Prometheus

Remote Secrets (Secret Providers)

Author: Daniel Hrabovcak
Contributors: N/A
Document state: Review
Created: Oct 26, 2023
Last major update: Nov 3, 2023

Objective

Allow Prometheus to read secrets remotely from secret providers.

Proposal

Introduce secret discovery, similar to service discovery, where different secret providers can contribute code to read secrets from their respective API.

Prior Art

The community highly requests this feature:

- A proposal was drafted to read AWS secrets directly and was supported, but was never finalized for review.
- A backend-only change has been proposed before for using secret paths as a plugin point but was closed because there was no proposal for how the configuration would look like.

There have been many requests for adding new secrets retrieval settings that this proposal would solve:

- GitHub Issue: Encrypt Password
- Prometheus Users Google Group: Encrypt Password
- Stackoverflow: Encrypt Password

Design Overview

Background

Today, Prometheus supports reading secrets from the filesystem and secrets inline the Prometheus configuration.

We will add the possibility to use a secret provider. To use BasicAuth as an example, we currently have the following configurations:

```
Unset
[ password: <secret> ]
[ password_file: <string> ]
```

Motivation

The biggest motivation for this change is that the current configurations require secrets to be inlined or in the filesystem and while most service providers provide sidecars that can do this, this is not always possible.

For service providers that do provide sidecars or when putting a secret on a file is possible, this option is more insecure. For example, to configure Prometheus in Kubernetes, users must create a Kubernetes ConfigMap resource or something similar that they could mount on the Prometheus pod filesystem. If the user wants to use a secret file, they would mount additional Kubernetes Secret resources.

There are two issues with using the filesystem:

1. Occasionally, they add additional setup steps. For example, if the customer is using Kubernetes, they have to do additional work to mount their external secret into the filesystem directly, or indirectly by converting it into a native Kubernetes secret.
2. Secrets being stored in the filesystem is less secure than fetching it and keeping it in memory. For example, in Kubernetes, any pod can access anything that has been mounted on its node, including from other pods. In the case where a Prometheus is federated or distributed across all nodes, an attacker can use any pod to get all secrets used with Prometheus.

Configuration

We will further add:

```
Unset
[ password_ref: <string> ]
```

This points to a secret in another section of the Prometheus configuration.

One such example for Kubernetes might look like:

```
Unset
secret_providers:
- kubernetes_sp_config:
    config:
      api-server: kube-api-url-1
    secrets:
    - name: xyz
      config:
        secret:
          name: secret1
          namespace: ns1
          key: k1
- kubernetes_sp_config:
    config:
      api-server: kube-api-url-2
    secrets:
    - name: abc
      config:
        config_map:
          name: config1
          namespace: ns2
          key: k1
```

Note: This is just an example. The Kubernetes secret provider will be added and finalized in a future proposal.

Here, Prometheus will scan secrets for an item whose name value is equal to `password_ref`. This allows for reuse of secrets in different scrape endpoints. For example, to use the secrets referenced above, this would look like:

```
Unset
basic_auth:
  username: user1
  password_ref: abc
# ...
basic_auth:
  username: user2
  password_ref: xyz
```

Secret names cannot overlap between secret providers. A benefit of this is that the user has the ability to change their secret provider and reuse the same secret name without updating their scrape config.

Detailed Design

Secret providers implement the secret interface:

```
Unset
// Secret represents a sensitive value.
type Secret interface {
	// Fetch fetches the secret content.
	Fetch(ctx context.Context) (string, error)
}
```

This method is called before each scrape. As such, secret providers may fetch data on demand or they may be cached. This detail is left for service providers. For example, a Kubernetes secret provider may use the Kubernetes Watch API and return a cached secret, as opposed to fetching the secret on demand which may otherwise overload the Kubernetes API server if too frequent.

This object is created by the provider interface:

```
Unset
// Provider is a secret provider.
type Provider[T comparable] interface {
	// Add returns the secret fetcher for the given configuration.
	Add(ctx context.Context, config *T) (Secret, error)

	// Update returns the secret fetcher for the new configuration.
	Update(ctx context.Context, configBefore, configAfter *T) (Secret, error)

	// Remove ensures that the secret fetcher for the configuration is invalid.
	Remove(ctx context.Context, config *T) error
}
```

The secret provider receives requests to add, update or remove secrets when the secret or secret provider configurations change. This allows secret providers which use a cache mechanism to keep network connections open.

Config Interface

Prometheus will not discover the secret provider without also implementing the `secrets.Config` interface:

```
Unset
// Config provides the configuration and constructor for a Provider.
type Config[T comparable] interface {
       // Name returns the name of the secret provider.
       Name() string

       // NewProvider creates a new Provider with the options.
       NewProvider(ctx context.Context, opts ProviderOptions) (Provider[T],
error)
}


type SecretProviderOptions struct {
    Logger log.Logger
}
```

The value returned by `Name()` should be short, descriptive, lowercase, and unique. It's used to tag the provided `Logger` and as the part of the YAML key for your secret provider's list of configs in `scrape_config` and `alertmanager_config` (e.g. `${NAME}_sp_config`).

Then register the secret provider configuration:

```
Unset
func init() {
    secrets.RegisterConfig(newKubernetesSecretProviderConfig())
}
```

The secret provider object is created at application start time and whenever the secret provider configuration changes. This allows for a particularly advanced secret provider to pre-load any caches or open any network connections if desired.

Metrics

The following metrics associated with secret providers will be available:

1. Current number of secret provider configurations that failed to load.
2. Current number of secret configurations that failed to load.
3. Current number of secret providers.
4. Current number of secrets.

Alternatives considered

Inline Secrets

Unlike secret references, we could have inlined secrets. However, to keep configurations small we decided to use a mapping.

An example where a user might want to use a mapping is when they use a service mesh such as Istio. Here they can specify a single secret to use for all endpoints.

Service Config Inline

It may seem more natural for users to create secret levels top-level, since names cannot overlap. Then, inline the services:

```
Unset
secrets:
- name: xyz
  kubernetes_sp_config:
    secret:
      name: secret1
      namespace: ns1
      key: k1
    api-server: kube-api-url-1
- name: abc
  kubernetes_sp_config:
    config_map:
      name: config1
      namespace: ns2
      key: k1
    api-server: kube-api-url-2
```

Pros:

- Easier for humans to scan which secrets they have.

Cons:

- Potentially lots of repeated data.

Service Config Mappings

It may seem more natural for users to create secret levels top-level, since names cannot overlap. If you inline service configurations, then some would be repeated. An alternative is to use a map:

```
Unset
secrets:
- name: xyz
  kubernetes_sp_config:
    secret:
      name: secret1
      namespace: ns1
      key: k1
  config_name: kube1
- name: abc
  kubernetes_sp_config:
    config_map:
      name: config1
      namespace: ns2
      key: k1
  config_name: kube2
```

Then the actual service configurations live on the outside:

```
Unset
secret_providers:
- name: kube1
  kubernetes_service_config:
    api-server: kube-api-url-1
- name: kube2
  kubernetes_service_config:
    api-server: kube-api-url-2
```

Pros:

- Easier for humans to scan which secrets they have.
- Data structures are easier to represent in code.

Cons:

- 3 layers of indirection: scrape endpoint→secret→secret providers.

Deprecate Existing Options

With this change, we can potentially deprecate all other forms of configurations and exclusively encourage secret providers. To use BasicAuth as an example, we currently have the following:

```
Unset
[ password: <secret> ]
[ password_file: <string> ]
```

These could be replaced with an `inline_sp_config` and `filesystem_sp_config` respectively.

To keep this proposal as small as possible, these types of details were omitted for a future proposal.

Filesystem Encryption

One common request from users is the ability to read an encrypted secret from the filesystem. This would be possible with a new service provider `filesystem_sp_config` with encryption fields.

To keep this proposal as small as possible, these types of details were omitted for a future proposal. Alternatively, users can simply register their own service provider.

Federation

One consideration when using secret providers is cost. Some secret providers charge a fee based on each secret access. As such, it is desirable to make as few secret accesses as possible. This becomes an issue when using federation, where multiple Prometheus instances are accessing the same secrets independently.

One solution for this is to allow child Prometheus instances to reach a centralized parent Prometheus which caches secrets, or a binary which only caches secrets.

Due to the increased complexity of this feature, it was omitted from this proposal but may be a future proposal. One possible solution may involve a special secret provider which handles this.