Installing & Configuring Kubernetes/OpenShift support in CyberArk DAP

Draft 0.1

April 23, 2020

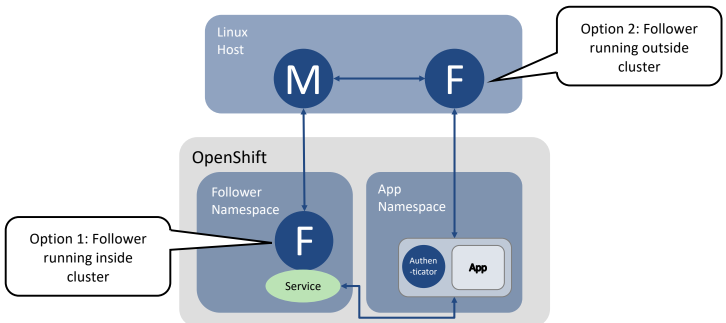
Table of Contents

Overview

Objectives

No secret zero	One of the fundamental challenges in securing the identity of an application is the "secret zero" problem. Put simply, how can one secure the initial credential (password, token, cert, etc.) required to authenticate the identity of the application and its access to other secrets? It has to be accessible to the application, but inaccessible to anything else, which is very hard to achieve. CyberArk DAP focuses on eliminating secret zero by defining identities in terms of attributes that can be verified with platforms or tools. DAP trusts the platform to validate the authenticity of an identity rather than trusting a credential that could easily be copied and used in unauthorized ways.
Simple developer experience	Easing the security burden for developers is also a top priority. <u>No need to know how to authenticate</u> Authentication in OpenShift and Kubernetes is performed by a small container that authenticates pods without writing any code. <u>Simple secrets retrieval w/ APIs</u> Applications can retrieve secrets directly from the DAP service using REST calls or the Go, Java, .Net and Ruby libraries. <u>Secrets injection options</u> Secrets injection eliminates the need for applications to retrieve secrets for themselves. • <u>Summon</u> CyberArk provides an open-source solution called Summon (https://cyberark.github.io/summon/) that runs in an application image, retrieves secrets for the application and calls the application with those secrets in environment variables. • <u>Secrets provider for K8s secrets</u> CyberArk DAP also supports dynamically retrieving secrets and providing them as Kubernetes secrets, accessible as files in the application's filesystem.
Security best-practices	CyberArk DAP supports a default-deny, zero trust model in which only authenticated identities can request secrets to which they have been granted access. DAP's granular RBAC model supports segregating duties across applications and personnel (e.g. cluster admins vs. developers).
Options for identities	The granularity of identity is also configurable: • Namespace/Project identities - all pods in a namespace/project share the same identity. • Service Accounts – a pod runs as specific Kubernetes service account.
Auto-Scalable	Auto-scaling of the DAP service is an important feature to support dynamic workloads which may rise and fall over time.
SPIFFE-compliant	SPIFFE is a project under the CNCF umbrella focused on providing identities for and authentication of workloads (rather than infrastructure). It is based on using x509 certificates as credentials and is implemented by multiple vendors. CyberArk DAP issues SPIFFE-compliant SPIFFE Verifiable Identity Documents (SVIDS, see: https://spiffe.io/spiffe/concepts/#spiffe-verifiable-identity-document-svid), which are x509 certs with a SPIFFE URI as one of the Subject Alternative Names.

Solution Architecture



DAP Master

The DAP Master does not run in the Kubernetes/OpenShift cluster. It is a stateful container that cannot be rescheduled. Running the DAP Master in-cluster is not supported.

DAP Followers

Kubernetes and OpenShift applications can authenticate to Followers running inside or outside of the cluster. CyberArk recommends running Followers in the cluster to take advantage of capabilities such as autoscaling, rolling upgrades, affinity rules, and scheduling.

Application Pods

Every application pod will contain at least two containers: the application, and either an authenticator (in diagram) or secrets provider container (not in diagram). The authenticator container handles only authentication. The secrets provider container handles authentication as well as provisioning of secrets to Kubernetes secrets.

Network configuration

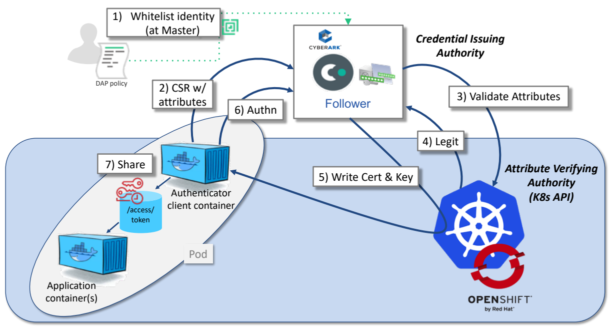
CyberArk DAP Masters and Followers communicate over ports that are mapped to ports on their hosts. By default, containers use ports 443 (https), 5432 (PostgreSQL replication) and 1999 (syslog forwarding) internally. These ports can be mapped to arbitrary host ports on the DAP Master; those host ports must be open for listening for IPV4 packets.

Authentication workflow overview

The authn-k8s workflow serves two purposes:

- to authenticate Followers to the Master, to fetch a seedfile for Follower self-initialization, and
- to authenticate applications to Followers, for secrets retrieval.

The authentication workflow for applications is depicted below.



At application configuration time:

1) A DAP administrator whitelists an identity annotated with namespace/project and optional service account by loading a DAP policy to the DAP Master. Typically, this only needs to be done once when onboarding a new application. The policy defines an application identity and grants it permission to call the DAP authenticator webservice endpoint for the cluster, defined by the Cluster Authenticator ID.

At application runtime:

2) The authenticator container formats and submits a Certificate Signing Request (CSR) to the Follower authn-k8s webservice for the cluster, which contains the Pod name, current IP address and Namespace/Project name as attributes to verify with the cluster.
3) If the application is not authorized to authenticate (per DAP policy) to the Cluster Athenticator ID endpoint, or if the certificate accompanying the request does not contain the Follower server SAN, the request is rejected outright.
   If authorized, the Follower parses the CSR, extracts the attributes and validates them by calling the Kubernetes API, using the DAP service account token and certificate.
4) If attributes are successfully verified,
5) The Follower issues an x509 certificate (SPIFFE SVID) and private key, and writes those files directly into the authenticator container's filesystem. This is a kind of second factor for authentication, since if the requestor is not where it said it was, it will not receive the authentication credentials.
6) The authenticator uses the certificate and private key as credentials to setup an mTLS connection to the Follower and receives a DAP access token.
7) The authenticator writes the DAP token to a shared memory volume, where it can be used by the application container(s) in the same pod.

Follower seed-fetching

Followers are configured using a tarfile, called a seedfile, containing certificates, keys and configuration settings from the Master. During Follower initialization, the values in a seedfile are used to setup a secure mTLS connection to the Master, and for encrypting data on the Follower.

Since seedfiles contain sensitive values, and since Follower initialization needs to be automatic to support auto-scaling and rescheduling in the cluster, the seedfile creates a secret-zero problem. CyberArk DAP solves this by using the same application authentication strategy described above, whereby an Init container in Follower pods authenticates to the Master and retrieves the seedfile over a secure connection. The seedfile and a startup script are placed in a shared memory volume. The entrypoint for the Follower invokes the startup script, which unpacks the seedfile and initializes the Follower.

Note that Followers running outside the cluster cannot use the seed-fetching facility (because they have no Init container) and must be explicitly initialized. <reference to section of this doc for how to standup Follower and enable for k8s authn>

DAP Identity Architecture for K8s

A Cluster Authenticator ID is a value that represents the cluster in which applications are deployed. It uniquely defines an authentication webservice endpoint for a particular cluster and prevents identity spoofing. Even if two clusters have identical structure (e.g. same names for namespaces and service accounts), identities are unique to each cluster. This ensures, for example, that identities in a test cluster cannot accidently access production secrets.

Conceptually, there are two levels of identity granularity for authn-k8s.

Level	Conceptual Representation	Usage Notes
Namespace	<cluster-authn-id>/<namespace>	• Applications in a namespace share the same identity. • Less restrictive, easier to manage. • Does not support strict segregation of duty and least-privilege principles.
Service Account	< cluster-authn-id>/<namespace>/<service-account>	• Deployment manifests must specify a service account to run with. • Service accounts can be shared between or unique to deployments in the same namespace. • Supports strict segregation of duty and least-privilege principles. • More restrictive, harder to manage.

Setup & Resource Requirements

Infrastructure

Notes for cluster administrators – PLEASE READ THOROUGHLY

Supported versions

OCP 3.9 and Kubernetes X.XX and above are supported. The term "namespace" is used throughout this document to refer to both K8s namespaces and OCP projects. The kubectl CLI is used for all configuration except where OpenShift-specific configuration requires the oc CLI. Those cases are called out as being OpenShift only.

Networking

Cluster nodes on which DAP Followers run require IPV4 access to the DAP Master host on ports 443, 5432 and 1999. IPV4 packet forwarding must be enabled on all cluster nodes that run DAP Followers.

This documentation assumes DAP Followers can use DNS to resolve the DAP Master hostname. A host alias in Follower deployment manifests can substitute for DNS resolution, as long as the hostname specified in the host alias matches one of the DNS entries in the master certificate.

RBAC and Namespace administration

All DAP cluster runtime artifacts (Follower pods, service accounts, etc.) are created and operate in a namespace named "cyberark". Applications run in their own namespaces and initiate requests to the DAP service in the cyberark namespace.

Cluster admin privileges are only required for namespace initialization to apply RBAC settings in the cyberark and application namespaces. The RBAC model in this documentation assumes there are cluster users ("namespace admins") to whom admin responsibilities are delegated. All registry access and deployments are assumed to be done by namespace admins. Those users are assumed to already have login accounts in the cluster.

If you are using OpenShift SDN multitenant isolation or other restrictive network policies, you will need to enable traffic between the application and cyberark namespaces. That configuration should be done during initial application namespace configuration and is not covered by this documentation.

DAP Followers use a cluster role to write authentication credentials into application pod filesystems. Follower access to application pods is restricted to just those namespaces for which a role binding is applied to the cluster role.

Application deployment includes copying a DAP config map from the cyberark namespace into application namespaces. Application namespace admins are granted a role binding to a cluster role that enables read access to the DAP config map in the cyberark namespace. The DAP config map contains non-sensitive information such as URLs, server certificates, etc. that are required to access the DAP service.

Configured DAP Master

This documentation assumes a DAP Master is already configured, running, and healthy. Configuration of the Master for Kubernetes/OpenShift authentication requires a shell on the DAP Master host. Many commands require "sudo docker exec" to execute them in the DAP Master container.

Images

All CyberArk DAP nodes (Master/Follower/Standby) are instances of the same Conjur Appliance image. The image is ONLY available for download from the CyberArk Support Vault. You will need to contact a CyberArk representative and be granted access to the CyberArk Conjur safe. All other images can be either downloaded from the CyberArk Conjur safe in the Support Vault, or pulled from DockerHub.com.

Image name	Support Vault name	DockerHub.com name
Conjur appliance	conjur-appliance-vXX.X.tar.gz NOTE: DAP version 11.3 or higher is required.	n/a
Follower seed-fetcher	seed-fetcher.tar.gz	cyberark/dap-seedfetcher
Authenticator client	conjur-kubernetes-authenticator_X.X.X.tar.gz	cyberark/conjur-authn-k8s-client
Secrets provider for K8s	secrets-provider.tar.gz	cyberark/secrets-provider-for-k8s

Distributed Configuration Management

Often the DAP Master is running on a host that is remote from the cluster, and administered by a different team than the one managing the OCP/K8s cluster. This introduces the opportunity for inconsistencies between configurations that can lead to incorrect function.

This documentation partitions tasks to minimize the amount of coordination required between teams. All information that must be consistent between Master and cluster configurations is documented in the Table of Configuration Variables below.

Table of Configuration Variables

Variable Name	Description
AUTHN_USERNAME	The name of the DAP admin user. Default is "admin". Scripts will prompt for this value when needed.
AUTHN_PASSWORD	The password for the DAP admin user, set at DAP Master configuration time. This should be known, but not recorded persistently. Scripts will prompt for this value when needed.
CONJUR_ACCOUNT	The default account value set during DAP Master configuration.
CONJUR_MASTER_HOSTNAME	The fully-qualified DNS name of the host on which the DAP master is running. If the DAP Master is listening on a port other than the default 443, include the port as part of the hostname, e.g. `hostname:port-number`
CONJUR_MASTER_URL	The URL for the making calls to the DAP Master, e.g. `https://$CONJUR_MASTER_HOSTNAME`
CLUSTER_AUTHN_ID	A unique name for an authenticator endpoint that briefly describes the function or location of the cluster it runs in, e.g. Dev, Test1, QA, Prod, US-East-1, etc.
CONUR_MASTER_CONTAINER_NAME	The Docker container name of the DAP Master, set during Master configuration.
CONJUR_FOLLOWER_SERVICE_NAME	The Kubernetes service endpoint for Followers in the cluster, e.g. `conjur-follower.cyberark.svc.cluster.local`

CONJUR_AUTHENTICATORS	A list of authenticators enabled for a DAP node, e.g. `authn,authn-k8s/test,authn-aws/us-east-1`
CONJUR_AUTHENTICATOR_URL	The URL specific to a cluster authentication endpoint, e.g. `https://conjur-follower.cyberark.svc.cluster.local api/authn-k8s/test`
CYBERARK_NAMESPACE_ADMIN	If using cluster RBAC, the K8s cluster username to whom admin privileges are given for the CyberArk namespace.
CONJUR_SEED_FETCHER_IMAGE	The registry entry for the DAP seed-fetcher image, e.g. `<registry-url>/<namespace>/dap-seedfetcher:latest`
CONJUR_APPLIANCE_IMAGE	The registry entry for the DAP appliance image, e.g. `<registry-url>/<namespace>/conjur-appliance:11.3.0`
APP_NAMESPACE_NAME	The name of the cluster namespace in which applications are deployed.
APP_NAMESPACE_ADMIN	If using cluster RBAC, the K8s cluster username to whom admin privileges are given for the application namespace.
APP_IMAGE	The registry entry for an application image to be deployed, e.g. `<registry-url>/<namespace>/test-app:latest`
AUTHENTICATOR_IMAGE	The registry entry for the CyberArk authenticator client image, e.g. `<registry-url>/<namespace>/conjur-authn-k8s-client:latest`
SECRETS_PROVIDER_IMAGE	The registry entry for the CyberArk Secrets Provider for K8s image, e.g. `<registry-url>/<namespace>/secrets-provider-for-k8s:latest`

To ensure consistency across environments, a best practice is to record these values in a shell script resource file (none of the values are sensitive) so they can be referenced as environment variables (using `$variable-name` syntax) or used by the sed utility to substitute those values in manifest and policy templates (using `{{ variable-name }}` syntax).

Example configuration variable resource file

```
export CONJUR_ACCOUNT=dev
export CONJUR_MASTER_HOSTNAME=linuxhost013.foo.dev
export CLUSTER_AUTHN_ID=doctest
export CONJUR_MASTER_CONTAINER_NAME=conjur1
export CONJUR_AUTHENTICATORS=authn,authn-k8s/doctest

export CONJUR_AUTHENTICATOR_URL=https://conjur-follower.cyberark.svc.cluster.local/api/authn-k8s/doctest
export CYBERARK_NAMESPACE_ADMIN=dapadmin
export CONJUR_SEED_FETCHER_IMAGE=192.168.1.10:5000/cyberark/dap-seedfetcher:latest
export CONJUR_APPLIANCE_IMAGE=192.168.1.10:5000/cyberark/conjur-appliance:11.3.0
export APP_NAMESPACE_NAME=testapps
export APP_NAMESPACE_ADMIN=appadmin
export APP_IMAGE=192.168.1.10:5000/testapps/test-app:latest
export AUTHENTICATOR_IMAGE=192.168.1.10:5000/testapps/conjur-authn-k8s-client:latest
export SECRETS_PROVIDER_IMAGE=192.168.1.10:5000/testapps/secrets-provider-for-k8s:latest
```

Scripting Tools

To facilitate loading of DAP policies and setting/getting of DAP variable values, you will need to cut & paste each of the bash shell scripts documented in the two sections below (load_policy.sh script and get_set.sh script) into files named "load_policy.sh" and "get_set.sh" respectively, then run "chmod +x" on both to make them executable. These scripts need to be available on both the DAP Master host and on the host from which `kubectl` CLI commands are run for Follower configuration. Be sure to edit the scripts to set values for CONJUR_APPLIANCE_URL and CONJUR_ACCOUNT, per the DAP Master host configuration. See above for descriptions of these configuration variables.

load_policy.sh script

```
#!/bin/bash

# Authenticates as admin user and loads policy file.
# If you set the environment variables AUTHN_USERNAME and AUTHN_PASSWORD
# to appropriate values, you can avoid having to enter the admin username
# and password every time this script runs. UNSET THEM WHEN FINISHED.

CONJUR_APPLIANCE_URL=
CONJUR_ACCOUNT=
if [ -z "${CONJUR_APPLIANCE_URL}" ]; then
  echo "You must set CONJUR_APPLIANCE_URL and CONJUR_ACCOUNT in script."
  exit -1
fi

################  MAIN   ################
# $1 - name of policy file to load
main() {

  if [[ $# < 2 ]] ; then
    printf "\nUsage: %s <policy-branch-id> <policy-filename> [ delete | replace ]\n" $0
    printf "\nExamples:\n"
    printf "\t$> %s root /tmp/policy.yml\n" $0
    printf "\t$> %s dev/my-app /tmp/policy.yml\n" $0
    printf "\nDefault is append mode, unless 'delete' or 'replace' is specified\n"
    exit -1
  fi
  local policy_branch=$1
  local policy_file=$2

  local LOAD_MODE="POST"
  if [[ $# == 3 ]]; then
    case $3 in
      delete)   LOAD_MODE="PATCH"
                  ;;
      replace)  LOAD_MODE="PUT"
                  ;;
      *) printf "\nSpecify 'delete' or 'replace' as load mode options.\n\n"
                  exit -1
    esac
  fi

  authn_user   # authenticate user
  if [[ "$AUTHN_TOKEN" == "" ]]; then
    echo "Authentication failed..."
    exit -1
  fi

  curl -sk \
     -H "Content-Type: application/json" \
     -H "Authorization: Token token=\"$AUTHN_TOKEN\"" \
     -X $LOAD_MODE -d "$(< $policy_file)" \
     $CONJUR_APPLIANCE_URL/policies/$CONJUR_ACCOUNT/policy/$policy_branch
  echo
}

##################
# AUTHN USER - sets AUTHN_TOKEN globally
# - no arguments
authn_user() {
  if [ -z ${AUTHN_USERNAME+x} ]; then
    echo
    echo -n Enter admin user name:
    read admin_uname
    echo -n Enter the admin password \(it will not be echoed\):
    read -s admin_pwd
    echo
    export AUTHN_USERNAME=$admin_uname
    export AUTHN_PASSWORD=$admin_pwd
  fi
  # Login user, authenticate and get API key for session
  local api_key=$(curl \
                    -sk \
                    --user $AUTHN_USERNAME:$AUTHN_PASSWORD \
                    $CONJUR_APPLIANCE_URL/authn/$CONJUR_ACCOUNT/login)

  local response=$(curl -sk \
                     --data $api_key \
                     $CONJUR_APPLIANCE_URL/authn/$CONJUR_ACCOUNT/$AUTHN_USERNAME/authenticate)
  AUTHN_TOKEN=$(echo -n $response| base64 | tr -d '\r\n')
}

main "$@"
```

get_set.sh script

```
#!/bin/bash

# Authenticates as a user and gets or sets value of a specified variable.
# If you set the environment variables AUTHN_USERNAME and AUTHN_PASSWORD
# to appropriate values, you can avoid having to enter the admin username
# and password every time this script runs. UNSET THEM WHEN FINISHED.

CONJUR_APPLIANCE_URL=
CONJUR_ACCOUNT=
if [ -z "${CONJUR_APPLIANCE_URL}" ]; then
  echo "You must set CONJUR_APPLIANCE_URL and CONJUR_ACCOUNT in script."
  exit -1
fi

###############  MAIN   ###############
# $1 - command (get or set)
# $2 - name of variable
# $3 - value to set
main() {
  case $1 in
    get)  local command=get
         local variable_name=$2
         ;;
    set)  local command=set
         local variable_name=$2
         local variable_value="$3"
         ;;
    *)  printf "\nUsage: %s [ get | set ] <variable-name> [ <variable-value> ]\n" $0
         exit -1
  esac

  authn_user   # authenticate user
  if [[ "$AUTHN_TOKEN" == "" ]]; then
    echo "Authentication failed..."
    exit -1
  fi

  variable_name=$(urlify "$variable_name")

  case $command in
    get)
         curl -sk -H "Content-Type: application/json" \
         -H "Authorization: Token token=\"$AUTHN_TOKEN\"" \
         $CONJUR_APPLIANCE_URL/secrets/$CONJUR_ACCOUNT/variable/$variable_name
         ;;
    set)
         curl -sk -H "Content-Type: application/json" \
         -H "Authorization: Token token=\"$AUTHN_TOKEN\"" \
         --data "$variable_value" \
         $CONJUR_APPLIANCE_URL/secrets/$CONJUR_ACCOUNT/variable/$variable_name
         ;;
  esac
}

##################
# AUTHN USER - sets AUTHN_TOKEN globally
# - no arguments
authn_user() {
  if [ -z ${AUTHN_USERNAME+x} ]; then
    >&2 echo
    >&2 echo -n Enter admin user name:
    read admin_uname
    >&2 echo -n Enter the admin password \(it will not be echoed\):
    read -s admin_pwd
    export AUTHN_USERNAME=$admin_uname
    export AUTHN_PASSWORD=$admin_pwd
  fi

  # Login user, authenticate and set authn token
  local api_key=$(curl -sk \
                    --user $AUTHN_USERNAME:$AUTHN_PASSWORD \
                    $CONJUR_APPLIANCE_URL/authn/$CONJUR_ACCOUNT/login)
  local response=$(curl -sk --data $api_key \
                     $CONJUR_APPLIANCE_URL/authn/$CONJUR_ACCOUNT/$AUTHN_USERNAME/authenticate)
  AUTHN_TOKEN=$(echo -n $response| base64 | tr -d '\r\n')
}

################
# URLIFY - url encodes input string
# in: $1 - string to encode
# out: encoded string on stdout
urlify() {
        local str=$1; shift
        str=$(echo $str | sed 's= =%20=g')
        str=$(echo $str | sed 's=/=%2F=g')
        str=$(echo $str | sed 's=:=%3A=g')
        str=$(echo $str | sed 's=+=%2B=g')
        str=$(echo $str | sed 's=&=%26=g')
        str=$(echo $str | sed 's=@=%40=g')
        echo $str
}

main "$@"
```

DAP Master Configuration for K8s authentication

BEFORE YOU START CHECKLIST

Ensure you have these resources and permissions:

- Configured DAP Master
- ssh & sudo rights on DAP Master host
- configuration variable resource file (see above)
- executable get_set.sh and load_policy.sh scripts (see above)

DAP Master Configuration - Workflow overview

- Step M1: Load Authenticator Policies
- Step M2: Initialize CA & enable (whitelist) authenticator endpoint
- Step M3: Verify CA values & authenticators permissions
- Step M4: Save Master & Follower certs

Authenticator setup task detail

Step M1: Load Authenticator Policies

Two policies must be loaded to initialize the DAP Master for Follower and application authentication.

- `conjur/seed-generation` policy whitelists the identity of the service account identity that Followers run under. It authorizes the seed-fetcher container, running as an init container in a Follower pod to invoke the seed-generation webservice in the DAP Master and retrieve the seed file needed to initialize the Follower.

```
---
# ================================================
# == Register the seed generation service in the Master
# ================================================
- !policy
  id: conjur/seed-generation
  body:
  # This webservice represents the Seed service API
  - !webservice

  # Hosts that can generate seeds become members of the
  # `consumers` group.
  - !group consumers

  # Authorize `consumers` to request seeds
  - !permit
    role: !group consumers
    privilege: [ "execute" ]
    resource: !webservice
```

Figure 1: Master seed Generation policy

Cut & paste the above into a text file named master-seed-generation-policy.yaml, taking care to capture all whitespace. Then load the policy with the load_policy.sh script.

```
./load_policy.sh root master-seed-generation-policy.yaml
```

- `conjur/authn-k8s/`**`{{ CLUSTER_AUTHN_ID }}`** policy creates the necessary variables, webservice, group, host and permissions that govern the authentication process.

```
---
# ================================================
# == Register the authentication service for a cluster
```

```
# ================================================
- !policy
  id: conjur/authn-k8s/{{ CLUSTER_AUTHN_ID }}
  annotations:
    description: authn-k8s defs for the DAP cluster
  body:

  # vars for ocp/k8s api url & access creds
  - !variable kubernetes/service-account-token
  - !variable kubernetes/ca-cert
  - !variable kubernetes/api-url

  # vars for CA for this authenticator ID
  - !variable ca/cert
  - !variable ca/key

  - !webservice
    annotations:
      description: authn service for cluster {{ CLUSTER_AUTHN_ID }}

  # Hosts that can authenticate become members of the
  # `consumers` group.
  - !group consumers

  # Grant consumers group role authentication privileges
  - !permit
    role: !group consumers
    privilege: [ read, authenticate ]
    resource: !webservice

# ================================================
# == Grant entitlements for Follower initialization
# ================================================

# Define Follower host identity for authentication service in CyberArk namespace
- !host
  id: {{ CLUSTER_AUTHN_ID }}/dap-authn-service
  annotations:
    authn-k8s/namespace: cyberark
    authn-k8s/service_account/name: dap-authn-service
    authn-k8s/authentication-container-name: authenticator

# Grant roles that gives Follower host identity permission to:
#  - authenticate to the cluster authn-k8s endpoint
#  - execute the seed-generation webservice
- !grant
  roles:
  - !group conjur/authn-k8s/{{ CLUSTER_AUTHN_ID }}/consumers
  - !group conjur/seed-generation/consumers
  members:
  - !host {{ CLUSTER_AUTHN_ID }}/dap-authn-service
```

Figure 2: Master authenticator policy

Cut & paste the above into a text file named master-authenticator-policy.yaml, taking care to capture all whitespace. Substitute the correct value for:

- CLUSTER_AUTHN_ID

Then load the policy with the load_policy.sh script.

```
./load_policy.sh root master-authenticator-policy.yaml
```

Example, where {{ CLUSTER_AUTHN_ID }} is replaced with ‘doctest’ in policy text:

```
$ ./load_policy.sh root master-authenticator-policy.yaml

Enter admin user name:admin
Enter the admin password (it will not be echoed):
{"created_roles":{"dev:host:doctest/dap-authn-service":{"id":"dev:host:doctest/dap-authn-
service","api_key":"13devw31kjk1ek116ywfz97g1az16n95sc3xhk7hr365d9d52a043ag"}},"version":2}
```

Step M2: Initialize CA & enable (whitelist) authenticator endpoint

Configuration variables used in this step:

- CLUSTER_AUTHN_ID
- CONJUR_MASTER_CONTAINER_NAME
- CONJUR_AUTHENTICATORS

Initialize the authenticator CA values by executing a Ruby function within the DAP Master container. This will create the CA cert & key and store those values in the ca/cert & ca/key variables defined by the authenticator policy.

```
sudo docker exec $CONJUR_MASTER_CONTAINER_NAME \
      chpst -u conjur conjur-plugin-service possum \
        rake authn_k8s:ca_init["conjur/authn-k8s/$CLUSTER_AUTHN_ID"]
```

Get a list of currently enabled authenticators:

```
sudo docker exec $CONJUR_MASTER_CONTAINER_NAME \
     evoke variable list CONJUR_AUTHENTICATORS
```

Any values returned by the above command should be preserved, e.g. if it returns CONJUR_AUTHENTICATORS=authn-oidc, include authn-oidc with the authn-k8s endpoint in the new list of authenticators. In this case, you would set the CONJUR_AUTHENTICATORS configuration variable to: `authn-oidc,authn-k8s/$CLUSTER_AUTHN_ID`

Set CONJUR_AUTHENTICATORS to new list, which will then restart the conjur service in the master and echo all the configuration values.

```
sudo docker exec $CONJUR_MASTER_CONTAINER_NAME \
     evoke variable set CONJUR_AUTHENTICATORS $CONJUR_AUTHENTICATORS
```

Step M3: Verify CA values & authenticators permissions

Configuration variables used in this step:

- CLUSTER_AUTHN_ID
- CONJUR_MASTER_HOSTNAME

```
./get_set.sh get conjur/authn-k8s/$CLUSTER_AUTHN_ID/ca/key

./get_set.sh get conjur/authn-k8s/$CLUSTER_AUTHN_ID/ca/cert

curl -k https://$CONJUR_MASTER_HOSTNAME/info
```

Verify the ca/key and ca/cert values were set and that the authn-k8s/$CLUSTER_AUTHN_ID appears in the list of authenticators as “configured” and “enabled”.

Step M4: Save Master & Follower certs

Configuration variables used in this step:

- CONJUR_MASTER_CONTAINER_NAME

```
sudo docker exec $CONJUR_MASTER_CONTAINER_NAME \
     cat /opt/conjur/etc/ssl/conjur.pem \
     | awk '{ print "    " $0 }' \
     > dap-master.indented-pem

sudo docker exec $CONJUR_MASTER_CONTAINER_NAME \
     bash -c "evoke ca issue $CONJUR_FOLLOWER_SERVICE_NAME"

sudo docker exec $CONJUR_MASTER_CONTAINER_NAME \
     cat /opt/conjur/etc/ssl/conjur-follower.pem \
     | awk '{ print "    " $0 }' \
     > dap-follower.indented-pem
```

The contents of the *.indented-pem files are indented four spaces in order to be pasted into config map manifests. They are used by Followers and applications for secure connections to the DAP service. You will need to copy, email or otherwise make their contents accessible for Follower configuration in the next section.

This concludes the steps that must be run on the DAP Master host. All subsequent steps are run on a host with access to kubectl cluster configuration commands.

DAP Follower Configuration for K8s Authentication

BEFORE YOU START CHECKLIST

From DAP Master configuration steps:

- Executable load_policy.sh and get_set.sh scripts
- Configuration variable resource file
- *.indented-pem files

On Follower configuration host:

- Install jq and base64 utilities
- Ensure kubectl access
- Verify Cluster admin login privileges for cluster admin workflow
- Verify/create CyberArk namespace admin user in cluster

Cluster Admin workflow overview

- Step F1a: Apply CyberArk admin RBAC manifest (skip if not using user RBAC)
- Step F1b: Apply CyberArk RBAC manifest
- Step F1c: Apply security context constraint (OpenShift only)

Cluster Admin task detail

If using cluster RBAC, login as a cluster administrator.

Step F1a: Apply CyberArk admin RBAC manifest **(skip if not using user RBAC)**

```
---
# Cluster role to enable other projects to access and copy the DAP config map
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dap-cm-access-role
rules:
- apiGroups: [""]
  resources: ["configmaps"]
  resourceNames: ["dap-config"]
  verbs: ["get", "list"]

---
# Grant namespace admin role to user {{ CYBERARK_NAMESPACE_ADMIN }}
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: project-admin-access-binding
  namespace: cyberark
subjects:
- kind: User
  name: "{{ CYBERARK_NAMESPACE_ADMIN }}"
roleRef:
  kind: ClusterRole
  name: admin
  apiGroup: rbac.authorization.k8s.io
```

Figure 3: CyberArk admin manifest template

Cut & paste the above into a text file named cyberark-admin-manifest.yaml, taking care to capture all whitespace. Substitute the correct value for:

- CYBERARK_NAMESPACE_ADMIN

Then apply the manifest with kubectl.

```
kubectl apply -f cyberark-admin-manifest.yaml -n cyberark
```

Step F1b: Apply CyberArk RBAC manifest

```
---
# Create CyberArk namespace for Followers
apiVersion: v1
kind: Namespace
metadata:
  name: cyberark
  labels:
    name: cyberark

---
# Create service account for authentication service
apiVersion: v1
kind: ServiceAccount
metadata:
  name: dap-authn-service
  namespace: cyberark

---
# Create cluster role for authentication service access to pods
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dap-authn-role
rules:
- apiGroups: [""]
  resources: ["pods", "serviceaccounts"]
  verbs: ["get", "list"]
- apiGroups: ["extensions"]
  resources: [ "deployments", "replicasets"]
  verbs: ["get", "list"]
- apiGroups: ["apps"]
  resources: [ "deployments", "statefulsets", "replicasets"]
  verbs: ["get", "list"]
- apiGroups: [""]
  resources: ["pods/exec"]
  verbs: ["create", "get"]

---
# Grant the authentication service account access to pods in CyberArk namespace
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dap-authn-service
  namespace: cyberark
subjects:
- kind: ServiceAccount
  name: dap-authn-service
  namespace: cyberark
roleRef:
  kind: ClusterRole
  name: dap-authn-role
  apiGroup: rbac.authorization.k8s.io
```

Figure 4: CyberArk RBAC manifest template

Cut & paste the above into a text file named cyberark-rbac-manifest.yaml, taking care to capture all whitespace. Then apply the policy with kubectl.

```
kubectl apply -f cyberark-rbac-manifest.yaml -n cyberark
```

Step F1c: Apply security context constraint (OpenShift only)

```
oc adm policy add-scc-to-user anyuid -z dap-authn-service -n cyberark
```

CyberArk namespace Admin workflow overview

CyberArk namespace Admin task detail

If using cluster RBAC, login to the cluster as the cyberark namespace admin.

Step F2a: Initialize & verify K8s API access secrets in DAP

Get the DAP service account token K8s secret name:

```
SA_TOKEN_NAME="$(kubectl get secrets -n cyberark \
    | grep "dap-authn-service.*service-account-token" \
    | head -n1 \
    | awk '{print $1}')" && echo $SA_TOKEN_NAME
```

Get and store the DAP service account token as a DAP secret:

```
# using SA_TOKEN_NAME from above step…
./get_set.sh set \
     conjur/authn-k8s/$CLUSTER_AUTHN_ID/kubernetes/service-account-token \
     $(kubectl get secret -n cyberark $SA_TOKEN_NAME -o json\
      | jq -r .data.token \
      | base64 -d)
```

Get and store the K8s cluster CA certificate as a DAP secret:

```
# using SA_TOKEN_NAME from above step…
./get_set.sh set \
    conjur/authn-k8s/$CLUSTER_AUTHN_ID/kubernetes/ca-cert \
    "$(kubectl get secret -n cyberark $SA_TOKEN_NAME -o json \
      | jq -r '.data["ca.crt"]' \
      | base64 -d)"
```

Get and store the K8s cluster API server URL as a DAP secret:

```
./get_set.sh set \
    conjur/authn-k8s/$CLUSTER_AUTHN_ID/kubernetes/api-url \
    "$(kubectl config view --minify -o yaml | grep server | awk '{print $2}')"
```

Verify K8s cluster API secret values:

```
echo "$(./get_set.sh get conjur/authn-k8s/$CLUSTER_AUTHN_ID/kubernetes/ca-cert)" > k8s.crt

TOKEN=$(./get_set.sh get conjur/authn-k8s/$CLUSTER_AUTHN_ID/kubernetes/service-account-token)

API=$(./get_set.sh get conjur/authn-k8s/$CLUSTER_AUTHN_ID/kubernetes/api-url)

curl -s --cacert k8s.crt --header "Authorization: Bearer ${TOKEN}" $API/healthz && echo
```

The first three commands above simply store values with no response. The response to the curl command in the last step should be “ok”. If this is not the case, verify values returned and stored in all previous steps. Do not proceed until verification is successful and “ok” is return by the curl call to the K8s cluster health check.

Once verified, remove the temp cert and environment variables.

```
rm k8s.crt && unset API TOKEN SA_TOKEN_NAME
```

Step F2b: Tag & push images to registry

- Load the conjur-appliance tarfile and tag as $CONJUR_APPLIANCE_IMAGE
- Load the seed-fetcher image (or pull from Dockerhub) and tag as $CONJUR_SEED_FETCHER_IMAGE
- Push both to your registry.

Step F2c: Create DAP config map

```
# Holds DAP config info for apps in all namespaces
# Access is gained via rolebinding to a clusterrole
apiVersion: v1
kind: ConfigMap
metadata:
  name: dap-config
data:
  CONJUR_ACCOUNT: {{ CONJUR_ACCOUNT }}
  CONJUR_MASTER_HOST_NAME: {{ CONJUR_MASTER_HOSTNAME }}
  CONJUR_MASTER_URL: https://{{ CONJUR_MASTER_HOSTNAME }}
  CLUSTER_AUTHN_ID: {{ CLUSTER_AUTHN_ID }}
  CONJUR_VERSION: "5"
  CONJUR_APPLIANCE_URL: https://conjur-follower.cyberark.svc.cluster.local
  CONJUR_AUTHN_URL: https://conjur-follower.cyberark.svc.cluster.local/api/authn-k8s/{{ CLUSTER_AUTHN_ID }}
  CONJUR_AUTHN_TOKEN_FILE: /run/conjur/access-token
  CONJUR_MASTER_CERTIFICATE: |
  <contents of dap-master.indented-pem>
  CONJUR_FOLLOWER_CERTIFICATE: |
  <contents of dap-follower.indented-pem>
```

Figure 5: DAP service config map

Cut & paste the above into a text file named dap-config-map-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- CONJUR_ACCOUNT
- CONJUR_MASTER_HOSTNAME
- CLUSTER_AUTHN_ID

Paste the contents of each .indented-pem file under each certificate label. Then apply the manifest with kubectl.

```
kubectl apply -f dap-config-map-manifest.yaml -n cyberark
```

Step F2d: Create Follower config map

```
---
apiVersion: v1
kind: ConfigMap
metadata:
  name: follower-config
data:
  FOLLOWER_HOSTNAME: conjur-follower # this should be the same value as the service name
  SEED_FILE_DIR: /tmp/seedfile
  CONJUR_SEED_FILE_URL: https://{{ CONJUR_MASTER_HOSTNAME }}/configuration/{{ CONJUR_ACCOUNT }}/seed/follower
  CONJUR_AUTHN_LOGIN: host/{{ CLUSTER_AUTHN_ID }}/dap-authn-service
  CONJUR_AUTHENTICATORS: {{ CONJUR_AUTHENTICATORS }}
```

Figure 6: Follower config map

Cut & paste the above into a text file named follower-config-map-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- CONJUR_MASTER_HOSTNAME
- CONJUR_ACCOUNT

- CLUSTER_AUTHN_ID
- CONJUR_AUTHENTICATORS

Then apply the manifest with kubectl.

```
kubectl apply -f follower-config-map-manifest.yaml -n cyberark
```

Step F2e: Apply Follower deployment manifest

```
---
apiVersion: v1
kind: Service
metadata:
  name: conjur-follower
  labels:
    app: conjur-follower
spec:
  ports:
  - port: 443
    name: https
  selector:
    app: conjur-follower
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  name: conjur-follower
spec:
  replicas: 1
  template:
    metadata:
      labels:
        app: conjur-follower
        name: conjur-follower
        role: follower
    spec:
      serviceAccountName: dap-authn-service
      volumes:
      - name: seedfile
        emptyDir:
          medium: Memory
      - name: conjur-token
        emptyDir:
          medium: Memory
```

```
      initContainers:
      - name: authenticator
        image: {{ CONJUR_SEED_FETCHER_IMAGE }}
        imagePullPolicy: IfNotPresent
        env:

        ## values from metadata ##
          - name: MY_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name

          - name: MY_POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath: metadata.namespace

          - name: MY_POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP

        ## values from Conjur config map ##
          - name: AUTHENTICATOR_ID
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CLUSTER_AUTHN_ID

          - name: CONJUR_ACCOUNT
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_ACCOUNT

          - name: CONJUR_SSL_CERTIFICATE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_MASTER_CERTIFICATE

        ## values from Follower config map ##
          - name: FOLLOWER_HOSTNAME
            valueFrom:
              configMapKeyRef:
                name: follower-config
                key: FOLLOWER_HOSTNAME

          - name: SEEDFILE_DIR
            valueFrom:
              configMapKeyRef:
                name: follower-config
                key: SEED_FILE_DIR

          - name: CONJUR_SEED_FILE_URL
            valueFrom:
              configMapKeyRef:
                name: follower-config
                key: CONJUR_SEED_FILE_URL

          - name: CONJUR_AUTHN_LOGIN
            valueFrom:
              configMapKeyRef:
                name: follower-config
                key: CONJUR_AUTHN_LOGIN

        resources:
          requests:
            cpu: "100m"
            memory: 32Mi
          limits:
            cpu: "100m"
            memory: 32Mi
        volumeMounts:
          - name: seedfile
            mountPath: /tmp/seedfile
          - name: conjur-token
            mountPath: /run/conjur
```

```
      containers:
      - name: conjur-appliance
        image: {{ CONJUR_APPLIANCE_IMAGE }}
        command: ["/tmp/seedfile/start-
follower.sh"]
        imagePullPolicy: IfNotPresent
        env:

          # from FOLLOWER_CONFIG_MAP
          - name: SEEDFILE_DIR
            valueFrom:
              configMapKeyRef:
                name: follower-config
                key: SEED_FILE_DIR

          - name: CONJUR_AUTHENTICATORS
            valueFrom:
              configMapKeyRef:
                name: follower-config
                key: CONJUR_AUTHENTICATORS
        ports:
        - containerPort: 443
          name: https
        - containerPort: 5432
          name: pg-main
        - containerPort: 1999
          name: pg-audit
        readinessProbe:
          httpGet:
            path: /health
            port: 443
            scheme: HTTPS
          initialDelaySeconds: 15
          timeoutSeconds: 5
        resources:
          requests:
            cpu: "500m"
            memory: 2Gi
          limits:
            cpu: "500m"
            memory: 2Gi
        volumeMounts:
          - name: seedfile
            mountPath: /tmp/seedfile
            readOnly: true
```

Figure 7: Follower deployment manifest

Cut & paste the above three columns sequentially into a text file named follower-deployment-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- CONJUR_SEED_FETCHER_IMAGE
- CONJUR_APPLIANCE_IMAGE

These image names should refer to the images in the registry. Then apply the manifest with kubectl.

```
kubectl apply -f follower-deployment-manifest.yaml -n cyberark
```

Step F2f: Debugging Follower deployment

When deployed, a Follower pod will self-initialize from the DAP Master, a process that takes upwards of 30 seconds depending on processor speed and the amount of CPU resources specified in the manifest. The Readiness probe will show the pod is ready once initialization completes.

If the Follower shows errors or fails to start, start from the pod and work back to the DAP master, checking the following:

On K8s configuration host:

- **`kubectl get events -n cyberark`**
  Ensure the seed-fetcher init container started without errors.
    - Image pull errors
        - check the image name spelling in the manifest and ensure the image was pushed to the registry and referenced correctly.
        - Ensure the user deploying the Follower has correct privileges to and can login to the registry.
    - Resource constraints (e.g. insufficient memory)
        - Get a quota exception for the namespace/deployment
        - Apply taints/tolerances to schedule Follower on a different node
- **`kubectl logs <follower-pod-name> -c authenticator -n cyberark`**
  Ensure seed-fetcher started and authenticates successfully.
- **`kubectl logs <follower-pod-name> -c authenticator -n cyberark`**
  Ensure seed-fetcher started and authenticates successfully.

On DAP Master host:

- **`docker logs <master-container-name> --since 1m`**
  Display last minute's worth of the DAP Master log to trace seed-fetcher authentication at server. Ensure authentication requests appear in the log. If not, possible network connection issue (firewall, security groups, etc.) or certificate validation failure.
- **`docker exec <master-container-name> evoke variable list CONJUR_AUTHENTICATORS`**
  Ensure authn-k8s/$CLUSTER_AUTHN_ID is present in list and spelled correctly.

Application Configuration for K8s Authentication

Cluster Admin workflow overview

- Step A1a: Edit & apply namespace admin RBAC manifests
- Step A1b: Edit & apply namespace RBAC manifest
- Step A1c: Edit & apply secrets provider RBAC manifest (if using secrets provider for K8s)

Cluster Admin task detail

If using cluster RBAC, login as a cluster administrator.

Step A1a: Edit & apply namespace admin RBAC manifests (skip if not using user RBAC)

```
---
# Grant {{ APP_NAMESPACE_ADMIN }} read-only access to the DAP config map
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dap-cm-access
subjects:
- kind: User
  name: {{ APP_NAMESPACE_ADMIN }}
roleRef:
  kind: ClusterRole
  name: dap-cm-access-role
  apiGroup: rbac.authorization.k8s.io
```

Commented [DM1]: Done

Figure 8: DAP config map access for app namespace admin manifest

Cut & paste the above into a text file named for the application's namespace, i.e. $APP_NAMESPACE_NAME-cm-access-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-cm-access-manifest.yaml -n cyberark
```

```
---
# Grant {{ APP_NAMESPACE_ADMIN }} namespace admin privileges
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: namespace-admin
  namespace: {{ APP_NAMESPACE_NAME }}
subjects:
- kind: User
  name: {{ APP_NAMESPACE_ADMIN }}
roleRef:
  kind: ClusterRole
  name: admin
  apiGroup: rbac.authorization.k8s.io
```

Figure 9: App namespace admin manifest

Cut & paste the above into a text file named for the application's namespace, i.e. $APP_NAMESPACE_NAME-admin-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-admin-manifest.yaml -n $APP_NAMESPACE_NAME
```

Step A1b: Edit & apply namespace RBAC manifest

```
---
# Define project namespace
apiVersion: v1
kind: Namespace
metadata:
  name: {{ APP_NAMESPACE_NAME }}
  labels:
    name: {{ APP_NAMESPACE_NAME }}

---
# Grant the authentication service access to pods in {{ APP_NAMESPACE_NAME }} namespace
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: dap-authn-service
  namespace: {{ APP_NAMESPACE_NAME }}
subjects:
- kind: ServiceAccount
  name: dap-authn-service
  namespace: cyberark
roleRef:
  kind: ClusterRole
  name: dap-authn-role
  apiGroup: rbac.authorization.k8s.io
```

Figure 10: App namespace RBAC manifest

Cut & paste the above into a text file named for the application's namespace, i.e. $APP_NAMESPACE_NAME-rbac-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-rbac-manifest.yaml -n $APP_NAMESPACE_NAME
```

Step A1c: Edit & apply secrets provider RBAC manifest (if using secrets provider for K8s)

```
---
# Service account role & binding for K8s secrets injection
apiVersion: v1
kind: ServiceAccount
metadata:
  name: {{ APP_NAMESPACE_NAME }}

# Cluster role to enable reading and patching of k8s secrets
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: secrets-access
rules:
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: [ "get", "patch" ]

# Grants {{ APP_NAMESPACE_NAME }} service account ability to read and patch k8s secrets in namespace
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  namespace: {{ APP_NAMESPACE_NAME }}
  name: secrets-access-binding
subjects:
  - kind: ServiceAccount
    namespace: {{ APP_NAMESPACE_NAME }}
    name: default
roleRef:
  kind: ClusterRole
  name: secrets-access
  apiGroup: rbac.authorization.k8s.io
```

Figure 11: Secrets Provider RBAC manifest

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-provider-rbac-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-provider-rbac-manifest.yaml -n $APP_NAMESPACE_NAME
```

Application namespace admin workflow overview

- Step A2a: Load DAP application policies
- Step A2b: Deploy images to the registry
- Step A2c: Copy DAP config map to app namespace
- Step A2d: Create application config map
- Step A2e: Application deployment with authenticator sidecar
- Step A2f: Application deployment with authenticator init container
- Step A2g: Application deployment with Secrets Provider for K8s
- **Error! Reference source not found.**

Application namespace Admin task detail

If using cluster RBAC, login as the application namespace admin.

Step A2a: Load DAP application policies

Application identities policy template

```
---
# This policy whitelists three identities permitted to authenticate
# to the authn-k8s endpoint configured for the cluster.

- !host
  id: app-example-sidecar
  annotations:
    authn-k8s/namespace: {{ APP_NAMESPACE_NAME }}
    authn-k8s/service-account: app-example-sidecar
    authn-k8s/authentication-container-name: authenticator
    kubernetes: "true"

- !host
  id: app-example-init
  annotations:
    authn-k8s/namespace: {{ APP_NAMESPACE_NAME }}
    authn-k8s/service-account: app-example-init
    authn-k8s/authentication-container-name: authenticator
    kubernetes: "true"

- !host
  id: app-example-provider
  annotations:
    authn-k8s/namespace: {{ APP_NAMESPACE_NAME }}
    authn-k8s/authentication-container-name: secrets-provider
    kubernetes: "true"

# Policy to aggregate application identities for namespace
- !policy
  id: {{ CLUSTER_AUTHN_ID }}/{{ APP_NAMESPACE_NAME }}
  annotations:
    description: Identities permitted to authenticate
  body:
  - !group
    id: apps
    annotations:
      description: Group of identities permitted to call authn svc

# Grant apps group role to identities
- !grant
  role: !group {{ CLUSTER_AUTHN_ID }}/{{ APP_NAMESPACE_NAME }}/apps
  members:
  - !host app-example-sidecar
  - !host app-example-init
  - !host app-example-provider

# Grant authn-k8s consumer role to apps group
- !grant
  roles:
  - !group conjur/authn-k8s/{{ CLUSTER_AUTHN_ID }}/consumers
  members:
  - !group {{ CLUSTER_AUTHN_ID }}/{{ APP_NAMESPACE_NAME }}/apps
```

Figure 12: Application identities definitions policy

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-identities-policy.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME
- CLUSTER_AUTHN_ID

Then load the policy with the load_policy.sh script.

```
./load_policy.sh root $APP_NAMESPACE_NAME-identities-policy.yaml
```

Test secrets policy template

```
---
# Define a couple of test secrets and a group w/ read-only access
- !policy
  id: test-secrets
  body:
  - &variables
    - !variable db-username
    - !variable db-password
  - !group consumers

  # Permit consumers to read (list) and execute (fetch) test secrets
  - !permit
    privileges: [ read, execute ]
    role: !group consumers
    resources: *variables

# Grant read-only access to test-secrets to application group
- !grant
  roles:
  - !group test-secrets/consumers
#  - !group vault/lobuser/safe/delegation/consumers    # Example of CyberArk EPV safe access grant
  members:
  - !group {{ CLUSTER_AUTHN_ID }}/{{ APP_NAMESPACE_NAME }}/apps
```

Figure 13: Test secrets policy

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-test-secrets-policy.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME
- CLUSTER_AUTHN_ID

Then apply the manifest with the load_policy.sh script.

```
./load_policy.sh root $APP_NAMESPACE_NAME-test-secrets-policy.yaml
```

Use the get_set.sh script to set values for the secrets just created.

```
./get_set.sh set test-secrets/db-username this-is-the-username
./get_set.sh set test-secrets/db-password 1234567890
```

Step A2b: Deploy images to the registry

- Application image(s) – these are often pushed as part of build pipeline, but if not tag with the value of $APP_IMAGE and push to the registry.
- Authenticator client – pull from Docker hub or load from tarfile, tag with the value of $AUTHENTICATOR_IMAGE and push to the registry.
- Secrets provider for K8s – pull from Docker hub or load from tarfile, tag with the value of $SECRETS_PROVIDER_IMAGE and push to the registry.

Step A2c: Copy DAP config map to app namespace

The dap-config config map contains non-secret resources needed to connect to the DAP service. Use kubectl & sed to extract the config map yaml, replace the “cyberark” namespace value with the application namespace name, and create a copy of the config map in the application namespace.

```
kubectl get cm dap-config -n cyberark -o yaml \
    | sed "s/namespace: cyberark/namespace: $APP_NAMESPACE_NAME/" \
    | kubectl create -f - -n $APP_NAMESPACE_NAME
```

Step A2d: Create application config map

Creates a config map containing the authentication URL for the cluster and identities enabled for authentication.

```
kubectl create configmap app-config \
        -n $APP_NAMESPACE_NAME \
        --from-literal=conjur-authn-url=$CONJUR_AUTHENTICATOR_URL \
        --from-literal=conjur-authn-login-sidecar=host/app-example-sidecar \
        --from-literal=conjur-authn-login-init=host/app-example-init \
        --from-literal=conjur-authn-login-provider=host/app-example-provider
```

Step A2e: Application deployment with authenticator sidecar

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-example-sidecar
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  labels:
    app: app-example-sidecar
  name: app-example-sidecar
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app-example-sidecar
  template:
    metadata:
      labels:
        app: app-example-sidecar
    spec:
      serviceAccountName: app-example-
sidecar
```

```
      containers:
      - image: {{ APP_IMAGE }}
        imagePullPolicy: IfNotPresent
        name: test-app
        env:

        # values from DAP config map
          - name: CONJUR_VERSION
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_VERSION
          - name: CONJUR_ACCOUNT
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_ACCOUNT
          - name: CONJUR_APPLIANCE_URL
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_APPLIANCE_URL
          - name: CONJUR_SSL_CERTIFICATE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_FOLLOWER_CERTIFICATE
          - name: CONJUR_AUTHN_TOKEN_FILE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_AUTHN_TOKEN_FILE
        resources:
          requests:
            cpu: "300m"
            memory: "250Mi"
          limits:
            cpu: "300m"
            memory: "250Mi"
        volumeMounts:
          - mountPath: /run/conjur
            name: conjur-access-token
            readOnly: true
```

```
      - image: {{ AUTHENTICATOR_IMAGE }}
        imagePullPolicy: IfNotPresent
        name: authenticator
        env:

        # hardcoded values
          - name: CONTAINER_MODE
            value: sidecar

        # values from pod metadata
          - name: MY_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: MY_POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath:
metadata.namespace
          - name: MY_POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP

        # values from DAP config map
          - name: CONJUR_VERSION
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_VERSION
          - name: CONJUR_ACCOUNT
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_ACCOUNT
          - name: CONJUR_SSL_CERTIFICATE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_FOLLOWER_CERTIFICATE
        # values from app config map
          - name: CONJUR_AUTHN_URL
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: conjur-authn-url
          - name: CONJUR_AUTHN_LOGIN
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: conjur-authn-login-
sidecar
        resources:
          requests:
            cpu: "50m"
            memory: "16Mi"
          limits:
            cpu: "50m"
            memory: "16Mi"
        volumeMounts:
          - mountPath: /run/conjur
            name: conjur-access-token
      imagePullSecrets:
        - name: dockerpullsecret
      volumes:
        - name: conjur-access-token
          emptyDir:
            medium: Memory
```

Figure 14: Application deployment manifest – authenticator running as sidecar

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-app-sidecar-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_IMAGE
- AUTHENTICATOR_IMAGE

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-app-sidecar-manifest.yaml -n $APP_NAMESPACE_NAME
```

Verify correct execution by echoing the access token in the application container.

```
kubectl exec \
     $(kubectl get pods --no-headers -n $APP_NAMESPACE_NAME \
     | grep app-example-sidecar | cut -d" " -f1) \
     cat /run/conjur/access-token -n $APP_NAMESPACE_NAME && echo
```

Step A2f: Application deployment with authenticator init container

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: app-example-init
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  labels:
    app: app-example-init
  name: app-example-init
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app-example-init
  template:
    metadata:
      labels:
        app: app-example-init
    spec:
      serviceAccountName: app-example-
init
```

```
      containers:
      - image: {{ APP_IMAGE }}
        imagePullPolicy: IfNotPresent
        name: test-app
        env:

        # values from DAP config map
          - name: CONJUR_VERSION
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_VERSION
          - name: CONJUR_ACCOUNT
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_ACCOUNT
          - name: CONJUR_APPLIANCE_URL
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_APPLIANCE_URL
          - name: CONJUR_SSL_CERTIFICATE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_FOLLOWER_CERTIFICATE
          - name:
CONJUR_AUTHN_TOKEN_FILE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_AUTHN_TOKEN_FILE
        resources:
          requests:
            cpu: "300m"
            memory: "250Mi"
          limits:
            cpu: "300m"
            memory: "250Mi"
        volumeMounts:
          - mountPath: /run/conjur
            name: conjur-access-token
            readOnly: true
```

```
      initContainers:
      - image: {{ AUTHENTICATOR_IMAGE }}
        imagePullPolicy: IfNotPresent
        name: authenticator
        env:

        # hardcoded values
          - name: CONTAINER_MODE
            value: init

        # values from pod metadata
          - name: MY_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: MY_POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath:
metadata.namespace
          - name: MY_POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP

        # values from DAP config map
          - name: CONJUR_VERSION
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_VERSION
          - name: CONJUR_ACCOUNT
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_ACCOUNT
          - name: CONJUR_SSL_CERTIFICATE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_FOLLOWER_CERTIFICATE
        # values from app config map
          - name: CONJUR_AUTHN_URL
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: conjur-authn-url
          - name: CONJUR_AUTHN_LOGIN
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: conjur-authn-login-
init

        resources:
          requests:
            cpu: "50m"
            memory: "16Mi"
          limits:
            cpu: "50m"
            memory: "16Mi"
        volumeMounts:
          - mountPath: /run/conjur
            name: conjur-access-token
      imagePullSecrets:
        - name: dockerpullsecret
      volumes:
        - name: conjur-access-token
          emptyDir:
            medium: Memory
```

Figure 15: Application deployment manifest – authenticator running as init container

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-app-init-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_IMAGE
- AUTHENTICATOR_IMAGE

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-app-init-manifest.yaml -n $APP_NAMESPACE_NAME
```

Notice how the manifests for the sidecar and init container deployments are practically identical. The only significant differences are the initContainers: heading and the CONTAINER_MODE value for the authenticator image.

Verify correct execution by echoing the access token in the application container.

```
kubectl exec \
     $(kubectl get pods --no-headers -n $APP_NAMESPACE_NAME \
     | grep app-example-init | cut -d" " -f1) \
     cat /run/conjur/access-token -n $APP_NAMESPACE_NAME && echo
```

Step A2g: Application deployment with Secrets Provider for K8s

The Secrets Provider for K8s retrieves secrets and injects their values into designated Kubernetes secrets. The Provider uses a Kubernetes secret containing a list of key/value pairs where the keys are mapped to the names of DAP secrets to retrieve. At runtime, the Provider authenticates the application pod, iterates over the map and patches the Kubernetes secrets to replace the DAP variable names with their values.

```
---
apiVersion: v1
kind: Secret
metadata:
  name: db-credentials
type: Opaque
data:
  # base64-encoded "myappDB"
  DBName:   bXlhcHBEQg==
stringData:
  conjur-map: |-
    username: test-secrets/db-username
    password: test-secrets/db-password
```

Figure 16: Kubernetes secret manifest with conjur-map of key/variable-name pairs

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-k8s-secrets-manifest.yaml, taking care to capture all whitespace. Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-k8s-secrets-manifest.yaml -n $APP_NAMESPACE_NAME
```

```
---
apiVersion: apps/v1beta1
kind: Deployment
metadata:
  labels:
    app: app-example-provider
  name: app-example-provider
spec:
  replicas: 1
  selector:
    matchLabels:
      app: app-example-provider
  template:
    metadata:
      labels:
        app: app-example-provider
    spec:
```

```
      containers:
      - image: {{ APP_IMAGE }}
        imagePullPolicy: IfNotPresent
        name: test-app
        env:

          - name: DB_UNAME
            valueFrom:
              secretKeyRef:
                name: db-credentials
                key: username

          - name: DB_PASSWORD
            valueFrom:
              secretKeyRef:
                name: db-credentials
                key: password

        volumeMounts:
          - name: secret-volume
            mountPath: /etc/secret-volume

        resources:
          requests:
            cpu: 50m
            memory: 250Mi
          limits:
            cpu: 50m
            memory: 250Mi
```

```
      initContainers:
      - image: {{ SECRETS_PROVIDER_IMAGE
}}
        imagePullPolicy: IfNotPresent
        name: secrets-provider
        env:
        # hardcoded values
          - name: CONTAINER_MODE
            value: init

        # values from pod metadata
          - name: MY_POD_NAME
            valueFrom:
              fieldRef:
                fieldPath: metadata.name
          - name: MY_POD_NAMESPACE
            valueFrom:
              fieldRef:
                fieldPath:
metadata.namespace
          - name: MY_POD_IP
            valueFrom:
              fieldRef:
                fieldPath: status.podIP

        # values from DAP config map
          - name: CONJUR_VERSION
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_VERSION
          - name: CONJUR_APPLIANCE_URL
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_APPLIANCE_URL
          - name: CONJUR_ACCOUNT
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key: CONJUR_ACCOUNT
          - name: CONJUR_SSL_CERTIFICATE
            valueFrom:
              configMapKeyRef:
                name: dap-config
                key:
CONJUR_FOLLOWER_CERTIFICATE

        # values from app config map
          - name: CONJUR_AUTHN_URL
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: conjur-authn-url
          - name: CONJUR_AUTHN_LOGIN
            valueFrom:
              configMapKeyRef:
                name: app-config
                key: conjur-authn-login-
provider

          - name: K8S_SECRETS
            value: db-credentials

          - name: SECRETS_DESTINATION
            value: k8s_secrets

          - name: DEBUG
            value: "true"

        resources:
          requests:
            cpu: 50m
            memory: 16Mi
          limits:
            cpu: 50m
            memory: 16Mi

      volumes:
        - name: secret-volume
          secret:
            secretName: db-credentials

      imagePullSecrets:
        - name: dockerpullsecret
```

Figure 17: Application deployment manifest – secrets provider for K8s

Cut & paste the above into a text file named $APP_NAMESPACE_NAME-app-provider-manifest.yaml, taking care to capture all whitespace. Substitute the correct values for:

- APP_NAMESPACE_NAME
- APP_IMAGE
- SECRETS_PROVIDER_IMAGE

Then apply the manifest with kubectl.

```
kubectl apply -f $APP_NAMESPACE_NAME-app-provider-manifest.yaml -n $APP_NAMESPACE_NAME
```

Verify correct execution by echoing the database password in the application container.

```
kubectl exec \
     $(kubectl get pods --no-headers -n $APP_NAMESPACE_NAME \
     | grep app-example-provider | cut -d" " -f1) \
     cat /etc/secret-volume/password -n $APP_NAMESPACE_NAME && echo
```