KIP-297: Externalizing Secrets for Connect Configurations

Status

Current state: *Accepted*

Discussion thread: *here*

JIRA: ⚠ Unable to render Jira issues macro, execution error.

Released: 2.0.0

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Kafka Connect allows integration with many types of external systems. Some of these systems may require secrets to be configured in order to access them. Many customers have an existing Secret Management strategy and are using centralized management systems such as Vault, Keywhiz, or AWS Secrets Manager. These Secret Management systems may satisfy the following customer requirements:

- No secret in cleartext at rest (such as on disk) or in transit (over the network)
- Secrets protected by access control mechanisms
- All access to secrets recorded in an audit log
- Support for secret versioning and rolling
- A common set of APIs for both applications and tools to access secrets
- Redundancy in case of failover so that secrets are always available
- Certification as conformant with required compliance standards

Other customers may be passing secrets into the host through various means (such as through Docker secrets), but do not want the secret to appear in cleartext in the Kafka Connect configuration.

Connect's connector configurations have plaintext passwords, and Connect stores these in cleartext either on the filesystem (for standalone mode) or in internal topics (for distributed mode). Connect's REST API also exposes these passwords when unsecured connections are used.

Connect should not store or transmit cleartext passwords in connector configurations. TLS can be enabled on Connect's REST API, and this proposal addresses how Connect deals with secrets in stored connector configurations by integrating with external secret management systems. First, since no single standard exists, Connect will provide an extension point for adding customized integrations and will provide a simple file-based extension as an example. Second, a Connect runtime can be configured to use one or more of these extensions, and will allow connector configurations to use placeholders that will be resolved by the runtime before passing the complete connector configurations to connectors. Therefore, existing connectors will not see any difference in the configurations that Connect provides to them at startup. And third, Connect's API will be changed to allow a connector to obtain the latest connector configuration at any time.

Relationship to Kafka Brokers and Clients

While this KIP focuses on Kafka Connect, we propose some common public interfaces and classes that could be used by other parts of Kafka, specifically:

- `ConfigProvider`, `ConfigChangeCallback, ConfigData`: These interfaces could potentially be used by the Broker in conjunction with KIP-226 to overlay configuration properties from a `ConfigProvider` (such as a `VaultConfigProvider`) onto existing configuration properties.
- `ConfigTransformer`: This class could potentially be used by other clients (in conjunction with the previous interfaces) to implement variants of KIP-76 and KIP-269.

Public Interfaces

The public interfaces that are not Connect-specific consist of the following:

- `ConfigProvider`, `ConfigChangeCallback, ConfigData`: These interfaces are used to abstract a provider of configuration properties.
- `ConfigTransformer`: This class is used to provide variable substitution for a configuration value, by looking up variables (or indirect references) from a set of `ConfigProvider` instances. It only provides one level of indirection.

The above classes will be in the package `org.apache.kafka.common.config`, except for `ConfigProvider`, which will be in the package `org.apache.kafka.common.config.provider`, along with any implementations of `ConfigProvider` (which is currently only `FileConfigProvider`).

```
public interface ConfigProvider extends Configurable, Closeable {

    // Configure this class with the initialization parameters
    void configure(Map<String, ?> configs);

    // Look up the data at the given path.
    ConfigData get(String path);

    // Look up the data with the given keys at the given path.
    ConfigData get(String path, Set<String> keys);

    // The ConfigProvider is responsible for making this callback whenever the key changes.
    // Some ConfigProviders may want to have a background thread with a configurable update interval.
    void subscribe(String path, Set<String> keys, ConfigChangeCallback callback);

    // Inverse of subscribe
    void unsubscribe(String path, Set<String> key, ConfigChangeCallback callback);

    // Remove all subscriptions
    void unsubscribeAll();

    // Close all subscriptions and clean up all resources
    void close();
}

public class ConfigData {

    private Long ttl;
    private Map<String, String> data;

    public ConfigData(Map<String, String> data, Long ttl) {
        this.ttl = ttl;
        this.data = data;
    }

    public ConfigData(Map<String, String> data) {
        this(null, data);
    }

    public Long ttl() {
        return ttl;
    }

    public Map<String, String> data() {
        return data;
    }
}
public interface ConfigChangeCallback {

    void onChange(String path, ConfigData data);
}
```

Also a helper class will be added that can provide variable substitutions using `ConfigProvider` instances. Here is an example skeleton.

```
/**
 * This class wraps a set of {@link ConfigProvider} instances and uses them to perform
 * transformations.
 */
public class ConfigTransformer {
    private static final Pattern DEFAULT_PATTERN = Pattern.compile("\\$\\{(.*?):((.*?):)?(.*?)\\}");

    private final Map<String, ConfigProvider> configProviders;
    private final Pattern pattern;

    public ConfigTransformer(Map<String, ConfigProvider> configProviders) {
        this(configProviders, DEFAULT_PATTERN);
    }

    public ConfigTransformer(Map<String, ConfigProvider> configProviders, Pattern pattern) {
        this.configProviders = configProviders;
        this.pattern = pattern;
    }

    public Map<String, String> transform(Map<String, String> configs) {
        ...
    }
}
```

An implementation of `ConfigProvider` called `FileConfigProvider` will be provided that can use secrets from a Properties file. When using the `FileConfigProvider` with the variable syntax `${file:path:key}`, the `path` will be the path to the file and the `key` will be the property key.

Implementations of `ConfigProvider`, such as `FileConfigProvider`, that are provided with Apache Kafka will be placed in the package `org.apache.kafka.common.config.provider`. This will facilitate frameworks such as Connect that treat instances of `ConfigProvider` as components that should be loaded in isolation. Connect will assume that all classes in the package `org.apache.kafka.common.config.provider` (except `ConfigProvider`) are such components.

Two existing interfaces that are specific to Connect will be modified. This will allow for Tasks to get the latest versions of their configs with all indirect references reloaded.

```
public interface SinkTaskContext {
    ...
    Map<String, String> configs();
    ...
}

public interface SourceTaskContext {
    ...
    Map<String, String> configs();
    ...
}
```

The following configuration properties for Connect will be added.

Config Option	**Description**	**Example**	**Default**
config.providers	A comma-separated list of names for providers.	config.providers=file,vault	N/A
config.providers.{name}.class	The Java class name for a provider.	config.providers.file.class=org.apache.kafka.connect.configs.FileConfigProvider	N/A
config.providers.{name}.param.{param-name}	A parameter to be passed to the above Java class on initialization.	config.providers.file.param.secrets=/run/mysecrets	N/A
config.reload.action	One of: • "none" - no action when onChange() is called • "restart" - schedule a restart when onChange() is called	config.reload.action=restart	restart

Proposed Changes

Currently the configuration for both Connectors and Tasks is stored in a Kafka topic. The goal is for these stored configurations to only contain indirect references to secrets. When a Connector or Task is started, the configuration will be read from Kafka and then passed to the specific Connector or Task. Before the configuration is passed to the Connector or Task, the indirect references need to be resolved.

The following are required in a design:

- Ability to specify one or more custom ConfigProviders that will resolve indirect references for configuration values. The ConfigProviders are plugins and use the same classloading mechanism as other plugins (converters, transformers, etc.).
- Ability to pass data to initialize a ConfigProvider on construction or instantiation.
- For indirect references, a special syntax using the dollar sign ($) will be used to indicate when a configuration value is an indirect reference and for which ConfigProvider(s).

The patterns for variable substitutions are of the form `${provider:[path:]key}`, where only one level of indirection is followed during substitutions. The `path` in the variable is optional. This means if you have the following:

```
foo=${file:bar}
bar=${file:baz}
```

and your file contains

```
bar=hello
baz=world
```

then the result will be

```
foo=hello
bar=world
```

As a further clarification, if the ConfigProvider provides a value of the form `${xxx:yyy}`, no further interpolation is done to try to find a provider for `xxx`, for example. Also, if a provider does not have a value for the corresponding key, the variable will remained unresolved and the final value will still be of the form `${provider:key}`.

Here is an example use case:

```
# Properties specified in the Worker config
config.providers=vault   # can have multiple comma-separated values
config.providers.vault.class=com.org.apache.connect.configs.VaultConfigProvider
config.providers.vault.param.uri=1.2.3.4
config.providers.vault.param.token=/run/secrets/vault-token

# Properties specified in the Connector config
mysql.db.password=${vault:vault_path:vault_db_password_key}
```

In the above example, VaultConfigProvider will be passed the string "/run/secrets/vault-token" on initialization, which could be the filename for a Docker sec ret containing the initial Vault token, residing on the tmpfs mount, for instance.   When resolving the value for "mysql.db.password", the VaultConfigProvider will use the path "vault_path" and the key "vault_db_password_key". The VaultConfigProvider would use this path and key to look up the corresponding secret. (VaultConfigProvider is a hypothetical example for illustration purposes only.)

Here is an example of a FileConfigProvider:

```
/**
 * An implementation of {@link ConfigProvider} that represents a Properties file.
 * All property keys and values are stored as cleartext.
 */
public class FileConfigProvider implements ConfigProvider {

    public void configure(Map<String, ?> configs) {
    }

    /**
     * Retrieves the data at the given Properties file.
     *
     * @param path the file where the data resides
     * @return the configuration data
     */
    public ConfigData get(String path) {
        Map<String, String> data = new HashMap<>();
        if (path == null || path.isEmpty()) {
            return new ConfigData(data);
        }
        try (Reader reader = reader(path)) {
            Properties properties = new Properties();
            properties.load(reader);
            Enumeration<Object> keys = properties.keys();
            while (keys.hasMoreElements()) {
                String key = keys.nextElement().toString();
                String value = properties.getProperty(key);
                if (value != null) {
                    data.put(key, value);
                }
            }
            return new ConfigData(data);
        } catch (IOException e) {
            throw new ConfigException("Could not read properties from file " + path);
        }
    }

    /**
     * Retrieves the data with the given keys at the given Properties file.
     *
     * @param path the file where the data resides
     * @param keys the keys whose values will be retrieved
     * @return the configuration data
     */
    public ConfigData get(String path, Set<String> keys) {
        Map<String, String> data = new HashMap<>();
        if (path == null || path.isEmpty()) {
            return new ConfigData(data);
        }
        try (Reader reader = reader(path)) {
            Properties properties = new Properties();
            properties.load(reader);
            for (String key : keys) {
                String value = properties.getProperty(key);
                if (value != null) {
                    data.put(key, value);
                }
            }
            return new ConfigData(data);
        } catch (IOException e) {
            throw new ConfigException("Could not read properties from file " + path);
        }
    }

    // visible for testing
    protected Reader reader(String path) throws IOException {
        return new BufferedReader(new InputStreamReader(new FileInputStream(path), StandardCharsets.UTF_8));
    }

    public void close() {
    }
}
```

Usage:

```
config.providers=file
config.providers.file.class=org.apache.kafka.connect.configs.FileConfigProvider
```

Secret Rotation

Secret Management systems such as Vault support secret rotation by associating a "lease duration" with a secret, which can be read by the client.

In general, secret rotation is orthogonal to a particular Connector. For example, a JDBC password may be stored in a Docker secret or in Vault. The JDBC connector does not need to know what the method of rotation is. Also, it is best if the JDBC connector is informed when it should re-obtain secrets rather than wait until a security exception occurs. So in this case, a push model is warranted.

Other connectors such as the S3 connector are tightly coupled with a particular secret manager, and may wish to handle rotation on their own.

To handle the different scenarios, the design offers support both a **push** model and a **pull** model for obtaining new secrets.

Different Connect components may have different responsibilities in handling secret rotation:

- **ConfigProvider**: The `ConfigProvider` may have knowledge of the method of rotation. For Vault, it would be a "lease duration". For a file-based provider, it could be file watches. If it knows when a secret is going to be reloaded, it would call `onChange()` to inform the Herder.
- **Herder:** The Herder can **push** information to the Connector indicating that secrets have expired or may expire in the future. When the Herder receives the `onChange()` call, it will check a new connector configuration property `config.reload.action` which can be one of the following:
    1. The value `restart`, which means to schedule a restart of the Connector and all its Tasks. This will be the default.
    2. The value `none`, which means to do nothing.
- **Connector Tasks:** A task may wish to handle rotation on its own (a **pull** model). In this case the Connector would need to set `config.reload.action` to `none`. The methods `SinkTaskContext.config()` and `SourceTaskContext.config()`would be used by the Task to reload the config and resolve indirect references again.

Compatibility, Deprecation, and Migration Plan

No changes are required for existing Connectors. Existing connector configurations with plaintext passwords will not be affected, and only after they are changed to use the variables (aka, indirect references) will the secrets not be stored by Connect.

Connectors that use a ConfigProvider and do not want the restart behavior can specify `config.reload.action` as `none`.

Assumptions and Limitations

A typical pattern in Connector implementations is for the Task configuration to be a superset of the Connector configuration.   That is, when creating a Task configuration, the Connector will copy the Connector configuration and then perhaps add some additional configuration properties. However, there is nothing in the Connect framework that enforces that this relationship need hold between the Task configuration and the Connector configuration.

For the implementation of this KIP, we will make the assumption that a variable reference in the property of the Task configuration has the same key as the variable reference in the Connector configuration. This is because when the Connector configuration is passed to the Connector, all variable references have been resolved, so that when the Connector returns the Task configuration to the Connect framework, the Connect framework can make use of this assumption to "reverse transform" the resolved value back to the original variable reference before saving the Task configuration.

This implies that when the assumption does not hold, which would be the case if the Connector copies the resolved value to a property with a *different* key, for example, then the Connect framework will not be able to reverse transform the value to the original variable reference, and the value will appear in plain-text when saved.   This is a limitation that is currently not addressed by this KIP.

Rejected Alternatives

The current scope of this proposal is for Connectors only. However, as mentioned above, it provides classes that can be used by Brokers and Clients for functionality in other KIPs.

A related feature for brokers is KIP-226, which allows for dynamic broker configuration. It can also store passwords. However,

1. It currently does not work for Kafka Connect.
2. One requirement is to not "leak" secrets to other systems, especially if the customer is already using a centralized Secret Management system.

A related feature for clients is KIP-76, which is for obtaining passwords through scripts. However,

1. It is not yet implemented.
2. It only applies to certain password fields.
3. It does not allow for custom plugins.

Another related feature for clients is KIP-269, which is for using variables for JAAS configuration. However,

1. It is not yet implemented.
2. It only applies to JAAS configuration.

Again, as mentioned above, the classes above can be used to implement or augment the behavior of these KIPs. However, that effort is a separate effort from this KIP.