
allowedCapabilities: null
defaultAddCapabilities: null
fsGroup:
  type: RunAsAny
groups:
- system:cluster-admins
priority: 10
readOnlyRootFilesystem: false
requiredDropCapabilities:
- MKNOD
runAsUser:
  type: RunAsAny
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
```

2. Create the custom SCC:

   Example: Create the `my-scc` SCC

   ```
   $ oc create -f my-scc.yaml
   ```

3. Create a custom service account:

   Example: Create a `fsgroup-runasany` service account

   ```
   $ oc create serviceaccount fsgroup-runasany
   ```

4. Associate the custom SCC with the custom service account:

   Example: Associate the `my-scc` SCC with the `fsgroup-runasany` service account

   ```
   $ oc adm policy add-scc-to-user my-scc -z fsgroup-runasany
   ```

   If you want to use the custom service account for privileged tasks, you can associate the **privileged** SCC with the custom service account by running the following command:

   Example: Associate the `privileged` SCC with the `fsgroup-runasany` service account

   ```
   $ oc adm policy add-scc-to-user privileged -z fsgroup-runasany
   ```

5. Use the custom service account in the pipeline run and task run:

   Example: Pipeline run YAML with `fsgroup-runasany` custom service account

   ```
   apiVersion: tekton.dev/v1
   kind: PipelineRun
   metadata:
     name: <pipeline-run-name>
   spec:
     pipelineRef:
       name: <pipeline-cluster-task-name>
     taskRunTemplate:
       serviceAccountName: 'fsgroup-runasany'
   ```

   Example: Task run YAML with `fsgroup-runasany` custom service account

   ```
   apiVersion: tekton.dev/v1
   kind: TaskRun
   metadata:
     name: <task-run-name>
   spec:
     taskRef:
   ```

```
    name: <cluster-task-name>
  taskRunTemplate:
    serviceAccountName: 'fsgroup-runasany'
```

3.4. ADDITIONAL RESOURCES

- Managing security context constraints.

CHAPTER 4. SECURING WEBHOOKS WITH EVENT LISTENERS

As an administrator, you can secure webhooks with event listeners. After creating a namespace, you enable HTTPS for the **Eventlistener** resource by adding the **operator.tekton.dev/enable-annotation=enabled** label to the namespace. Then, you create a **Trigger** resource and a secured route using the re-encrypted TLS termination.

Triggers in Red Hat OpenShift Pipelines support insecure HTTP and secure HTTPS connections to the **Eventlistener** resource. HTTPS secures connections within and outside the cluster.

Red Hat OpenShift Pipelines runs a **tekton-operator-proxy-webhook** pod that watches for the labels in the namespace. When you add the label to the namespace, the webhook sets the **service.beta.openshift.io/serving-cert-secret-name=<secret_name>** annotation on the **EventListener** object. This, in turn, creates secrets and the required certificates.

```
service.beta.openshift.io/serving-cert-secret-name=<secret_name>
```

In addition, you can mount the created secret into the **Eventlistener** pod to secure the request.

4.1. PROVIDING SECURE CONNECTION WITH OPENSHIFT ROUTES

To create a route with the re-encrypted TLS termination, run:

```
$ oc create route reencrypt --service=<svc-name> --cert=tls.crt --key=tls.key --ca-cert=ca.crt --
hostname=<hostname>
```

Alternatively, you can create a re-encrypted TLS termination YAML file to create a secure route.

Example re-encrypt TLS termination YAML to create a secure route

```
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: route-passthrough-secured 1
spec:
  host: <hostname>
  to:
    kind: Service
    name: frontend 2
  tls:
    termination: reencrypt 3
    key: [as in edge termination]
    certificate: [as in edge termination]
    caCertificate: [as in edge termination]
    destinationCACertificate: |- 4
      -----BEGIN CERTIFICATE-----
      [...]
      -----END CERTIFICATE-----
```

1 2 The name of the object, which is limited to only 63 characters.

3 The termination field is set to **reencrypt**. This is the only required TLS field.

4 This is required for re-encryption. The **destinationCACertificate** field specifies a CA certificate to validate the endpoint certificate, thus securing the connection from the router to the destination

- The service uses a service signing certificate.
- The administrator specifies a default CA certificate for the router, and the service has a certificate signed by that CA.

You can run the **oc create route reencrypt --help** command to display more options.

4.2. CREATING A SAMPLE EVENTLISTENER RESOURCE USING A SECURE HTTPS CONNECTION

This section uses the pipelines-tutorial example to demonstrate creation of a sample EventListener resource using a secure HTTPS connection.

Procedure

1. Create the **TriggerBinding** resource from the YAML file available in the pipelines-tutorial repository:

   ```
   $ oc create -f https://raw.githubusercontent.com/openshift/pipelines-
   tutorial/master/03_triggers/01_binding.yaml
   ```

2. Create the **TriggerTemplate** resource from the YAML file available in the pipelines-tutorial repository:

   ```
   $ oc create -f https://raw.githubusercontent.com/openshift/pipelines-
   tutorial/master/03_triggers/02_template.yaml
   ```

3. Create the **Trigger** resource directly from the pipelines-tutorial repository:

   ```
   $ oc create -f https://raw.githubusercontent.com/openshift/pipelines-
   tutorial/master/03_triggers/03_trigger.yaml
   ```

4. Create an **EventListener** resource using a secure HTTPS connection:

   a. Add a label to enable the secure HTTPS connection to the **Eventlistener** resource:

   ```
   $ oc label namespace <ns-name> operator.tekton.dev/enable-annotation=enabled
   ```

   b. Create the **EventListener** resource from the YAML file available in the pipelines-tutorial repository:

   ```
   $ oc create -f https://raw.githubusercontent.com/openshift/pipelines-
   tutorial/master/03_triggers/04_event_listener.yaml
   ```

   c. Create a route with the re-encrypted TLS termination:

   ```
   $ oc create route reencrypt --service=<svc-name> --cert=tls.crt --key=tls.key --ca-
   cert=ca.crt --hostname=<hostname>
   ```

CHAPTER 5. AUTHENTICATING PIPELINES WITH REPOSITORIES USING SECRETS

Pipelines and tasks can require credentials to authenticate with Git repositories and container repositories. In Red Hat OpenShift Pipelines, you can use secrets to authenticate pipeline runs and task runs that interact with a Git repository or container repository during execution.

A secret for authentication with a Git repository is known as a *Git secret*.

A pipeline run or a task run gains access to the secrets through an associated service account. Alternatively, you can define a workspace in the pipeline or task and bind the secret to the workspace.

5.1. PREREQUISITES

- You installed the **oc** OpenShift command line utility.

5.2. PROVIDING SECRETS USING SERVICE ACCOUNTS

You can use service accounts to provide secrets for authentication with Git repositories and container repositories.

You can associate a secret with a service account. The information in the secret becomes available to the tasks that run under this service account.

5.2.1. Types and annotation of secrets for service accounts

If you provide authentication secrets using service accounts, OpenShift Pipelines supports several secret types. For most of these secret types, you must provide annotations that define the repositories for which the authentication secret is valid.

5.2.1.1. Git authentication secrets

If you provide authentication secrets using service accounts, OpenShift Pipelines supports the following types of secrets for Git authentication:

- **kubernetes.io/basic-auth**: A username and password for basic authentication
- **kubernetes.io/ssh-auth**: Keys for SSH-based authentication

If you provide authentication secrets using service accounts, a Git secret must have one or more annotation keys. The names of each key must begin with **tekton.dev/git-** and the value is the URL of the host for which OpenShift Pipelines must use the credentials in the secret.

In the following example, OpenShift Pipelines uses a **basic-auth** secret to access repositories at **github.com** and **gitlab.com**.

Example: Credentials for basic authentication with multiple Git repositories

```
apiVersion: v1
kind: Secret
metadata:
  name: git-secret-basic
  annotations:
    tekton.dev/git-0: github.com
    tekton.dev/git-1: gitlab.com
type: kubernetes.io/basic-auth
stringData:
  username: <username> (1)
  password: <password> (2)
```

(1) Username for the repository

(2) Password or personal access token for the repository

You can also use **ssh-auth** secret to provide a private key for accessing a Git repository, as in the following example:

Example: Private key for SSH-based authentication

```
apiVersion: v1
kind: Secret
metadata:
  name: git-secret-ssh
  annotations:
    tekton.dev/git-0: https://github.com
type: kubernetes.io/ssh-auth
stringData:
  ssh-privatekey: (1)
```

(1) The content of the SSH private key file.

5.2.1.2. Container registry authentication secrets

If you provide authentication secrets using service accounts, OpenShift Pipelines supports the following types of secrets for container (Docker) registry authentication:

- **kubernetes.io/basic-auth**: A username and password for basic authentication
- **kubernetes.io/dockercfg**: A serialized **~/.dockercfg** file
- **kubernetes.io/dockerconfigjson**: A serialized **~/.docker/config.json** file

If you provide authentication secrets using service accounts, a container registry secret of the **kubernetes.io/basic-auth** type must have one or more annotation keys. The names of each key must begin with **tekton.dev/docker-** and the value is the URL of the host for which OpenShift Pipelines must use the credentials in the secret. This annotation is not required for other types of container registry secrets.

In the following example, OpenShift Pipelines uses a **basic-auth** secret, which relies on a username and password, to access container registries at **quay.io** and **my-registry.example.com**.

Example: Credentials for basic authentication with multiple container repositories

```
apiVersion: v1
kind: Secret
metadata:
  name: docker-secret-basic
  annotations:
    tekton.dev/docker-0: quay.io
    tekton.dev/docker-1: my-registry.example.com
type: kubernetes.io/basic-auth
stringData:
  username: <username> 1
  password: <password> 2
```

1. Username for the registry
2. Password or personal access token for the registry

You can create **kubernetes.io/dockercfg** and **kubernetes.io/dockerconfigjson** secrets from an existing configuration file, as in the following example:

Example: Command for creating a secret for authenticating to a container repository from an existing configuration file

```
$ oc create secret generic docker-secret-config \
    --from-file=config.json=/home/user/.docker/config.json \
    --type=kubernetes.io/dockerconfigjson
```

You can also use the **oc** command line utility to create **kubernetes.io/dockerconfigjson** secrets from credentials, as in the following example:

Example: Command for creating a secret for authenticating to a container repository from credentials

```
$ oc create secret docker-registry docker-secret-config \
  --docker-email=<email> \ 1
  --docker-username=<username> \ 2
  --docker-password=<password> \ 3
  --docker-server=my-registry.example.com:5000 4
```

1. Email address for the registry
2. Username for the registry
3. Password or personal access token for the registry
4. The host name and port for the registry

5.2.2. Configuring basic authentication for Git using a service account

For a pipeline to retrieve resources from password-protected repositories, you can configure the basic authentication for that pipeline.

NOTE

Consider using SSH-based authentication rather than basic authentication.

To configure basic authentication for a pipeline, create a basic authentication secret, associate this secret with a service account, and associate this service account with a **TaskRun** or **PipelineRun** resource.

NOTE

For GitHub, authentication using a plain password is deprecated. Instead, use a personal access token.

Procedure

1. Create the YAML manifest for the secret in the **secret.yaml** file. In this manifest, specify the username and password or GitHub personal access token to access the target Git repository.

```
apiVersion: v1
kind: Secret
metadata:
  name: basic-user-pass 1
  annotations:
    tekton.dev/git-0: https://github.com
type: kubernetes.io/basic-auth
stringData:
  username: <username> 2
  password: <password> 3
```

1 Name of the secret. In this example, **basic-user-pass**.

2 Username for the Git repository.

3 Password or personal access token for the Git repository.

2. Create the YAML manifest for the service account in the **serviceaccount.yaml** file. In this manifest, associate the secret with the service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-bot 1
secrets:
  - name: basic-user-pass 2
```

1 Name of the service account. In this example, **build-bot**.

2 Name of the secret. In this example, **basic-user-pass**.

3. Create the YAML manifest for the task run or pipeline run in the **run.yaml** file and associate the service account with the task run or pipeline run. Use one of the following examples:

   - Associate the service account with a **TaskRun** resource:

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: build-push-task-run-2 (1)
spec:
  taskRunTemplate:
    serviceAccountName: build-bot (2)
  taskRef:
    name: build-push (3)
```

(1) Name of the task run. In this example, **build-push-task-run-2**.

(2) Name of the service account. In this example, **build-bot**.

(3) Name of the task. In this example, **build-push**.

- Associate the service account with a **PipelineRun** resource:

```
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: demo-pipeline (1)
  namespace: default
spec:
  taskRunTemplate:
    serviceAccountName: build-bot (2)
  pipelineRef:
    name: demo-pipeline (3)
```

(1) Name of the pipeline run. In this example, **demo-pipeline**.

(2) Name of the service account. In this example, **build-bot**.

(3) Name of the pipeline. In this example, **demo-pipeline**.

4. Apply the YAML manifests that you created by entering the following command:

```
$ oc apply --filename secret.yaml,serviceaccount.yaml,run.yaml
```

5.2.3. Configuring SSH authentication for Git using a service account

For a pipeline to retrieve resources from repositories configured with SSH keys, you must configure the SSH-based authentication for that pipeline.

To configure SSH-based authentication for a pipeline, create an authentication secret with the SSH private key, associate this secret with a service account, and associate this service account with a **TaskRun** or **PipelineRun** resource.

Procedure

1. Generate an SSH private key, or copy an existing private key, which is usually available in the **~/.ssh/id_rsa** file.

2. Create the YAML manifest for the secret in the **secret.yaml** file. In this manifest, set the value of **ssh-privatekey** to the content of the SSH private key file, and set the value of **known_hosts** to the content of the known hosts file.

```
apiVersion: v1
kind: Secret
metadata:
  name: ssh-key (1)
  annotations:
    tekton.dev/git-0: github.com
type: kubernetes.io/ssh-auth
stringData:
  ssh-privatekey: (2)
  known_hosts: (3)
```

(1) Name of the secret containing the SSH private key. In this example, **ssh-key**.

(2) The content of the SSH private key file.

(3) The content of the known hosts file.

IMPORTANT

If you omit the known hosts file, OpenShift Pipelines accepts the public key of any server.

3. Optional: Specify a custom SSH port by adding **:<port_number>** to the end of the annotation value. For example, **tekton.dev/git-0: github.com:2222**.

4. Create the YAML manifest for the service account in the **serviceaccount.yaml** file. In this manifest, associate the secret with the service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: build-bot (1)
secrets:
  - name: ssh-key (2)
```

(1) Name of the service account. In this example, **build-bot**.

(2) Name of the secret containing the SSH private key. In this example, **ssh-key**.

5. In the **run.yaml** file, associate the service account with a task run or a pipeline run. Use one of the following examples:

- To associate the service account with a task run:

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: build-push-task-run-2 (1)
spec:
  taskRunTemplate:
    serviceAccountName: build-bot (2)
  taskRef:
    name: build-push (3)
```

(1) Name of the task run. In this example, **build-push-task-run-2**.

(2) Name of the service account. In this example, **build-bot**.

(3) Name of the task. In this example, **build-push**.

- To associate the service account with a pipeline run:

```
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: demo-pipeline (1)
  namespace: default
spec:
  taskRunTemplate:
    serviceAccountName: build-bot (2)
  pipelineRef:
    name: demo-pipeline (3)
```

(1) Name of the pipeline run. In this example, **demo-pipeline**.

(2) Name of the service account. In this example, **build-bot**.

(3) Name of the pipeline. In this example, **demo-pipeline**.

6. Apply the changes.

```
$ oc apply --filename secret.yaml,serviceaccount.yaml,run.yaml
```

5.2.4. Configuring container registry authentication using a service account

For a pipeline to retrieve container images from a registry or push container images to a registry, you must configure the authentication for that registry.

To configure registry authentication for a pipeline, create an authentication secret with the Docker configuration file, associate this secret with a service account, and associate this service account with a **TaskRun** or **PipelineRun** resource.

Procedure

1. Create the container registry authentication secret from an existing **config.json** file, which contains the authentication information, by entering the following command:

```
$ oc create secret generic my-registry-credentials \ (1)
  --from-file=config.json=/home/user/credentials/config.json (2)
```

(1) The name of the secret, in this example, **my-registry-credentials**

2 The path name of the **config.json** file, in this example, **/home/user/credentials/config.json**

2. Create the YAML manifest for the service account in the **serviceaccount.yaml** file. In this manifest, associate the secret with the service account.

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: container-bot 1
secrets:
  - name: my-registry-credentials 2
```

1 Name of the service account. In this example, **container-bot**.

2 Name of the secret containing the SSH private key. In this example, **my-registry-credentials**.

3. Create a YAML manifest for a task run or pipeline run as the **run.yaml** file. In this file, associate the service account with a task run or a pipeline run. Use one of the following examples:

   - To associate the service account with a task run:

```
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: build-container-task-run-2 1
spec:
  taskRunTemplate:
    serviceAccountName: container-bot 2
  taskRef:
    name: build-container 3
```

1 Name of the task run. In this example, **build-container-task-run-2**.

2 Name of the service account. In this example, **container-bot**.

3 Name of the task. In this example, **build-container**.

   - To associate the service account with a pipeline run:

```
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: demo-pipeline 1
  namespace: default
spec:
  taskRunTemplate:
    serviceAccountName: container-bot 2
  pipelineRef:
    name: demo-pipeline 3
```

1 Name of the pipeline run. In this example, **demo-pipeline**.

2 Name of the service account. In this example, **container-bot**.

3 Name of the pipeline. In this example, **demo-pipeline**.

4. Apply the changes by entering the following command:

```
$ oc apply --filename serviceaccount.yaml,run.yaml
```

5.2.5. Additional considerations for authentication using service accounts

In certain cases, you must complete additional steps to use authentication secrets that you provide using service accounts.

5.2.5.1. SSH Git authentication in tasks

You can directly invoke Git commands in the steps of a task and use SSH authentication, but you must complete an additional step.

OpenShift Pipelines provides the SSH files in the **/tekton/home/.ssh** directory and sets the **$HOME** variable to **/tekton/home**. However, Git SSH authentication ignores the **$HOME** variable and uses the home directory specified in the **/etc/passwd** file for the user. Therefore, a step that uses Git command must symlink the **/tekton/home/.ssh** directory to the home directory of the associated user.

For example, if the task runs as the **root** user, the step must include the following command before Git commands:

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: example-git-task
spec:
  steps:
    - name: example-git-step
#     ...
      script:
        ln -s $HOME/.ssh /root/.ssh
#     ...
```

However, explicit symlinks are not necessary when you use a pipeline resource of the **git** type or the **git-clone** task available in the Tekton catalog.

As an example of using SSH authentication in **git** type tasks, refer to authenticating-git-commands.yaml.

5.2.5.2. Use of secrets as a non-root user

You might need to use secrets as a non-root user in certain scenarios, such as:

- The users and groups that the containers use to execute runs are randomized by the platform.
- The steps in a task define a non-root security context.
- A task specifies a global non-root security context, which applies to all steps in a task.

In such scenarios, consider the following aspects of executing task runs and pipeline runs as a non-root user:

- SSH authentication for Git requires the user to have a valid home directory configured in the **/etc/passwd** directory. Specifying a UID that has no valid home directory results in authentication failure.
- SSH authentication ignores the **$HOME** environment variable. So you must or symlink the appropriate secret files from the **$HOME** directory defined by OpenShift Pipelines (**/tekton/home**), to the non-root user's valid home directory.

In addition, to configure SSH authentication in a non-root security context, refer to the **git-clone-and-check** step in the example for authenticating git commands.

5.3. PROVIDING SECRETS USING WORKSPACES

You can use workspaces to provide secrets for authentication with Git repositories and container repositories.

You can configure a named workspace in a task, specifying a path where the workspace is mounted. When you run the task, provide the secret as the workspace with this name. When OpenShift Pipelines executes the task, the information in the secret is available to the task.

If you provide authentication secrets using workspaces, annotations for the secrets are not required.

5.3.1. Configuring SSH authentication for Git using workspaces

For a pipeline to retrieve resources from repositories configured with SSH keys, you must configure the SSH-based authentication for that pipeline.

To configure SSH-based authentication for a pipeline, create an authentication secret with the SSH private key, configure a named workspace for this secret in the task, and specify the secret when running the task.

Procedure

1. Create the Git SSH authentication secret from files in an existing **.ssh** directory by entering the following command:

```
$ oc create secret generic my-github-ssh-credentials \ 1
  --from-file=id_ed25519=/home/user/.ssh/id_ed25519 \ 2
  --from-file=known_hosts=/home/user/.ssh/known_hosts 3
```

1 The name of the secret, in this example, **my-github-ssh-credentials**

2 The name and full path name of the private key file, in this example, **/home/user/.ssh/id_ed25519**

3 The name and full path name of the known hosts file, in this example, **/home/user/.ssh/known_hosts**

2. In your task definition, configure a named workspace for the Git authentication, for example, **ssh-directory**:

Example definition of a workspace

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: git-clone
spec:
  workspaces:
    - name: ssh-directory
      description: |
        A .ssh directory with private key, known_hosts, config, etc.
```

3. In the steps of the task, access the directory using the path in the **$(workspaces.<workspace_name>.path)** environment variable, for example, **$(workspaces.ssh-directory.path)**

4. When running the task, specify the secret for the named workspace by including the **--workspace** argument in the **tkn task start** command:

```
$ tkn task start <task_name>
      --workspace name=<workspace_name>,secret=<secret_name> (1)
      # ...
```

(1) Replace **<workspace_name>** with the name of the workspace that you configured and **<secret_name>** with the name of the secret that you created.

Example task for cloning a Git repository using an SSH key for authentication

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: git-clone
spec:
  workspaces:
    - name: output
      description: The git repo will be cloned onto the volume backing this Workspace.
    - name: ssh-directory
      description: |
        A .ssh directory with private key, known_hosts, config, etc. Copied to
        the user's home before git commands are executed. Used to authenticate
        with the git remote when performing the clone. Binding a Secret to this
        Workspace is strongly recommended over other volume types
  params:
    - name: url
      description: Repository URL to clone from.
      type: string
    - name: revision
      description: Revision to checkout. (branch, tag, sha, ref, etc...)
      type: string
      default: ""
    - name: gitInitImage
      description: The image providing the git-init binary that this Task runs.
      type: string
      default: "gcr.io/tekton-releases/github.com/tektoncd/pipeline/cmd/git-init:v0.37.0"
```

```
results:
  - name: commit
    description: The precise commit SHA that was fetched by this Task.
  - name: url
    description: The precise URL that was fetched by this Task.
steps:
  - name: clone
    image: "$(params.gitInitImage)"
    script: |
      #!/usr/bin/env sh
      set -eu
      # This is necessary for recent version of git
      git config --global --add safe.directory '*'
      cp -R "$(workspaces.ssh-directory.path)" "${HOME}"/.ssh (1)
      chmod 700 "${HOME}"/.ssh
      chmod -R 400 "${HOME}"/.ssh/*
      CHECKOUT_DIR="$(workspaces.output.path)/"
      /ko-app/git-init \
        -url="$(params.url)" \
        -revision="$(params.revision)" \
        -path="${CHECKOUT_DIR}"
      cd "${CHECKOUT_DIR}"
      RESULT_SHA="$(git rev-parse HEAD)"
      EXIT_CODE="$?"
      if [ "${EXIT_CODE}" != 0 ] ; then
        exit "${EXIT_CODE}"
      fi
      printf "%s" "${RESULT_SHA}" > "$(results.commit.path)"
      printf "%s" "$(params.url)" > "$(results.url.path)"
```

(1) The script copies the content of the secret (in the form of a folder) to **${HOME}/.ssh**, which is the standard folder where **ssh** searches for credentials.

Example command for running the task

```
$ tkn task start git-clone
    --param url=git@github.com:example-github-user/buildkit-tekton
    --workspace name=output,emptyDir=""
    --workspace name=ssh-directory,secret=my-github-ssh-credentials
    --use-param-defaults --showlog
```

5.3.2. Configuring container registry authentication using workspaces

For a pipeline to retrieve container images from a registry, you must configure the authentication for that registry.

To configure authentication for a container registry, create an authentication secret with the Docker configuration file, configure a named workspace for this secret in the task, and specify the secret when running the task.

Procedure

1. Create the container registry authentication secret from an existing **config.json** file, which contains the authentication information, by entering the following command:

```
$ oc create secret generic my-registry-credentials \ (1)
  --from-file=config.json=/home/user/credentials/config.json (2)
```

(1) The name of the secret, in this example, **my-registry-credentials**

(2) The path name of the **config.json** file, in this example, **/home/user/credentials/config.json**

2. In your task definition, configure a named workspace for the Git authentication, for example, **ssh-directory**:

   Example definition of a workspace

   ```
   apiVersion: tekton.dev/v1beta1
   kind: Task
   metadata:
     name: skopeo-copy
   spec:
     workspaces:
       - name: dockerconfig
         description: Includes a docker `config.json`
   # ...
   ```

3. In the steps of the task, access the directory by using the path in the **$(workspaces.<workspace_name>.path)** environment variable, for example, **$(workspaces.dockerconfig.path)**.

4. To run the task, specify the secret for the named workspace by including the **--workspace** argument in the **tkn task start** command:

   ```
   $ tkn task start <task_name>
         --workspace name=<workspace_name>,secret=<secret_name> (1)
         # ...
   ```

   (1) Replace **<workspace_name>** with the name of the workspace that you configured and **<secret_name>** with the name of the secret that you created.

Example task for copying an image from a container repository using Skopeo

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: skopeo-copy
spec:
  workspaces:
    - name: dockerconfig (1)
      description: Includes a docker `config.json`
  steps:
    - name: clone
      image: quay.io/skopeo/stable:v1.8.0
      env:
      - name: DOCKER_CONFIG
```

```
        value: $(workspaces.dockerconfig.path) (2)
      script: |
        #!/usr/bin/env sh
        set -eu
        skopeo copy docker://docker.io/library/ubuntu:latest
docker://quay.io/example_repository/ubuntu-copy:latest
```

1. The name of the workspace that contains the **config.json** file.

2. The **DOCKER_CONFIG** environment variable points to the location of the **config.json** file in the **dockerconfig** workspace. Skopeo uses this environment variable to get the authentication information.

Example command for running the task

```
$ tkn task start skopeo-copy
    --workspace name=dockerconfig,secret=my-registry-credentials
    --use-param-defaults --showlog
```

5.3.3. Limiting a secret to particular steps using workspaces

When you provide authentication secrets using workspaces and define the workspace in a task, by default the workspace is available to all steps in the task.

To limit a secret to specific steps, define the workspace both in the task specification and in the step specification.

Procedure

- Add the **workspaces:** definition under both the task specification and the step specification, as in the following example:

  Example task definition where only one step can access the credentials workspace

```
apiVersion: tekton.dev/v1beta1
kind: Task
metadata:
  name: git-clone-build
spec:
  workspaces: (1)
    - name: ssh-directory
      description: |
        A .ssh directory with private key, known_hosts, config, etc.
# ...
  steps:
    - name: clone
      workspaces: (2)
        - name: ssh-directory
# ...
    - name: build (3)
# ...
```

1. The definition of the **ssh-directory** workspace in the task specification.

2. The definition of the **ssh-directory** workspace in the step specification. The authentication information is available to this step as the **$(workspaces.ssh-**

3. As this step does not include a definition of the **ssh-directory** workspace, the authentication information is not available to this step.

CHAPTER 6. BUILDING OF CONTAINER IMAGES USING BUILDAH AS A NON-ROOT USER

Running OpenShift Pipelines as the root user on a container can expose the container processes and the host to other potentially malicious resources. You can reduce this type of exposure by running the workload as a specific non-root user in the container. To run builds of container images using Buildah as a non-root user, you can perform the following steps:

- Define custom service account (SA) and security context constraint (SCC).
- Configure Buildah to use the **build** user with id **1000**.
- Start a task run with a custom config map, or integrate it with a pipeline run.

6.1. CONFIGURING CUSTOM SERVICE ACCOUNT AND SECURITY CONTEXT CONSTRAINT

The default **pipeline** SA allows using a user id outside of the namespace range. To reduce dependency on the default SA, you can define a custom SA and SCC with necessary cluster role and role bindings for the **build** user with user id **1000**.

IMPORTANT

At this time, enabling the **allowPrivilegeEscalation** setting is required for Buildah to run successfully in the container. With this setting, Buildah can leverage **SETUID** and **SETGID** capabilities when running as a non-root user.

Procedure

- Create a custom SA and SCC with necessary cluster role and role bindings.

Example: Custom SA and SCC for used id 1000

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipelines-sa-userid-1000 (1)
---
kind: SecurityContextConstraints
metadata:
  annotations:
  name: pipelines-scc-userid-1000 (2)
allowHostDirVolumePlugin: false
allowHostIPC: false
allowHostNetwork: false
allowHostPID: false
allowHostPorts: false
allowPrivilegeEscalation: true (3)
allowPrivilegedContainer: false
allowedCapabilities: null
apiVersion: security.openshift.io/v1
defaultAddCapabilities: null
fsGroup:
  type: MustRunAs
groups:
- system:cluster-admins
priority: 10
readOnlyRootFilesystem: false
requiredDropCapabilities:
- MKNOD
- KILL
runAsUser: (4)
  type: MustRunAs
  uid: 1000
seLinuxContext:
  type: MustRunAs
supplementalGroups:
  type: RunAsAny
users: []
volumes:
- configMap
- downwardAPI
- emptyDir
- persistentVolumeClaim
- projected
- secret
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pipelines-scc-userid-1000-clusterrole (5)
rules:
- apiGroups:
  - security.openshift.io
  resourceNames:
  - pipelines-scc-userid-1000
  resources:
  - securitycontextconstraints
  verbs:
  - use
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pipelines-scc-userid-1000-rolebinding (6)
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: pipelines-scc-userid-1000-clusterrole
subjects:
- kind: ServiceAccount
  name: pipelines-sa-userid-1000
```

1. Define a custom SA.

2. Define a custom SCC created based on restricted privileges, with modified **runAsUser** field.

3.

At this time, enabling the **allowPrivilegeEscalation** setting is required for Buildah to run successfully in the container. With this setting, Buildah can leverage **SETUID** and **SETGID**

4 Restrict any pod that gets attached with the custom SCC through the custom SA to run as user id **1000**.

5 Define a cluster role that uses the custom SCC.

6 Bind the cluster role that uses the custom SCC to the custom SA.

6.2. CONFIGURING BUILDAH TO USE BUILD USER

You can define a Buildah task to use the **build** user with user id **1000**.

Procedure

1. Create a copy of the **buildah** cluster task as an ordinary task.

```
$ oc get clustertask buildah -o yaml | yq '. |= (del .metadata |= with_entries(select(.key ==
"name" )))' | yq '.kind="Task"' | yq '.metadata.name="buildah-as-user"' | oc create -f -
```

2. Edit the copied **buildah** task.

```
$ oc edit task buildah-as-user
```

Example: Modified Buildah task with build user

```
apiVersion: tekton.dev/v1
kind: Task
metadata:
  name: buildah-as-user
spec:
  description: >-
    Buildah task builds source into a container image and
    then pushes it to a container registry.
    Buildah Task builds source into a container image using Project Atomic's
    Buildah build tool.It uses Buildah's support for building from Dockerfiles,
    using its buildah bud command.This command executes the directives in the
    Dockerfile to assemble a container image, then pushes that image to a
    container registry.
  params:
  - name: IMAGE
    description: Reference of the image buildah will produce.
  - name: BUILDER_IMAGE
    description: The location of the buildah builder image.
    default:
registry.redhat.io/rhel8/buildah@sha256:99cae35f40c7ec050fed3765b2b27e0b8bbea2aa2da7
c16408e2ca13c60ff8ee
  - name: STORAGE_DRIVER
    description: Set buildah storage driver
    default: vfs
  - name: DOCKERFILE
    description: Path to the Dockerfile to build.
  default: ./Dockerfile
- name: CONTEXT
  description: Path to the directory to use as context.
  default: .
- name: TLSVERIFY
  description: Verify the TLS on the registry endpoint (for push/pull to a non-TLS registry)
  default: "true"
- name: FORMAT
  description: The format of the built container, oci or docker
  default: "oci"
- name: BUILD_EXTRA_ARGS
  description: Extra parameters passed for the build command when building images.
  default: ""
- description: Extra parameters passed for the push command when pushing images.
  name: PUSH_EXTRA_ARGS
  type: string
  default: ""
- description: Skip pushing the built image
  name: SKIP_PUSH
  type: string
  default: "false"
results:
- description: Digest of the image just built.
  name: IMAGE_DIGEST
  type: string
workspaces:
- name: source
steps:
- name: build
  securityContext:
    runAsUser: 1000 1
  image: $(params.BUILDER_IMAGE)
  workingDir: $(workspaces.source.path)
  script: |
    echo "Running as USER ID `id`" 2
    buildah --storage-driver=$(params.STORAGE_DRIVER) bud \
      $(params.BUILD_EXTRA_ARGS) --format=$(params.FORMAT) \
      --tls-verify=$(params.TLSVERIFY) --no-cache \
      -f $(params.DOCKERFILE) -t $(params.IMAGE) $(params.CONTEXT)
    [[ "$(params.SKIP_PUSH)" == "true" ]] && echo "Push skipped" && exit 0
    buildah --storage-driver=$(params.STORAGE_DRIVER) push \
      $(params.PUSH_EXTRA_ARGS) --tls-verify=$(params.TLSVERIFY) \
      --digestfile $(workspaces.source.path)/image-digest $(params.IMAGE) \
      docker://$(params.IMAGE)
    cat $(workspaces.source.path)/image-digest | tee /tekton/results/IMAGE_DIGEST
  volumeMounts:
  - name: varlibcontainers
    mountPath: /home/build/.local/share/containers 3
volumes:
- name: varlibcontainers
  emptyDir: {}
```

1. Run the container explicitly as the user id **1000**, which corresponds to the **build** user in the Buildah image.

2 Display the user id to confirm that the process is running as user id **1000**.

3 You can change the path for the volume mount as necessary.

6.3. STARTING A TASK RUN WITH CUSTOM CONFIG MAP, OR A PIPELINE RUN

After defining the custom Buildah cluster task, you can create a **TaskRun** object that builds an image as a **build** user with user id **1000**. In addition, you can integrate the **TaskRun** object as part of a **PipelineRun** object.

Procedure

1. Create a **TaskRun** object with a custom **ConfigMap** and **Dockerfile** objects.

Example: A task run that runs Buildah as user id 1000

```
apiVersion: v1
data:
  Dockerfile: |
    ARG BASE_IMG=registry.access.redhat.com/ubi9/ubi
    FROM $BASE_IMG AS buildah-runner
    RUN dnf -y update && \
        dnf -y install git && \
        dnf clean all
    CMD git
kind: ConfigMap
metadata:
  name: dockerfile 1
---
apiVersion: tekton.dev/v1
kind: TaskRun
metadata:
  name: buildah-as-user-1000
spec:
  taskRunTemplate:
    serviceAccountName: pipelines-sa-userid-1000 2
  params:
  - name: IMAGE
    value: image-registry.openshift-image-registry.svc:5000/test/buildahuser
  taskRef:
    kind: Task
    name: buildah-as-user
  workspaces:
  - configMap:
      name: dockerfile 3
    name: source
```

1 Use a config map because the focus is on the task run, without any prior task that fetches some sources with a Dockerfile.

2 The name of the service account that you created.

3 Mount a config map as the source workspace for the **buildah-as-user** task.

2. (Optional) Create a pipeline and a corresponding pipeline run.

Example: A pipeline and corresponding pipeline run

```
apiVersion: tekton.dev/v1
kind: Pipeline
metadata:
  name: pipeline-buildah-as-user-1000
spec:
  params:
  - name: IMAGE
  - name: URL
  workspaces:
  - name: shared-workspace
  - name: sslcertdir
    optional: true
  tasks:
  - name: fetch-repository (1)
    taskRef:
      name: git-clone
      kind: ClusterTask
    workspaces:
    - name: output
      workspace: shared-workspace
    params:
    - name: url
      value: $(params.URL)
    - name: subdirectory
      value: ""
    - name: deleteExisting
      value: "true"
  - name: buildah
    taskRef:
      name: buildah-as-user (2)
    runAfter:
    - fetch-repository
    workspaces:
    - name: source
      workspace: shared-workspace
    - name: sslcertdir
      workspace: sslcertdir
    params:
    - name: IMAGE
      value: $(params.IMAGE)
---
apiVersion: tekton.dev/v1
kind: PipelineRun
metadata:
  name: pipelinerun-buildah-as-user-1000
spec:
  taskRunSpecs:
    - pipelineTaskName: buildah
      taskServiceAccountName: pipelines-sa-userid-1000 (3)
```

```
params:
- name: URL
  value: https://github.com/openshift/pipelines-vote-api
- name: IMAGE
  value: image-registry.openshift-image-registry.svc:5000/test/buildahuser
pipelineRef:
  name: pipeline-buildah-as-user-1000
workspaces:
- name: shared-workspace 4
  volumeClaimTemplate:
    spec:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 100Mi
```

1 Use the **git-clone** cluster task to fetch the source containing a Dockerfile and build it using the modified Buildah task.

2 Refer to the modified Buildah task.

3 Use the service account that you created for the Buildah task.

4 Share data between the **git-clone** task and the modified Buildah task using a persistent volume claim (PVC) created automatically by the controller.

3. Start the task run or the pipeline run.

6.4. LIMITATIONS OF UNPRIVILEGED BUILDS

The process for unprivileged builds works with most **Dockerfile** objects. However, there are some known limitations might cause a build to fail:

- Using the **--mount=type=cache** option might fail due to lack of necessay permissions issues. For more information, see this article.
- Using the **--mount=type=secret** option fails because mounting resources requires additionnal capabilities that are not provided by the custom SCC.

Additional resources

- Managing security context constraints (SCCs)