Secrets

A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key. Such information might otherwise be put in a Pod specification or in a container image. Using a Secret means that you don't need to include confidential data in your application code.

Because Secrets can be created independently of the Pods that use them, there is less risk of the Secret (and its data) being exposed during the workflow of creating, viewing, and editing Pods. Kubernetes, and applications that run in your cluster, can also take additional precautions with Secrets, such as avoiding writing secret data to nonvolatile storage.

Secrets are similar to ConfigMaps but are specifically intended to hold confidential data.

> **Caution:**
> Kubernetes Secrets are, by default, stored unencrypted in the API server's underlying data store (etcd). Anyone with API access can retrieve or modify a Secret, and so can anyone with access to etcd. Additionally, anyone who is authorized to create a Pod in a namespace can use that access to read any Secret in that namespace; this includes indirect access such as the ability to create a Deployment.
>
> In order to safely use Secrets, take at least the following steps:
>
> 1. Enable Encryption at Rest for Secrets.
> 2. Enable or configure RBAC rules with least-privilege access to Secrets.
> 3. Restrict Secret access to specific containers.
> 4. Consider using external Secret store providers.
>
> For more guidelines to manage and improve the security of your Secrets, refer to Good practices for Kubernetes Secrets.

See Information security for Secrets for more details.

Uses for Secrets

There are three main ways for a Pod to use a Secret:

- As files in a volume mounted on one or more of its containers.
- As container environment variable.
- By the kubelet when pulling images for the Pod.

The Kubernetes control plane also uses Secrets; for example, bootstrap token Secrets are a mechanism to help automate node registration.

Alternatives to Secrets

Rather than using a Secret to protect confidential data, you can pick from alternatives.

Here are some of your options:

- If your cloud-native component needs to authenticate to another application that you know is running within the same Kubernetes cluster, you can use a ServiceAccount and its tokens to identify your client.

- There are third-party tools that you can run, either within or outside your cluster, that provide secrets management. For example, a service that Pods access over HTTPS, that reveals a secret if the client correctly authenticates (for example, with a ServiceAccount token).
- For authentication, you can implement a custom signer for X.509 certificates, and use CertificateSigningRequests to let that custom signer issue certificates to Pods that need them.
- You can use a device plugin to expose node-local encryption hardware to a specific Pod. For example, you can schedule trusted Pods onto nodes that provide a Trusted Platform Module, configured out-of-band.

You can also combine two or more of those options, including the option to use Secret objects themselves.

For example: implement (or deploy) an operator that fetches short-lived session tokens from an external service, and then creates Secrets based on those short-lived session tokens. Pods running in your cluster can make use of the session tokens, and operator ensures they are valid. This separation means that you can run Pods that are unaware of the exact mechanisms for issuing and refreshing those session tokens.

Types of Secret

When creating a Secret, you can specify its type using the `type` field of the Secret resource, or certain equivalent `kubectl` command line flags (if available). The Secret type is used to facilitate programmatic handling of the Secret data.

Kubernetes provides several built-in types for some common usage scenarios. These types vary in terms of the validations performed and the constraints Kubernetes imposes on them.

Built-in Type	Usage
`Opaque`	arbitrary user-defined data
`kubernetes.io/service-account-token`	ServiceAccount token
`kubernetes.io/dockercfg`	serialized `~/.dockercfg` file
`kubernetes.io/dockerconfigjson`	serialized `~/.docker/config.json` file
`kubernetes.io/basic-auth`	credentials for basic authentication
`kubernetes.io/ssh-auth`	credentials for SSH authentication
`kubernetes.io/tls`	data for a TLS client or server
`bootstrap.kubernetes.io/token`	bootstrap token data

You can define and use your own Secret type by assigning a non-empty string as the `type` value for a Secret object (an empty string is treated as an `Opaque` type).

Kubernetes doesn't impose any constraints on the type name. However, if you are using one of the built-in types, you must meet all the requirements defined for that type.

If you are defining a type of secret that's for public use, follow the convention and structure the secret type to have your domain name before the name, separated by a `/`. For example: `cloud-hosting.example.net/cloud-api-credentials`.

Opaque secrets

`Opaque` is the default Secret type if omitted from a Secret configuration file. When you create a Secret using `kubectl`, you will use the `generic` subcommand to indicate an `Opaque` Secret type. For example, the following command creates an empty Secret of type `Opaque`.

```
kubectl create secret generic empty-secret
kubectl get secret empty-secret
```

The output looks like:

```
NAME           TYPE     DATA   AGE
empty-secret   Opaque   0      2m6s
```

The `DATA` column shows the number of data items stored in the Secret. In this case, `0` means you have created an empty Secret.

Service account token Secrets

A `kubernetes.io/service-account-token` type of Secret is used to store a token credential that identifies a service account.

> **Note:**
> Versions of Kubernetes before v1.22 automatically created credentials for accessing the Kubernetes API. This older mechanism was based on creating token Secrets that could then be mounted into running Pods. In more recent versions, including Kubernetes v1.27, API credentials are obtained directly by using the TokenRequest API, and are mounted into Pods using a projected volume. The tokens obtained using this method have bounded lifetimes, and are automatically invalidated when the Pod they are mounted into is deleted.
>
> You can still manually create a service account token Secret; for example, if you need a token that never expires. However, using the TokenRequest subresource to obtain a token to access the API is recommended instead. You can use the `kubectl create token` command to obtain a token from the `TokenRequest` API.

You should only create a service account token Secret object if you can't use the `TokenRequest` API to obtain a token, and the security exposure of persisting a non-expiring token credential in a readable API object is acceptable to you.

When using this Secret type, you need to ensure that the `kubernetes.io/service-account.name` annotation is set to an existing service account name. If you are creating both the ServiceAccount and the Secret objects, you should create the ServiceAccount object first.

After the Secret is created, a Kubernetes controller fills in some other fields such as the `kubernetes.io/service-account.uid` annotation, and the `token` key in the `data` field, which is populated with an authentication token.

The following example configuration declares a service account token Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-sa-sample
  annotations:
    kubernetes.io/service-account.name: "sa-name"
type: kubernetes.io/service-account-token
data:
  # You can include additional key value pairs as you do with Opaque Secrets
  extra: YmFyCg==
```

After creating the Secret, wait for Kubernetes to populate the `token` key in the `data` field.

See the ServiceAccount documentation for more information on how service accounts work. You can also check the `automountServiceAccountToken` field and the `serviceAccountName` field of the `Pod` for information on referencing service account credentials from within Pods.

Docker config Secrets

You can use one of the following `type` values to create a Secret to store the credentials for accessing a container image registry:

- `kubernetes.io/dockercfg`
- `kubernetes.io/dockerconfigjson`

The `kubernetes.io/dockercfg` type is reserved to store a serialized `~/.dockercfg` which is the legacy format for configuring Docker command line. When using this Secret type, you have to ensure the Secret `data` field contains a `.dockercfg` key whose value is content of a `~/.dockercfg` file encoded in the base64 format.

The `kubernetes.io/dockerconfigjson` type is designed for storing a serialized JSON that follows the same format rules as the `~/.docker/config.json` file which is a new format for `~/.dockercfg` . When using this Secret type, the `data` field of the Secret object must contain a `.dockerconfigjson` key, in which the content for the `~/.docker/config.json` file is provided as a base64 encoded string.

Below is an example for a `kubernetes.io/dockercfg` type of Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-dockercfg
type: kubernetes.io/dockercfg
data:
  .dockercfg: |
    "<base64 encoded ~/.dockercfg file>"
```

> **Note:** If you do not want to perform the base64 encoding, you can choose to use the `stringData` field instead.

When you create these types of Secrets using a manifest, the API server checks whether the expected key exists in the `data` field, and it verifies if the value provided can be parsed as a valid JSON. The API server doesn't validate if the JSON actually is a Docker config file.

When you do not have a Docker config file, or you want to use `kubectl` to create a Secret for accessing a container registry, you can do:

```
kubectl create secret docker-registry secret-tiger-docker \
  --docker-email=tiger@acme.example \
  --docker-username=tiger \
  --docker-password=pass1234 \
  --docker-server=my-registry.example:5000
```

That command creates a Secret of type `kubernetes.io/dockerconfigjson`. If you dump the `.data.dockerconfigjson` field from that new Secret and then decode it from base64:

```
kubectl get secret secret-tiger-docker -o jsonpath='{.data.*}' | base64 -d
```

then the output is equivalent to this JSON document (which is also a valid Docker configuration file):

```
{
  "auths": {
    "my-registry.example:5000": {
      "username": "tiger",
      "password": "pass1234",
      "email": "tiger@acme.example",
      "auth": "dGlnZXI6cGFzczEyMzQ="
    }
  }
}
```

> **Note:** The `auth` value there is base64 encoded; it is obscured but not secret. Anyone who can read that Secret can learn the registry access bearer token.

Basic authentication Secret

The `kubernetes.io/basic-auth` type is provided for storing credentials needed for basic authentication. When using this Secret type, the `data` field of the Secret must contain one of the following two keys:

- `username`: the user name for authentication
- `password`: the password or token for authentication

Both values for the above two keys are base64 encoded strings. You can, of course, provide the clear text content using the `stringData` for Secret creation.

The following manifest is an example of a basic authentication Secret:

```
apiVersion: v1
kind: Secret
metadata:
  name: secret-basic-auth
type: kubernetes.io/basic-auth
stringData:
  username: admin # required field for kubernetes.io/basic-auth
  password: t0p-Secret # required field for kubernetes.io/basic-auth
```

The basic authentication Secret type is provided only for convenience. You can create an `Opaque` type for credentials used for basic authentication. However, using the defined and public Secret type (`kubernetes.io/basic-auth`) helps other people to understand the purpose of your Secret, and sets a convention for what key names to expect. The Kubernetes API verifies that the required keys are set for a Secret of this type.

SSH authentication secrets

The builtin type `kubernetes.io/ssh-auth` is provided for storing data used in SSH authentication. When using this Secret type, you will have to specify a `ssh-privatekey` key-value pair in the `data` (or `stringData`) field as the SSH credential to use.

The following manifest is an example of a Secret used for SSH public/private key authentication:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-ssh-auth
type: kubernetes.io/ssh-auth
data:
  # the data is abbreviated in this example
  ssh-privatekey: |
    MIIEpQIBAAKCAQEAulqb/Y ...
```

The SSH authentication Secret type is provided only for user's convenience. You could instead create an `Opaque` type Secret for credentials used for SSH authentication. However, using the defined and public Secret type (`kubernetes.io/ssh-auth`) helps other people to understand the purpose of your Secret, and sets a convention for what key names to expect. and the API server does verify if the required keys are provided in a Secret configuration.

> **Caution:** SSH private keys do not establish trusted communication between an SSH client and host server on their own. A secondary means of establishing trust is needed to mitigate "man in the middle" attacks, such as a `known_hosts` file added to a ConfigMap.

TLS secrets

Kubernetes provides a builtin Secret type `kubernetes.io/tls` for storing a certificate and its associated key that are typically used for TLS.

One common use for TLS secrets is to configure encryption in transit for an Ingress, but you can also use it with other resources or directly in your workload. When using this type of Secret, the `tls.key` and the `tls.crt` key must be provided in the `data` (or `stringData`) field of the Secret configuration, although the API server doesn't actually validate the values for each key.

The following YAML contains an example config for a TLS Secret:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: secret-tls
type: kubernetes.io/tls
data:
  # the data is abbreviated in this example
  tls.crt: |
    MIIC2DCCAcCgAwIBAgIBATANBgkqh ...
  tls.key: |
    MIIEpgIBAAKCAQEA7yn3bRHQ5FHMQ ...
```

The TLS Secret type is provided for user's convenience. You can create an `Opaque` for credentials used for TLS server and/or client. However, using the builtin Secret type helps ensure the consistency of Secret format in your project; the API server does verify if the required keys are provided in a Secret configuration.

When creating a TLS Secret using `kubectl`, you can use the `tls` subcommand as shown in the following example:

```
kubectl create secret tls my-tls-secret \
  --cert=path/to/cert/file \
  --key=path/to/key/file
```

The public/private key pair must exist before hand. The public key certificate for `--cert` must be DER format as per Section 5.1 of RFC 7468, and must match the given private key for `--key` (PKCS #8 in DER format; Section 11 of RFC 7468).

> **Note:**
> A kubernetes.io/tls Secret stores the Base64-encoded DER data for keys and certificates. If you're familiar with PEM format for private keys and for certificates, the base64 data are the same as that format except that you omit the initial and the last lines that are used in PEM.
>
> For example, for a certificate, you do **not** include `--------BEGIN CERTIFICATE-----` and `-------END CERTIFICATE----`.

Bootstrap token Secrets

A bootstrap token Secret can be created by explicitly specifying the Secret `type` to `bootstrap.kubernetes.io/token`. This type of Secret is designed for tokens used during the node bootstrap process. It stores tokens used to sign well-known ConfigMaps.

A bootstrap token Secret is usually created in the `kube-system` namespace and named in the form `bootstrap-token-<token-id>` where `<token-id>` is a 6 character string of the token ID.

As a Kubernetes manifest, a bootstrap token Secret might look like the following:

```
apiVersion: v1
kind: Secret
metadata:
  name: bootstrap-token-5emitj
  namespace: kube-system
type: bootstrap.kubernetes.io/token
data:
  auth-extra-groups: c3lzdGVtOmJvb3RzdHJhcHBlcnM6a3ViZWFkbTpkZWZhdWx0LW5vZGUtdG9r
  expiration: MjAyMC0wOS0xM1QwNDozOToxMFo=
  token-id: NWVtaXRq
  token-secret: a3E0Z2lodnN6emduMXAwcg==
  usage-bootstrap-authentication: dHJ1ZQ==
  usage-bootstrap-signing: dHJ1ZQ==
```

A bootstrap type Secret has the following keys specified under `data`:

- `token-id`: A random 6 character string as the token identifier. Required.
- `token-secret`: A random 16 character string as the actual token secret. Required.
- `description`: A human-readable string that describes what the token is used for. Optional.
- `expiration`: An absolute UTC time using RFC3339 specifying when the token should be expired. Optional.
- `usage-bootstrap-<usage>`: A boolean flag indicating additional usage for the bootstrap token.
- `auth-extra-groups`: A comma-separated list of group names that will be authenticated as in addition to the `system:bootstrappers` group.

The above YAML may look confusing because the values are all in base64 encoded strings. In fact, you can create an identical Secret using the following YAML:

```yaml
apiVersion: v1
kind: Secret
metadata:
  # Note how the Secret is named
  name: bootstrap-token-5emitj
  # A bootstrap token Secret usually resides in the kube-system namespace
  namespace: kube-system
type: bootstrap.kubernetes.io/token
stringData:
  auth-extra-groups: "system:bootstrappers:kubeadm:default-node-token"
  expiration: "2020-09-13T04:39:10Z"
  # This token ID is used in the name
  token-id: "5emitj"
  token-secret: "kq4gihvszzgn1p0r"
  # This token can be used for authentication
  usage-bootstrap-authentication: "true"
  # and it can be used for signing
  usage-bootstrap-signing: "true"
```

Working with Secrets

Creating a Secret

There are several options to create a Secret:

- Use `kubectl`
- Use a configuration file
- Use the Kustomize tool

Constraints on Secret names and data

The name of a Secret object must be a valid DNS subdomain name.

You can specify the `data` and/or the `stringData` field when creating a configuration file for a Secret. The `data` and the `stringData` fields are optional. The values for all keys in the `data` field have to be base64-encoded strings. If the conversion to base64 string is not desirable, you can choose to specify the `stringData` field instead, which accepts arbitrary strings as values.

The keys of `data` and `stringData` must consist of alphanumeric characters, `-`, `_` or `.`. All key-value pairs in the `stringData` field are internally merged into the `data` field. If a key appears in both the `data` and the `stringData` field, the value specified in the `stringData` field takes precedence.

Size limit

Individual secrets are limited to 1MiB in size. This is to discourage creation of very large secrets that could exhaust the API server and kubelet memory. However, creation of many smaller secrets could also exhaust memory. You can use a resource quota to limit the number of Secrets (or other resources) in a namespace.

Editing a Secret

You can edit an existing Secret unless it is immutable. To edit a Secret, use one of the following methods:

- Use `kubectl`
- Use a configuration file

You can also edit the data in a Secret using the Kustomize tool. However, this method creates a new `Secret` object with the edited data.

Depending on how you created the Secret, as well as how the Secret is used in your Pods, updates to existing `Secret` objects are propagated automatically to Pods that use the data. For more information, refer to Using Secrets as files from a Pod section.

Using a Secret

Secrets can be mounted as data volumes or exposed as environment variables to be used by a container in a Pod. Secrets can also be used by other parts of the system, without being directly exposed to the Pod. For example, Secrets can hold credentials that other parts of the system should use to interact with external systems on your behalf.

Secret volume sources are validated to ensure that the specified object reference actually points to an object of type Secret. Therefore, a Secret needs to be created before any Pods that depend on it.

If the Secret cannot be fetched (perhaps because it does not exist, or due to a temporary lack of connection to the API server) the kubelet periodically retries running that Pod. The kubelet also reports an Event for that Pod, including details of the problem fetching the Secret.

Optional Secrets

When you reference a Secret in a Pod, you can mark the Secret as *optional*, such as in the following example. If an optional Secret doesn't exist, Kubernetes ignores it.

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
    - name: mypod
      image: redis
      volumeMounts:
        - name: foo
          mountPath: "/etc/foo"
          readOnly: true
  volumes:
    - name: foo
      secret:
        secretName: mysecret
        optional: true
```

By default, Secrets are required. None of a Pod's containers will start until all non-optional Secrets are available.

If a Pod references a specific key in a non-optional Secret and that Secret does exist, but is missing the named key, the Pod fails during startup.

Using Secrets as files from a Pod

If you want to access data from a Secret in a Pod, one way to do that is to have Kubernetes make the value of that Secret be available as a file inside the filesystem of one or more of the Pod's containers.

For instructions, refer to Distribute credentials securely using Secrets.

When a volume contains data from a Secret, and that Secret is updated, Kubernetes tracks this and updates the data in the volume, using an eventually-consistent approach.

> **Note:** A container using a Secret as a subPath volume mount does not receive automated Secret updates.

The kubelet keeps a cache of the current keys and values for the Secrets that are used in volumes for pods on that node. You can configure the way that the kubelet detects changes from the cached values. The `configMapAndSecretChangeDetectionStrategy` field in the kubelet configuration controls which strategy the kubelet uses. The default strategy is `Watch`.

Updates to Secrets can be either propagated by an API watch mechanism (the default), based on a cache with a defined time-to-live, or polled from the cluster API server on each kubelet synchronisation loop.

As a result, the total delay from the moment when the Secret is updated to the moment when new keys are projected to the Pod can be as long as the kubelet sync period + cache propagation delay, where the cache propagation delay depends on the chosen cache type (following the same order listed in the previous paragraph, these are: watch propagation delay, the configured cache TTL, or zero for direct polling).

Using Secrets as environment variables

To use a Secret in an environment variable in a Pod:

1. For each container in your Pod specification, add an environment variable for each Secret key that you want to use to the `env[].valueFrom.secretKeyRef` field.
2. Modify your image and/or command line so that the program looks for values in the specified environment variables.

For instructions, refer to Define container environment variables using Secret data.

Invalid environment variables

If your environment variable definitions in your Pod specification are considered to be invalid environment variable names, those keys aren't made available to your container. The Pod is allowed to start.

Kubernetes adds an Event with the reason set to `InvalidVariableNames` and a message that lists the skipped invalid keys. The following example shows a Pod that refers to a Secret named `mysecret`, where `mysecret` contains 2 invalid keys: `1badkey` and `2alsobad`.

```
kubectl get events
```

The output is similar to:

```
LASTSEEN   FIRSTSEEN   COUNT     NAME            KIND      SUBOBJECT
0s         0s          1         dapi-test-pod   Pod
```

Container image pull secrets

If you want to fetch container images from a private repository, you need a way for the kubelet on each node to authenticate to that repository. You can configure *image pull secrets* to make this possible. These secrets are configured at the Pod level.

Using imagePullSecrets

The `imagePullSecrets` field is a list of references to secrets in the same namespace. You can use an `imagePullSecrets` to pass a secret that contains a Docker (or other) image registry password to the kubelet. The kubelet uses this information to pull a private image on behalf of your Pod. See the PodSpec API for more information about the `imagePullSecrets` field.

Manually specifying an imagePullSecret

You can learn how to specify `imagePullSecrets` from the container images documentation.

Arranging for imagePullSecrets to be automatically attached

You can manually create `imagePullSecrets`, and reference these from a ServiceAccount. Any Pods created with that ServiceAccount or created with that ServiceAccount by default, will get their `imagePullSecrets` field set to that of the service account. See Add ImagePullSecrets to a service account for a detailed explanation of that process.

Using Secrets with static Pods

You cannot use ConfigMaps or Secrets with static Pods.

Use cases

Use case: As container environment variables

You can create a Secret and use it to set environment variables for a container.

Use case: Pod with SSH keys

Create a Secret containing some SSH keys:

```
kubectl create secret generic ssh-key-secret --from-file=ssh-privatekey=/path/to/
```

The output is similar to:

```
secret "ssh-key-secret" created
```

You can also create a `kustomization.yaml` with a `secretGenerator` field containing ssh keys.

> **Caution:**
> Think carefully before sending your own SSH keys: other users of the cluster may have access to the Secret.
>
> You could instead create an SSH private key representing a service identity that you want to be accessible to all the users with whom you share the Kubernetes cluster, and that you can revoke if the credentials are compromised.

Now you can create a Pod which references the secret with the SSH key and consumes it in a volume:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
  labels:
    name: secret-test
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: ssh-key-secret
  containers:
    - name: ssh-test-container
      image: mySshImage
      volumeMounts:
        - name: secret-volume
          readOnly: true
          mountPath: "/etc/secret-volume"
```

When the container's command runs, the pieces of the key will be available in:

```
/etc/secret-volume/ssh-publickey
/etc/secret-volume/ssh-privatekey
```

The container is then free to use the secret data to establish an SSH connection.

Use case: Pods with prod / test credentials

This example illustrates a Pod which consumes a secret containing production credentials and another Pod which consumes a secret with test environment credentials.

You can create a `kustomization.yaml` with a `secretGenerator` field or run `kubectl create secret`.

```
kubectl create secret generic prod-db-secret --from-literal=username=produser --f
```

The output is similar to:

```
secret "prod-db-secret" created
```

You can also create a secret for test environment credentials.

```
kubectl create secret generic test-db-secret --from-literal=username=testuser --f
```

The output is similar to:

```
secret "test-db-secret" created
```

> **Note:**
> Special characters such as `$`, `\`, `*`, `=`, and `!` will be interpreted by your shell and require escaping.
>
> In most shells, the easiest way to escape the password is to surround it with single quotes (`'`). For example, if your actual password is `S!B\*d$zDsb=`, you should execute the command this way:

```
kubectl create secret generic dev-db-secret --from-literal=username=devuser -
```

You do not need to escape special characters in passwords from files (`--from-file`).

Now make the Pods:

```
cat <<EOF > pod.yaml
apiVersion: v1
kind: List
items:
- kind: Pod
  apiVersion: v1
  metadata:
    name: prod-db-client-pod
    labels:
      name: prod-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: prod-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
- kind: Pod
  apiVersion: v1
  metadata:
    name: test-db-client-pod
    labels:
      name: test-db-client
  spec:
    volumes:
    - name: secret-volume
      secret:
        secretName: test-db-secret
    containers:
    - name: db-client-container
      image: myClientImage
      volumeMounts:
      - name: secret-volume
        readOnly: true
        mountPath: "/etc/secret-volume"
EOF
```

Add the pods to the same `kustomization.yaml` :

```
cat <<EOF >> kustomization.yaml
resources:
- pod.yaml
EOF
```

Apply all those objects on the API server by running:

```
kubectl apply -k .
```

Both containers will have the following files present on their filesystems with the values for each container's environment:

```
/etc/secret-volume/username
/etc/secret-volume/password
```

Note how the specs for the two Pods differ only in one field; this facilitates creating Pods with different capabilities from a common Pod template.

You could further simplify the base Pod specification by using two service accounts:

1. `prod-user` with the `prod-db-secret`
2. `test-user` with the `test-db-secret`

The Pod specification is shortened to:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: prod-db-client-pod
  labels:
    name: prod-db-client
spec:
  serviceAccount: prod-db-client
  containers:
    - name: db-client-container
      image: myClientImage
```

Use case: dotfiles in a secret volume

You can make your data "hidden" by defining a key that begins with a dot. This key represents a dotfile or "hidden" file. For example, when the following secret is mounted into a volume, `secret-volume`:

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: dotfile-secret
data:
  .secret-file: dmFsdWUtMg0KDQo=
---
apiVersion: v1
kind: Pod
metadata:
  name: secret-dotfiles-pod
spec:
  volumes:
    - name: secret-volume
      secret:
        secretName: dotfile-secret
  containers:
    - name: dotfile-test-container
      image: registry.k8s.io/busybox
      command:
        - ls
        - "-l"
        - "/etc/secret-volume"
      volumeMounts:
        - name: secret-volume
          readOnly: true
          mountPath: "/etc/secret-volume"
```

The volume will contain a single file, called `.secret-file`, and the `dotfile-test-container` will have this file present at the path `/etc/secret-volume/.secret-file`.

> **Note:** Files beginning with dot characters are hidden from the output of `ls -l`; you must use `ls -la` to see them when listing directory contents.

Use case: Secret visible to one container in a Pod

Consider a program that needs to handle HTTP requests, do some complex business logic, and then sign some messages with an HMAC. Because it has complex application logic, there might be an unnoticed remote file reading exploit in the server, which could expose the private key to an attacker.

This could be divided into two processes in two containers: a frontend container which handles user interaction and business logic, but which cannot see the private key; and a signer container that can see the private key, and responds to simple signing requests from the frontend (for example, over localhost networking).

With this partitioned approach, an attacker now has to trick the application server into doing something rather arbitrary, which may be harder than getting it to read a file.

Immutable Secrets

FEATURE STATE: `Kubernetes v1.21 [stable]`

Kubernetes lets you mark specific Secrets (and ConfigMaps) as *immutable*. Preventing changes to the data of an existing Secret has the following benefits:

- protects you from accidental (or unwanted) updates that could cause applications outages
- (for clusters that extensively use Secrets - at least tens of thousands of unique Secret to Pod mounts), switching to immutable Secrets improves the performance of your cluster by significantly reducing load on kube-apiserver. The kubelet does not need to maintain a [watch] on any Secrets that are marked as immutable.

Marking a Secret as immutable

You can create an immutable Secret by setting the `immutable` field to `true`. For example,

```yaml
apiVersion: v1
kind: Secret
metadata: ...
data: ...
immutable: true
```

You can also update any existing mutable Secret to make it immutable.

> **Note:** Once a Secret or ConfigMap is marked as immutable, it is *not* possible to revert this change nor to mutate the contents of the `data` field. You can only delete and recreate the Secret. Existing Pods maintain a mount point to the deleted Secret - it is recommended to recreate these pods.

Information security for Secrets

Although ConfigMap and Secret work similarly, Kubernetes applies some additional protection for Secret objects.

Secrets often hold values that span a spectrum of importance, many of which can cause escalations within Kubernetes (e.g. service account tokens) and to external systems. Even if an individual app can reason about the power of the Secrets it expects to interact with, other apps within the same namespace can render those assumptions invalid.

A Secret is only sent to a node if a Pod on that node requires it. For mounting secrets into Pods, the kubelet stores a copy of the data into a `tmpfs` so that the confidential data is not written to durable storage. Once the Pod that depends on the Secret is deleted, the kubelet deletes its local copy of the confidential data from the Secret.

There may be several containers in a Pod. By default, containers you define only have access to the default ServiceAccount and its related Secret. You must explicitly define environment variables or map a volume into a container in order to provide access to any other Secret.

There may be Secrets for several Pods on the same node. However, only the Secrets that a Pod requests are potentially visible within its containers. Therefore, one Pod does not have access to the Secrets of another Pod.

> **Warning:** Any containers that run with `privileged: true` on a node can access all Secrets used on that node.

What's next

- For guidelines to manage and improve the security of your Secrets, refer to Good practices for Kubernetes Secrets.
- Learn how to manage Secrets using `kubectl`
- Learn how to manage Secrets using config file
- Learn how to manage Secrets using kustomize
- Read the API reference for `Secret`

Feedback

Was this page helpful?

Yes No

Last modified May 17, 2023 at 9:47 PM PST: Removed unnecessary < and > (b8c826abeb)