Secrets

Kubernetes Secrets let you store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a Pod definition or in a container image. See Secrets design document for more information.

Overview of 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 an image. Users can create secrets and the system also creates some secrets.

To use a secret, a Pod needs to reference the secret. A secret can be used with a Pod in two ways:

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

Built-in Secrets

Service accounts automatically create and attach Secrets with API credentials

Kubernetes automatically creates secrets which contain credentials for accessing the API and automatically modifies your Pods to use this type of secret.

See the ServiceAccount documentation for more information on how service accounts work.

Creating your own Secrets

Creating a Secret Using `kubectl`

Secrets can contain user credentials required by Pods to access a database. For example, a database connection string consists of a username and password. You can store the username in a file `./username.txt` and the password in a file `./password.txt` on your local machine.

```
# Create files needed for the rest of the example.
echo -n 'admin' > ./username.txt
echo -n '1f2d1e2e67df' > ./password.txt
```

The `kubectl create secret` command packages these files into a Secret and creates the object on the API server. The name of a Secret object must be a valid DNS subdomain name.

```
kubectl create secret generic db-user-pass --from-file=./username.txt --from-file=./pass
```

The output is similar to:

```
secret "db-user-pass" 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 --from-
```

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

You can check that the secret was created:

The output is similar to:

```
NAME                  TYPE                                  DATA      AGE
db-user-pass          Opaque                                2         51s
```

You can view a description of the secret:

```
kubectl describe secrets/db-user-pass
```

The output is similar to:

```
Name:            db-user-pass
Namespace:       default
Labels:          <none>
Annotations:     <none>

Type:            Opaque

Data
====
password.txt:    12 bytes
username.txt:    5 bytes
```

Note: The commands `kubectl get` and `kubectl describe` avoid showing the contents of a secret by default. This is to protect the secret from being exposed accidentally to an onlooker, or from being stored in a terminal log.

See decoding a secret to learn how to view the contents of a secret.

Creating a Secret manually

You can also create a Secret in a file first, in JSON or YAML format, and then create that object. The name of a Secret object must be a valid DNS subdomain name. The Secret contains two maps: `data` and `stringData`. The `data` field is used to store arbitrary data, encoded using base64. The `stringData` field is provided for convenience, and allows you to provide secret data as unencoded strings.

For example, to store two strings in a Secret using the `data` field, convert the strings to base64 as follows:

```
echo -n 'admin' | base64
```

The output is similar to:

```
echo -n '1f2d1e2e67df' | base64
```

The output is similar to:

```
MWYyZDFlMmU2N2Rm
```

Write a Secret that looks like this:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Now create the Secret using `kubectl apply`:

```
kubectl apply -f ./secret.yaml
```

The output is similar to:

```
secret "mysecret" created
```

For certain scenarios, you may wish to use the `stringData` field instead. This field allows you to put a non-base64 encoded string directly into the Secret, and the string will be encoded for you when the Secret is created or updated.

A practical example of this might be where you are deploying an application that uses a Secret to store a configuration file, and you want to populate parts of that configuration file during your deployment process.

For example, if your application uses the following configuration file:

```
apiUrl: "https://my.api.com/api/v1"
username: "user"
password: "password"
```

You could store this in a Secret using the following definition:

```
  name: mysecret
type: Opaque
stringData:
  config.yaml: |-
    apiUrl: "https://my.api.com/api/v1"
    username: {{username}}
    password: {{password}}
```

Your deployment tool could then replace the `{{username}}` and `{{password}}` template variables before running `kubectl apply`.

The `stringData` field is a write-only convenience field. It is never output when retrieving Secrets. For example, if you run the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2018-11-15T20:40:59Z
  name: mysecret
  namespace: default
  resourceVersion: "7225"
  uid: c280ad2e-e916-11e8-98f2-025000000001
type: Opaque
data:
  config.yaml: YXBpVXJsOiAiaHR0cHM6Ly9teS5hcGkuY29tL2FwaS92MSIKdXNlcm5hbWU6IHt7dXNlcm5hb
```

If a field, such as `username`, is specified in both `data` and `stringData`, the value from `stringData` is used. For example, the following Secret definition:

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
stringData:
  username: administrator
```

Results in the following Secret:

```
  creationTimestamp: 2018-11-15T20:46:46Z
  name: mysecret
  namespace: default
  resourceVersion: "7579"
  uid: 91460ecb-e917-11e8-98f2-025000000001
type: Opaque
data:
  username: YWRtaW5pc3RyYXRvcg==
```

Where `YWRtaW5pc3RyYXRvcg==` decodes to `administrator`.

The keys of `data` and `stringData` must consist of alphanumeric characters, '-', '_' or '.'.

Note: The serialized JSON and YAML values of secret data are encoded as base64 strings. Newlines are not valid within these strings and must be omitted. When using the `base64` utility on Darwin/macOS, users should avoid using the `-b` option to split long lines. Conversely, Linux users *should* add the option `-w 0` to `base64` commands or the pipeline `base64 | tr -d '\n'` if the `-w` option is not available.

Creating a Secret from a generator

Since Kubernetes v1.14, `kubectl` supports managing objects using Kustomize. Kustomize provides resource Generators to create Secrets and ConfigMaps. The Kustomize generators should be specified in a `kustomization.yaml` file inside a directory. After generating the Secret, you can create the Secret on the API server with `kubectl apply`.

Generating a Secret from files

You can generate a Secret by defining a `secretGenerator` from the files ./username.txt and ./password.txt:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: db-user-pass
  files:
  - username.txt
  - password.txt
EOF
```

Apply the directory, containing the `kustomization.yaml`, to create the Secret.

```
kubectl apply -k .
```

You can check that the secret was created:

```
kubectl get secrets
```

The output is similar to:

```
NAME                             TYPE                                  DATA      AGE
db-user-pass-96mffmfh4k          Opaque                                2         51s
```

```
kubectl describe secrets/db-user-pass-96mffmfh4k
```

The output is similar to:

```
Name:             db-user-pass
Namespace:        default
Labels:           <none>
Annotations:      <none>

Type:             Opaque

Data
====
password.txt:     12 bytes
username.txt:     5 bytes
```

Generating a Secret from string literals

You can create a Secret by defining a `secretGenerator` from literals `username=admin` and `password=secret`:

```
cat <<EOF >./kustomization.yaml
secretGenerator:
- name: db-user-pass
  literals:
  - username=admin
  - password=secret
EOF
```

Apply the directory, containing the `kustomization.yaml`, to create the Secret.

The output is similar to:

```
secret/db-user-pass-dddghtt9b5 created
```

Note: When a Secret is generated, the Secret name is created by hashing the Secret data and appending this value to the name. This ensures that a new Secret is generated each time the data is modified.

Decoding a Secret

Secrets can be retrieved by running `kubectl get secret`. For example, you can view the Secret created in the previous section by running the following command:

```
kubectl get secret mysecret -o yaml
```

The output is similar to:

```
apiVersion: v1
kind: Secret
metadata:
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

Decode the `password` field:

```
echo 'MWYyZDFlMmU2N2Rm' | base64 --decode
```

The output is similar to:

```
1f2d1e2e67df
```

Editing a Secret

An existing Secret may be edited with the following command:

This will open the default configured editor and allow for updating the base64 encoded Secret values in the `data` field:

```
# Please edit the object below. Lines beginning with a '#' will be ignored,
# and an empty file will abort the edit. If an error occurs while saving this file will
# reopened with the relevant failures.
#
apiVersion: v1
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
kind: Secret
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: { ... }
  creationTimestamp: 2016-01-22T18:41:56Z
  name: mysecret
  namespace: default
  resourceVersion: "164619"
  uid: cfee02d6-c137-11e5-8d73-42010af00002
type: Opaque
```

Using Secrets

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.

Using Secrets as files from a Pod

To consume a Secret in a volume in a Pod:

1. Create a secret or use an existing one. Multiple Pods can reference the same secret.
2. Modify your Pod definition to add a volume under `.spec.volumes[]`. Name the volume anything, and have a `.spec.volumes[].secret.secretName` field equal to the name of the Secret object.
3. Add a `.spec.containers[].volumeMounts[]` to each container that needs the secret. Specify `.spec.containers[].volumeMounts[].readOnly = true` and `.spec.containers[].volumeMounts[].mountPath` to an unused directory name where you would like the secrets to appear.

This is an example of a Pod that mounts a Secret in a volume:

```
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
```

Each Secret you want to use needs to be referred to in `.spec.volumes`.

If there are multiple containers in the Pod, then each container needs its own `volumeMounts` block, but only one `.spec.volumes` is needed per Secret.

You can package many files into one secret, or use many secrets, whichever is convenient.

Projection of Secret keys to specific paths

You can also control the paths within the volume where Secret keys are projected. You can use the `.spec.volumes[].secret.items` field to change the target path of each key:

```
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
      readOnly: true
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
```

What will happen:

- `username` secret is stored under `/etc/foo/my-group/my-username` file instead of `/etc/foo/username`.
- `password` secret is not projected.

If `.spec.volumes[].secret.items` is used, only keys specified in `items` are projected. To consume all keys from the secret, all of them must be listed in the `items` field. All listed keys must exist in the corresponding secret. Otherwise, the volume is not created.

Secret files permissions

You can set the file access permission bits for a single Secret key. If you don't specify any permissions, `0644` is used by default. You can also set a default mode for the entire Secret volume and override per key if needed.

For example, you can specify a default mode like this:

```
    name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      defaultMode: 256
```

Then, the secret will be mounted on `/etc/foo` and all the files created by the secret volume mount will have permission `0400`.

Note that the JSON spec doesn't support octal notation, so use the value 256 for 0400 permissions. If you use YAML instead of JSON for the Pod, you can use octal notation to specify permissions in a more natural way.

You can also use mapping, as in the previous example, and specify different permissions for different files like this:

```
apiVersion: v1
kind: Pod
metadata:
  name: mypod
spec:
  containers:
  - name: mypod
    image: redis
    volumeMounts:
    - name: foo
      mountPath: "/etc/foo"
  volumes:
  - name: foo
    secret:
      secretName: mysecret
      items:
      - key: username
        path: my-group/my-username
        mode: 511
```

In this case, the file resulting in `/etc/foo/my-group/my-username` will have permission value of `0777`. Owing to JSON limitations, you must specify the mode in decimal notation.

Note that this permission value might be displayed in decimal notation if you read it later.

base64 decoded and stored inside these files. This is the result of commands executed inside the container from the example above:

```
ls /etc/foo/
```

The output is similar to:

```
username
password
```

```
cat /etc/foo/username
```

The output is similar to:

```
admin
```

```
cat /etc/foo/password
```

The output is similar to:

```
1f2d1e2e67df
```

The program in a container is responsible for reading the secrets from the files.

Mounted Secrets are updated automatically

When a secret currently consumed in a volume is updated, projected keys are eventually updated as well. The kubelet checks whether the mounted secret is fresh on every periodic sync. However, the kubelet uses its local cache for getting the current value of the Secret. The type of the cache is configurable using the `ConfigMapAndSecretChangeDetectionStrategy` field in the KubeletConfiguration struct. A Secret can be either propagated by watch (default), ttl-based, or simply redirecting all requests directly to the API server. 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 (it equals to watch propagation delay, ttl of cache, or zero correspondingly).

FEATURE STATE: `Kubernetes v1.18` alpha

The Kubernetes alpha feature *Immutable Secrets and ConfigMaps* provides an option to set individual Secrets and ConfigMaps as immutable. For clusters that extensively use Secrets (at least tens of thousands of unique Secret to Pod mounts), preventing changes to their data has the following advantages:

- protects you from accidental (or unwanted) updates that could cause applications outages
- improves performance of your cluster by significantly reducing load on kube-apiserver, by closing watches for secrets marked as immutable.

To use this feature, enable the `ImmutableEphemeralVolumes` feature gate and set your Secret or ConfigMap `immutable` field to `true`. For example:

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

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.

Using Secrets as environment variables

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

1. Create a secret or use an existing one. Multiple Pods can reference the same secret.
2. Modify your Pod definition in each container that you wish to consume the value of a secret key to add an environment variable for each secret key you wish to consume. The environment variable that consumes the secret key should populate the secret's name and key in `env[].valueFrom.secretKeyRef`.
3. Modify your image and/or command line so that the program looks for values in the specified environment variables.

This is an example of a Pod that uses secrets from environment variables:

```
  name: secret-env-pod
spec:
  containers:
  - name: mycontainer
    image: redis
    env:
      - name: SECRET_USERNAME
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: username
      - name: SECRET_PASSWORD
        valueFrom:
          secretKeyRef:
            name: mysecret
            key: password
  restartPolicy: Never
```

Consuming Secret Values from environment variables

Inside a container that consumes a secret in an environment variables, the secret keys appear as normal environment variables containing the base64 decoded values of the secret data. This is the result of commands executed inside the container from the example above:

```
echo $SECRET_USERNAME
```

The output is similar to:

```
admin
```

```
echo $SECRET_PASSWORD
```

The output is similar to:

```
1f2d1e2e67df
```

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

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 it 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.

Automatic mounting of manually created Secrets

Manually created secrets (for example, one containing a token for accessing a GitHub account) can be automatically attached to pods based on their service account. See Injecting Information into Pods Using a PodPreset for a detailed explanation of that process.

Details

Restrictions

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.

Secret resources reside in a namespace. Secrets can only be referenced by Pods in that same namespace.

Individual secrets are limited to 1MiB in size. This is to discourage creation of very large secrets which would exhaust the API server and kubelet memory. However, creation of many smaller secrets could also exhaust memory. More comprehensive limits on memory usage due to secrets is a planned feature.

The kubelet only supports the use of secrets for Pods where the secrets are obtained from the API server. This includes any Pods created using `kubectl`, or indirectly via a replication controller. It does not include Pods created as a result of the kubelet `--manifest-url` flag, its `--config` flag, or its REST API (these are not common ways to create Pods.)

Secrets must be created before they are consumed in Pods as environment variables unless they are marked as optional. References to secrets that do not exist will prevent the Pod from starting.

References (`secretKeyRef` field) to keys that do not exist in a named Secret will prevent the Pod from starting.

that were skipped. The example shows a pod which refers to the default/mysecret that 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
```

Secret and Pod lifetime interaction

When a Pod is created by calling the Kubernetes API, there is no check if a referenced secret exists. Once a Pod is scheduled, the kubelet will try to fetch the secret value. If the secret cannot be fetched because it does not exist or because of a temporary lack of connection to the API server, the kubelet will periodically retry. It will report an event about the Pod explaining the reason it is not started yet. Once the secret is fetched, the kubelet will create and mount a volume containing it. None of the Pod's containers will start until all the Pod's volumes are mounted.

Use cases

Use-Case: As container environment variables

Create a secret

```
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  USER_NAME: YWRtaW4=
  PASSWORD: MWYyZDFlMmU2N2Rm
```

Create the Secret:

```
kubectl apply -f mysecret.yaml
```

```
apiVersion: v1
kind: Pod
metadata:
  name: secret-test-pod
spec:
  containers:
    - name: test-container
      image: k8s.gcr.io/busybox
      command: [ "/bin/sh", "-c", "env" ]
      envFrom:
      - secretRef:
          name: mysecret
  restartPolicy: Never
```

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/.ssh/id
```

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. Use a service account which you want to be accessible to all the users with whom you share the Kubernetes cluster, and can revoke this account if the users are compromised.

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

```
  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 --from-lit
```

The output is similar to:

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

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

The output is similar to:

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 --from-
```

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

Now make the Pods:

```
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 .
```

```
/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:

```
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`:

```
  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: k8s.gcr.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.

Clients that use the Secret API

When deploying applications that interact with the Secret API, you should limit access using authorization policies such as RBAC.

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.

For these reasons `watch` and `list` requests for secrets within a namespace are extremely powerful capabilities and should be avoided, since listing secrets allows the clients to inspect the values of all secrets that are in that namespace. The ability to `watch` and `list` all secrets in a cluster should be reserved for only the most privileged, system-level components.

Applications that need to access the Secret API should perform `get` requests on the secrets they need. This lets administrators restrict access to all secrets while white-listing access to individual instances that the app needs.

For improved performance over a looping `get`, clients can design resources that reference a secret then `watch` the resource, re-requesting the secret when the reference changes. Additionally, a "bulk watch" API to let clients `watch` individual resources has also been proposed, and will likely be available in future releases of Kubernetes.

Security properties

Protections

Because secrets can be created independently of the Pods that use them, there is less risk of the secret being exposed during the workflow of creating, viewing, and editing Pods. The system can also take additional precautions with Secrets, such as avoiding writing them to disk where possible.

A secret is only sent to a node if a Pod on that node requires it. The kubelet stores the secret into a `tmpfs` so that the secret is not written to disk storage. Once the Pod that depends on the secret is deleted, the kubelet will delete its local copy of the secret data as well.

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.