Design

Kerberized Worker Harness

Radek Stankiewicz (radoslaws@google.com),

Overview

The "Kerberized worker" feature aims to simplify Kerberos authentication for Apache Beam, starting with KafkaIO. Currently, configuring Kerberos requires manual file uploads (keytab, krb5.conf), setting JVM system variables in case of Java IO, and potentially building custom containers or expansion services. This process is cumbersome and adds significant friction for developers. The goal is to make Kerberos configuration as straightforward as using GCP application default credentials, providing a foundation for broader Kerberos support across various Beam IOs.

Goals

- Expose Kerberos options
- Configure worker's harness container so all other modules can benefit. Put files in place, setup environment variables and JVM properties.
- Implement KafkaIO and KafkaIO Schema Provider changes in Java SDK based on Kerberos Options
- Implement Python wrapper changes

- Implement KerberosOptions' files staging in various runners.

Non-goals

- Support for all runners
- Implement changes to other IOs

Background

In order to set up a client connection (with Kafka or other service) with Kerberos authentication, we need to provide several pieces of information - principal, krb5.conf file, keytab and optionally service name.

Configuration pieces

Principal

A principal is a unique identity that can be assigned tickets to access Kerberos-aware services. Think of it like your username in a secure network, but with some extra details.

krb5.conf

krb5.conf is a crucial configuration file for Kerberos clients. It provides the client with all the essential information it needs to interact with Kerberos infrastructure and authenticate itself to services. It includes information about Key Distribution Center (KDC) - KDC is the heart of Kerberos, responsible for issuing tickets that grant access to services. krb5.conf specifies the Kerberos realms the client should operate in. A realm is like a security domain within Kerberos. It also contains defaults, like encryption types. Lastly, krb5.conf can map hostnames to specific Kerberos realms.

Krb5.conf has default location in /etc/krb5.conf for linux operating systems or overridden with KRB5_CONFIG.

It is possible to authenticate kerberos without the krb5.conf file if we provide KDC but it may not be enough in more complex environments with multiple realms.

Keytab

A file that stores Kerberos principal and its encrypted key. This key is derived from password but is never actually stored in plain text. Still, ensure that only the necessary services and users have access to the keytab.

Kerberos service name

Service name is a configuration parameter that specifies the name of the service to which a client is attempting to authenticate. In the context of KafkaIO it's typically kafka, for HBaseIO it's usually hbase.

Libraries

There are various libraries that provide higher level framework for kerberos authentication - here are the most popular ones for languages in scope of the change:
Java Authentication and Authorization Service (JAAS) - Java extension supporting multiple authentication schemes, including OAuth, Kerberos and 2-way ssl.
Python kerberos package - Python wrapper around kerberos.

Problem statement

As a python or java developer, in order to configure kerberos in kafkaIO I need to:

- Upload files (keytab, krb5.conf, keystore, truststore) to worker, set the system variable within JVM via JVMInitializer.
- Alternatively build a new container and add those files. Set JVM system variables via JVMInitializer
- When using Python SDK I need to inject Jvminitializer into the classpath of the expansion service or build and maintain my own expansion service.
- Configure KafkaIO with Kafka config

All the steps are documented in Java SDK and Python SDK notes tabs.

Part 1 - Staging files on harness

There are several files that have to be placed on each harness in order to setup Kerberos or SSL properly.

Java SDK provides FileStagingOption that helps staging any file and placing it on CWD. The default value is the list of jars from the main program's classpath. As the main purpose of it is passing jars, boot.go in the container is adding all those files as java classpath which doesn't make sense when uploading a truststore or a keytab.

There is no such option exposed in the Python SDK but pipeline runner is leveraging Stager.py to prepare a list of all artifacts to stage.

In both cases, there is a need to prepare new options (e.g. KerberosOptions) that will provide user ability to set krb5.conf, keytab paths and those will be added automatically to artifacts to be staged.

Secondly, Python SDK would expose a new flag (mentioned here) that would allow staging any file to CWD e.g. keystore and trustore.

Part 2 - Initializing worker harness

There are several files and configuration items that have to be provided and set at OS level in order to Kerberos function properly:

- Keytab has to be placed in the filesystem, preferably in CWD
- Krb5.conf has to be placed in CWD and KRB5_CONFIG to be set in case of Python SDK

- For JVM container java should have additional flag java.security.krb5.conf set as Java is not respecting KRB5_CONFIG

Option 1 - Initializing worker harness during boot time

Scope of this design is to add ability in worker's boot.go to set environment variables and add ability to pass JVM variables.

Option 2 (preferred) - Initializing worker harness during SDK init

Create kerberos specific https://s.apache.org/python_sdk_worker_initialization and JVM Initializer that will be configuring KRB5_CONFIG in Python SDK and java.security.krb5.conf in Java SDK. This should work in both regular worker SDK and expansion service.

Part 3 - Simplify Kerberos in KafkaIO

Once all the files are on SDK, developer can setup KafkaIO with following properties in config updates:

```
None
'security.protocol' : 'SASL_PLAINTEXT',
'sasl.mechanism' : 'GSSAPI',
'sasl.jaas.config' : 'com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true
storeKey=true keyTab="<KEYTAB PATH>" principal="<principal>@<REALM>";'
```

As part of this design we should add following KerberosOptions - principal, serviceName and use them in helper functions in JavaSDK:

```
Java
  public Read<K, V> withKerberosCredentials() {
      return withConsumerConfigUpdates(
          ImmutableMap.of(
              CommonClientConfigs.SECURITY_PROTOCOL_CONFIG,
              "SASL_PLAINTEXT",
              SaslConfigs.SASL_MECHANISM,
              "GSSAPI",
              SaslConfigs.SASL_JAAS_CONFIG,
              "com.sun.security.auth.module.Krb5LoginModule required useKeyTab=true
storeKey=true keyTab=\"" + options.getKeyTab() + "\" principal=\"" + options.getPrincipal() +
"\";"));
    }
```

To support external transform, a boolean flag could be exposed to add kerberos credentials using the above helper method.

Appendix

PKI - SSL

Depending on the service, a custom PKI is created which requires providing a truststore along with passwords for 1-way authentication and a keystore with password for 2-way authentication.

Future Work

List out additional pieces of the solution which may be useful but were not included in this design.

- Implement Kerberos support to HBaseIO, HDFSIO
- Implement Kerberos in other runners - Spark and Flink along with integration tests
- Implement Kerberos support in Yaml
- Implement changes to Managed KafkaIO
- Implement 2-way authn helper in KafkaIO

Java SDK - notes

Java SDK

Here are snippets of code on how to configure kerberos so other components can benefit.

KerberosOptions - whole class is needed to encapsulate all Kerberos config. Feel free to use it or copy relevant fields to your options.

```
Java
package com.google.dce.options;
import org.apache.beam.sdk.options.Description;
import org.apache.beam.sdk.options.PipelineOptions;
@SuppressWarnings("unused")
public interface KerberosOptions extends PipelineOptions {
  @Description(
      "Kerberos principal name to authenticate as with kafka, leave unassigned if kerberos"
          + " authentication is not needed")
  String getKerberosPrincipalName();
  void setKerberosPrincipalName(String value);
KerberosKrb5
String getKerberosKrb5();
  void setKerberosKrb5(String value);
  String getKeytabName();
  void setKeytabName(String value);
  @Description("Key Distribution Center address")
}
```

JVMInitializer - this class will be invoked on every worker.
This JVM initializer also adds a consumer factory for Kafka.

```
Java
package com.google.dce.kafka;
import com.google.auto.service.AutoService;
import com.google.common.base.MoreObjects;
import com.google.dce.options.KerberosOptions;
import com.google.dce.secrets.SecretManagerHelper;
import com.google.protobuf.ByteString;
import java.io.BufferedOutputStream;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
```

```
import java.util.HashMap;
import java.util.Map;
import java.util.Objects;
import java.util.stream.Stream;
import org.apache.beam.sdk.extensions.gcp.options.GcpOptions;
import org.apache.beam.sdk.harness.JvmInitializer;
import org.apache.beam.sdk.options.PipelineOptions;
import org.apache.beam.sdk.transforms.SerializableFunction;
import org.apache.kafka.clients.consumer.Consumer;
import org.apache.kafka.clients.consumer.KafkaConsumer;
import org.checkerframework.checker.nullness.qual.NonNull;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
@AutoService(JvmInitializer.class)
public class KerberosJvmInitializer implements JvmInitializer {
  private static final Logger LOG = LoggerFactory.getLogger(KerberosJvmInitializer.class);
  private static volatile Path keytabFile = null;


  public static SerializableFunction<Map<String, Object>, Consumer<byte[], byte[]>>
      getConsumerFactory(String kerberosPrincipal) {
    return configs -> {
      configs = MoreObjects.firstNonNull(configs, new HashMap<>());
      if (keytabFile != null) {
        LOG.info(
            "configuring kerberos authentication for kafka consumer with the downloaded
keytab");
        configs.put("security.protocol", "SASL_PLAINTEXT");
        configs.put("sasl.kerberos.service.name", "kafka");
        configs.put(
            "sasl.jaas.config",
            String.format(
                "com.sun.security.auth.module.Krb5LoginModule required"
                    + "  useKeyTab=true"
                    + "  storeKey=true"
                    + "  refreshKrb5Config=true"
                    + "  keyTab=\"%s\""
                    + "  principal=\"%s\";",
                keytabFile.toAbsolutePath(), kerberosPrincipal));
      } else {
        LOG.warn(
            "requested kafka consumer factory supporting downloaded keytab, but the keytab
wasn't"
                + " downloaded");
      }
      return new KafkaConsumer<>(configs);
    };
```

```
  }
  private static ByteString downloadKeytab(String keytab) {
    return KerberosUtil.download(keytab);
  }
  public static void initializeKerberos( String keytab, String config, String configName,
String keytabName) {
    System.setProperty("java.security.krb5.conf", configName);
    KerberosUtil.downloadTo(config, configName);
    try {
      keytabFile = Files.createTempFile(keytabName, null);
    } catch (IOException e) {
      throw new RuntimeException("unable to create a temporary file to store the keytab", e);
    }
    ByteString keytab = downloadKeytab(secretManagerHelper, keytabSecretName);
    try (BufferedOutputStream outputStream =
        new BufferedOutputStream(Files.newOutputStream(keytabFile))) {
      keytab.writeTo(outputStream);
    } catch (IOException e) {
      throw new RuntimeException("unable to write the keytab to a temp file", e);
    }
  }
  @Override
  public void beforeProcessing(@NonNull PipelineOptions options) {
    KerberosOptions kerberosOptions = options.as(KerberosOptions.class);
    GcpOptions gcpOptions = options.as(GcpOptions.class);
    String project = gcpOptions.getProject();
    String name = kerberosOptions.getKeytabName();
    String config = kerberosOptions.getKerberosKrb5();

    initializeKerberos(name, config, KerberosUtil.extractFileName(name));
  }
}
```

Python SDK - notes

Python SDK

Here are snippets of code how to pass kerberos configuration to expansion service.

Options with matching names

Remember that dest field names should be camel case with lowercase first letter and should match Java SDK names in KerberosOptions

Python

```
class KerberosOptions(PipelineOptions):
    """DirectRunner-specific execution options."""
    @classmethod
    def _add_argparse_args(cls, parser):
        parser.add_argument(
            '--KerberosPrincipalName',
            dest='kerberosPrincipalName',
            help='princ.')
        parser.add_argument(
            '--KerberosKrb5',
            type=str,
            dest='kerberosKrb5',
            help='kdc')
        parser.add_argument(
            '--KeytabPath',
            type=str,
            dest='keytabPath',
            help='path.')
```

Pipeline

Place KafkaKerberosInitializer artifact next to your pipeline code.
Patch classpath for expansion service and add custom jar.

Python

```
# consumer_config_updates - build sasl.jaas.config from
pipeline_options.KerberosPrincipalName etc
  producer_config = {
      'bootstrap.servers': bootstrap_servers,
      'security.protocol' : 'SASL_PLAINTEXT',
```

```
      'sasl.mechanism' : 'GSSAPI',
      'sasl.kerberos.service.name': 'kafka',
      'sasl.jaas.config' : 'com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true storeKey=true keyTab="/tmp/dataflow.keytab"
principal="dataflow@US-CENTRAL1-F.C.RADOSLAWS-PLAYGROUND-PSO.INTERNAL";'
  }


  rTransform = ReadFromKafka(producer_config=producer_config, topic=topic)
  rTransform._expansion_service._classpath = ["KafkaKerberosInitializer-1.0-SNAPSHOT.jar"]


# pcol of Row
pcol = (
        pipeline
             | readTransform
             | beam.FlatMap(lambda l: log(l))
        )
```

Or managed:

```
Python
config_updates = {
            'security.protocol' : 'SASL_PLAINTEXT',
            'sasl.mechanism' : 'GSSAPI',
            'sasl.kerberos.service.name': 'kafka',
            'sasl.jaas.config' : 'com.sun.security.auth.module.Krb5LoginModule required
useKeyTab=true storeKey=true keyTab="/tmp/dataflow.keytab"
principal="dataflow@US-CENTRAL1-F.C.RADOSLAWS-PLAYGROUND-PSO.INTERNAL";'
    }
  kafka_write_config = {
      'topic': topic,
      'bootstrap_servers': bootstrap_servers,
      'data_format': "RAW",
      'producer_config_updates': config_updates
  }

  kafka_read_config = {
      'topic': topic,
      'bootstrap_servers': bootstrap_servers,
      'data_format': "RAW",
      'consumer_config_updates': config_updates
```

```
  }
  readTransformManaged = beam.transforms.managed.Read(beam.transforms.managed.KAFKA,
                                                      config=kafka_read_config)
  readTransformManaged._expansion_service._classpath =
["KafkaKerberosInitializer-1.0-SNAPSHOT.jar"]
  writeTransformManaged = beam.transforms.managed.Write(beam.transforms.managed.KAFKA,
                                                      config=kafka_write_config)
  writeTransformManaged._expansion_service._classpath =
["KafkaKerberosInitializer-1.0-SNAPSHOT.jar"]

ride_col = (
        pipeline
         | readTransformManaged
         | writeTransformManaged
        )

    _ = ride_col | beam.FlatMap(lambda l: log(l))
```