How Spark on Kubernetes will access Secure HDFS

User Flow

Short-running jobs are jobs that last shorter than 24 hours. As such, they don't require the refresh of its **Delegation Token (DT)**, which lasts for 24 hours unless renewed. These jobs do not require helps from the renew service pod described below.

Long-running jobs are jobs that run for longer than 24 hours and as such will require renewed and reissued **DT**s. For jobs that last between 1 day and 7 days, you would just renew the original **TGT** as it can be renewed until 7 days. However, for long-running jobs that last beyond 7 days, you would obtain a newly issued **DT**, which would in turn require a new **TGT (Ticket Granting Ticket)**, and new sign on with Kerberos.

A special service pod would both renew existing DTs and obtain new DTs to extend the lifetime of long-running jobs. The service pod will use its own Kerberos principal. For the service principal to renew existing DTs, the DTs would specify the service principal as renewer. For the service principal to obtain newly issued DTs on behalf of users, the principal should be configured as proxy principle in the HDFS namenode config. The renew service pod will be using a Kerberos **keytab** file containing the credential of the service principal.

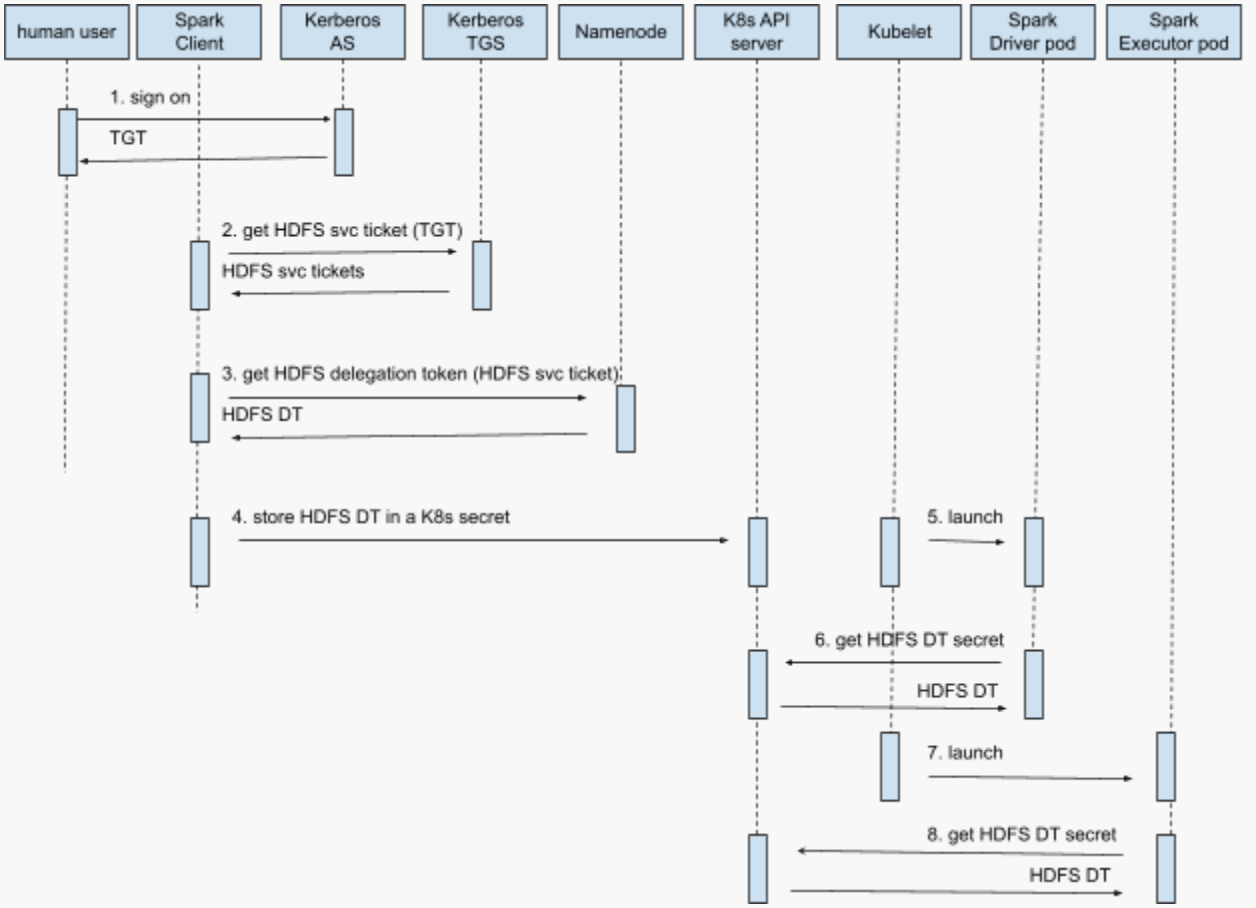
General Overview

This doc investigates how we propose that Spark on Kubernetes should work under Kerberos, in particular how it interacts with HDFS. The design is based on YARN logic, described in How Spark on Yarn accesses Kerberized HDFS, while leveraging the primitives that Kubernetes offers. In particular, we will be leveraging k8s secrets to share HDFS DTs among the job client, driver and executor pods.

Design

Basic Interaction

The followings describe the basic interaction between job client, the driver and executor pods that take place to access Kerberized HDFS. When the given job lasts less than 1 day, this basic interaction is all that happens.



The job user begins with a $kinit shell command, where s/he will execute $kinit, before running spark-submit. With $kinit the client will manually sign into Kerberos on the client host. The $kinit will obtain the *Job User's* **TGT** from the **Kerberos AS (Authentication Server)**. **TGT** is the Kerberos master ticket for single sign-in. Kinit stores it under a local disk directory that caches Kerberos tickets: **local ticket cache (LTC)**.

Now in the spark-submit, with the **TGT** located in the **LTC**, aftering calling **JAAS** methods like obtainCredentials(), you can obtain a HDFS **DT**. With a Client Object that contains both a **TGT** and **DT**, the next step is to have the submission client create a secret containing the **DT**. The **DT** secret creation step will thusly be added to the SubmissionStepOrchestrator. This phase is particularly important from a security perspective because we must ensure that the mounting of the **DT** secret is done securely and that secret has read/write access only by the client / service groups who resides in that given namespace. We stress the importance of the authorization provider to leverage **RBAC** Authorization to give read / write access to secrets, within a namespace, via Role Bindings to ensure that unwanted user has access to reading or writing secrets in namespaces aside from the one allocated to their jobs.

With the secrets containing the **DT**. Upon the launching of the driver at the end of Client.run(), the Driver pod will read the **DT** from the secret. The driver at this point has a **UGI** that has the

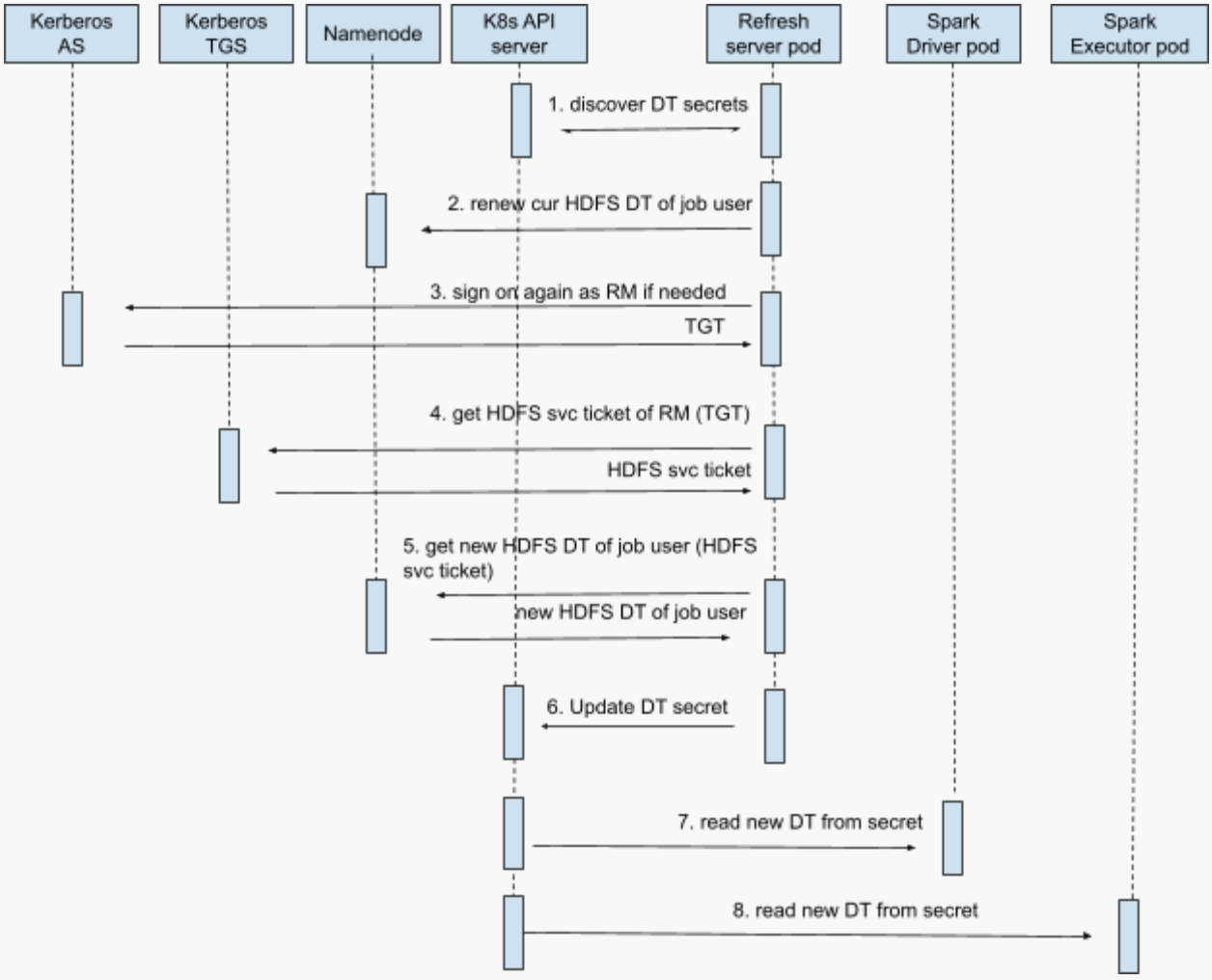
DT. Following, when the executor's are spawned, they will be mounted with the same **DT** from the same secret. Note the driver and executor pods do not need to enable Kerberos in the Hadoop configuration, since having the DT is sufficient for accessing secure HDFS.

At this point in time, the strategy is to leverage the SparkHadoopUtils.runAsSparkUser() method which will be initialized with a dummy UGI variable (to avoid triggering a Kerberos sign on process), that will be updated upon the utilization of the mounted **DT** to transfer credentials to the Job User. At this point, the executors all have a valid **DT** to create a valid **UGI** for secure HDFS interaction. (Note that if the job is short-living, there is no need for renewal so no additional logic is needed for these kinds of jobs within the driver or executors).

Upon termination of the job the containers are destroyed and the secrets with the **DT** are removed and also destroyed. However with future short-running jobs, the **TGT** might still remain in the **LTC** so you may re-run spark-submit without a $kinit in that case. To account for such destruction secrets per job will include labels that have: applicationID-timestamp-version. The driver pod will handle this deletion upon completion of the application.

Long Running Job Support

The following will happen to support jobs that last longer than one day. To support such jobs, a renew service pod will extend the lifetime of DT(s) that the job is using.



The design of handling the renewal/refresh of Kerberos **DT** in a Spark on K8S system will be leveraged with a Service pod that has the option to span multiple namespaces or be per-user specified namespace. This Service pod will function similarly to how the **Resource Manager (RM)** of Yarn works in its own renewal/reissue service. The reasoning for this Service pod is its intrinsic generic nature in being able to handle a multitude to HDFS-based jobs including but not limited to Spark: Hive, etc.. , the natural code reusability of Yarn's RM, and the ability to decouple logic away from the driver. The steps to handle refresh/renewal for long running jobs are as follows:

Steps:

0. *Launching Service pod with --keytab and --principal*
   User (client) will be required to launch a service pod by passing in the given **keytab (--keytab)** and **principal (principal)**. The **principal** used by this service may differ from job users' **principal**. The **DTs** however will belong to the job users' account. In other words, this Service pod will be a separate service **principal** that will renew and reissue job users' **DT**s. To allow this, the service principal should be designated by the DTs as the renewal principal. Each DT has a field specifying which principal can renew the DT.

In YARN, this is always the YARN RM principal. For k8s, this will be the principal that the renew service pod is using.
Also the HDFS NameNode should configure the service principal as the proxyuser of the job users' groups. This **keytab** will be mounted in the secret of the namespace and upon creation the Service pod will use the keytab to retrieve a **TGT** for the use of retrieving a **DT** upon requests that hit the service. To retrieve a **TGT**, the service pod will sign-in into the **Kerberos Authentication Server (Kerberos AS)** and receive a **TGT.** (The TGT will be simply cached in a local Kerberos ticket cache dir inside the service pod. I.e. It won't be part of the keytab secret) There will also be a timer thread running inside of the service pod that will be refreshing the **TGT**. The **TGT** class will contain an expiration time and sequence (version) number for the sake of book-keeping. The **TGT** will thusly be stored in the local dir of the Service pod. The Service pod now uses the **TGT** to communicate with the **Kerberos Ticket Granting Service (TGS)** to retrieve an **HDFS Service Token**. This **HDFS Service Token (HDFS SVC)** should be cached to minimize the load that Kerberos takes in constantly calling the **TGS** with the then valid **TGT.** (This will require investigation to determine how long the **HDFS SVC** will last.) This **HDFS SVC** will then be used to communicate with the **NameNode** to retrieve a valid **DT** of a job user. The service pod will now do something we will call **Secret Discovery**. Secret Discovery is when the Service pod uses watchers on the secrets across whichever namespace(s) it is associated, to keep track of newly created secrets with kerberos **DT** logic. These secrets will be book-keeped and updated upon reaching 75% of the expiration date, which is extrapolated from the label. At the same time, the Service pod will be refreshing its **TGT** once it reaches 75% of its expiration date in a separate thread. Upon seeing a book-keeped secret reaching its renewal point, the Service pod will take its valid **TGT** and request from the **TGS** a **HDFS SVC**, or a still valid cached **HDFS SVC**, that it will use to communicate with the **NameNode** to retrieve a **DT.** It will then update the **DT** secret so that the driver and executors will be able to read updated **DT**.
Implementation details for the Service pod could be that there is one per namespace where all of them share the same proxy user where multiple pods (replicas, maybe: replica-set or deployment) can manage credentials for that proxy user, thusly, by putting all of the pods behind a K8s service and by contacting the K8s service you get load balancing.
Notes:

- Storing the keytab as a secret in a separate service pod, allows for tightly restricting access to the keytab for use cases where not all users on a namespace should have access to: i.e. the production keytab directly
- The Service pod will have access to the secrets of all the containers and pods and thusly can update them to propagate the new **DTs** to the Spark Driver Pod and Spark Executor Pod accordingly. This is a key advantage that the Kubernetes architecture has over Yarn.
- The Service pod will also use the namenode config to specify the proxyuser for increased restriction and safety.

- The Service pod can / should be across multiple-namespaces and must leverage an external IP to be seen across multiple names
- Secrets specific to a job should have labels associated with that specific job for **Secret Discovery** by the Service Pod

1. *Initial retrieval of **DT** by Driver*
   Upon startup of the process the initial retrieval of the **DT** by the Driver will function the same way as with the short-running job. As such, the submission client will retrieve the Job User's **TGT** from the **LTC** leveraging the same "reaching-out-step" as explained above. As such, the Driver pod will create a new secret containing the **DT,** with an environmental variable: **HDFS token file location** (**HADOOP_TOKEN_FILE_LOCATION)** pointing to the file mounted on secrets**.**
   The driver pod spec defined by SubmissionStep will thusly specify which secret, with which label, will be mounted for the sake of security and store this secret's location information as an environment variable on the Driver pod for itself and to spawn future executors. At this point, upon launching the Driver pod, the pod will read the contents of the secret and have the **DT** create a valid **Hadoop User Group Information (UGI)** for secure HDFS interaction. The initial contents will be taken from the **TGT** obtained via $kinit or the Job User's principal. As such, the **DT** mounted on the job's namespace secrets will be one obtained via the Job User's principal. Which has the benefits of, if the Service pod is down, there wouldn't be a blocker with this type of implementation and the service pod will know that if there is a secret recently created with a **DT** then the Job User is validated. The driver pod will have the **DT** automatically updated because the service pod will see the newly created secret and book-keep when to update said **DT.** As such, the renewal/refresh logic is abstracted away and there is no request needed to be done by the Driver pod.

2. *Initial retrieval of **DT** by Executor*
   Upon getting spawned by the driver pod with the secret's label and location stored in its environment variables, the executor pod will have the **DT** mounted into its secrets as well as the env variable **HADOOP_TOKEN_FILE_LOCATION** defined. At this point in time, the strategy is to leverage the SparkHadoopUtils.runAsSparkUser() method which will be initialized with a dummy UGI variable (to act as the Yarn NM Kerberos Principal), that will be updated upon the utilization of the **DT** to transfer credentials to the Job User. At this point, the executors all have a valid **DT** to create a valid **UGI** for secure HDFS interaction.
3. *Driver thread and retrieval of new DT upon nearing expiration*
   Because of the expensive nature of Watchers, the driver pod will merely re-read its Secrets to retrieve a new **DT** upon reaching the expiration date, because we are working off of the assumption that if the Service pod is running, then the secret will be updated by then, because of the sufficiently early renewal time by the Service pod **(75% of**

expiration date) in its book-keeping. Now the Driver pod will retrieve the new **DT** from its mounted secrets.

4. *Executor thread and retrieval of new DT upon nearing expiration*
   Because of the expensive nature of Watchers, the executors will merely re-read their Secrets to retrieve a new **DT** upon reaching the expiration date, because we are working off of the assumption that if the Service pod is running, then the secret will be updated by then, because of the sufficiently early request by the Driver pod to ask for an update. Now the executors will retrieve the new **DT** from their mounted secrets

Access Control of Secrets

We store both HDFS DTs and the renew service keytab in K8s secrets. K8s admin users can restrict access to K8s secrets to the desired users or programs.

K8s supports access control of secrets using authorization modules. There are multiple implementations including the new K8s RBAC module. Our access control design will remain neutral to these different plugins. Below, we will occasionally refer to the new K8s RBAC plugin as a reference example. This will happen only when needed.

Any K8s authorization mechanism should deal with two types of entities that will access resources in a K8s cluster:

A. Human users issuing commands outside the K8s cluster.
B. Programs run in K8s pods. They run inside the K8s cluster.

In K8s, identities for (A) human users and (B) pods are usually different. For human users, different organizations may prefer an authentication mechanism that they already used before adopting K8s. So K8s supports multiple authentication plugins for human users.

For pods, K8s has special accounts called service accounts. Even service accounts can be opted out in favor of custom credentials for accessing the K8s API server. Programs running on behalf of users do not need to have all the users' privileges. So pod accounts such as service accounts can have a subset of the privileges required for the programs. Moreover, pod accounts are scoped by namespaces so that the same user can run programs in different namespaces using the smallest subset of privileges for the needs.

So our access control design will also remain neutral to these different authentication mechanisms. Below, we will only talk about human users and pods. They concern us only in that we design how K8s should restrict access to K8s secrets for those human users and pods.

Requirements

- The job owner's HDFS DT should not be exposed to other users or their jobs. Otherwise, they could steal or destroy the job owner's HDFS data. The renew service pod helping long running jobs is an exception in that it should be able to read or replace the DTs.
- The renew service's keytab file should be accessible only to the system admin user or the renew service pod. I.e. It should not be exposed to any job users or their jobs. Otherwise, they could use the keytab file to steal or destroy HDFS data of many users.

Strictly speaking, the K8s secrets containing DT and keytab should be accessed only in the following ways:

	Job owner human user	Job owner's pods	Other human users	Other users' pods	Renew service pods
Access to the DT secret	create	get	none	none	get, update
Access to the renewal keytab secret	none	none	none	none	get

Can K8s authorization mechanisms meet this strict requirements? It depends on which authorization mechanism is used. Below, we will talk about multiple choices that can satisfy this requirement referring to RBAC as an example. But your authorization mechanism, which may not be RBAC, might be able to choose a similar approach.

K8s RBAC uses Role to represent the privileges for accessing K8s resources. A rule in Role can refer to either types of resources, or a list of existing resource instances. A Role is tied to a specific namespace. For example, RBAC Roles can express:

1. The "**read"** access to **all** secrets in the namespace **foo**.
2. Or, the "**read"** access to the existing **secret-1, secret-2, …, secret-10** in the namespace **foo**.

The current version of RBAC can't select a list of resource instances dynamically based on a user-specified predicate such as a K8s label:

- The "**read**" access to **all** current secrets in the namespace **foo** that has label **owner=john**

But we can imagine other authorization mechanism or future RBAC versions support (A). So we'll include this in the design choices below. When (A) is not supported, one could consider namespaces and (1) as a way to handle a dynamic list of secrets.

In RBAC, RoleBindings can grant Roles to human users or pod service accounts. Just like Role, RoleBinding is tied to a namespace. A service accounts is also scoped by namespaces. When these are used together, the namespace should be the same one. So, a RoleBinding can express:

- Grant the "**read"** access to **all** secrets in the namespace foo (Role 1 above) to human user **john**, and the service account **bar** in the namespace **foo**.
- Grant the "**read"** access to all secrets in the namespace foo with label **owner=john** (Role A above) to human user **john**, and the service account **sa-john**. Also grant access to the service account sa-john only to the human user john.

ClusterRole and ClusterRoleBinding can be used to express access and granting across namespaces. Note ClusterRoleBinding can be used together with a Role, tied to a namespace, in order to give the access to accounts in different namespaces.

Solutions

Now, we can map the above requirements to access control setup. All solutions below except (5) assumes that the cluster admin user creates the ACLs as a one-time setup. We used request verbs that K8s API server uses. In particular, verb "get" is for reading resources.

1. Create a namespace per user

For each user, create a dedicated namespace for the user, say namespace **john-ns** for user **john**. Create a single service account say john-sa or an equivalent credential in the namespace, and grant the access to the human user account john. The user john will launch pods using the granted access to the pod account in the namespace. Grant the human user and pod account access to all secrets in the namespace.

In case the jobs should run longer than a day, add another namespace, say renewal-ns for the renew service pod will run in. Create a service account say renewal-sa or an equivalent credential in this other namespace, and grant access to the admin user account say tom who will launch the renew service pod. Grant the admin user access to all secrets in the john-ns. (In RBAC, this requires ClusterRole and ClusterRoleBinding) Also grant the admin user and the pod account access to all secrets in the renewal-ns.

When the user john submits a spark job, he will specify the namespace and the service account to use, say john-ns and john-sa respectively. The submission client will create a K8s secret in the namespace while storing a DT in it. Then the client will launch the driver and executor pods in the namespace using the service account. The launched pods will read the DT in the secret.

In case of long running jobs, the renew service pod in the renewal-ns will read/update the DT in the DT secret.

For this use case, the cluster admin user can set up the following access control:

	Job owner, **john**	Job owner's pod account **john-sa**	Other users	renew service owner, **tom**	Renew service pod account **renew-sa** in namespace **renewal-ns**
Access to **all** secrets in namespace **john-ns**	create	get	none	none	get, update
Access to pod account **john-ns/john-sa**	get	N/A	none	none	none
Access to resources in namespace **renewal-ns**	none	none	none	Full access	Full access

2. Create a namespace per user group. User secrets are exposed to other users in the same namespace.

It can be the case that creating a namespace for each and every user is too expensive. Then, one can create a namespace for each group of users, say **finance-ns** for the **finance** group. Then grant the namespace access to every user in the group, say **john** and **jerry**. Create as many service accounts or the equivalents as needed. (Below, we assume each human user has a separate service account or the equivalent, but it doesn't need to be the case. Some of them could share some service accounts.) Grant all users and pod accounts access to all secrets in the namespace.

This works if users of the group do not mind exposing their DTs to other users in the group. For instance, they may not have meaningful personal data and they all assume others in the group are well-behaving.

For this, the cluster admin can reuse access control setup of solution (1) with minor modification.

	Job owner in the finance group, **john**	Job owner's pod accounts **john-sa**	Job owner in the finance group, **jerry**	Job owner's pod accounts **jerry-sa**	Users in other groups	renew service owner, **tom**	Renew service pod account **renew-sa** in namespace renewal-ns
Access to **all** secrets in namespace **finance-ns**	create	get	create	get	none	none	get, update
Access to pod account **finance-ns/john-sa**	Use	N/A	none	none	none	none	none
Access to pod account **finance-ns/jerry-sa**	none	none	get	N/A	none	none	none
Access to resources in namespace **renewal-ns**	none	none	none	none	none	Full access	Full access

3. Create a namespace per user group. Use pre-populated secrets per user.

A variation of solution (2) is possible. All users in a group share a single namespace. But unlike (2), the users have personal HDFS data and want to keep the data private. But the authorization plugin does not know how to find a list of users' secrets dynamically say using a label per user, which is the limitation of the current RBAC version. Like RBAC, The authorization plugin supports applying privilege to a list of named secret instances.

So the admin user creates a list of secrets, say john-secret-1, john-secret-2, etc per user john. Then the admin sets up ACLs that would grant the user access to the pre-populated secret lists.

The pre-populated secrets may be empty at the start. When a user submits a spark job, he will specify the secret name to use. The submission client will find the secret and store a DT in it. The rest is same as before.

Note the renew service does not need to run in a separate namespace. We no longer grant all users blank access to all secrets in the namespace. The keytab DT secret can also pre-populated and then be kept private to the renew service pod account. The table below assumes the renew service uses the same namespace.

	Job owner in the finance group, **john**	Job owner's pod accounts **john-sa**	Job owner in the finance group, **jerry**	Job owner's pod accounts **jerry-sa**	Users in other groups	renew service owner, **tom**	Renew service pod account **renew-sa** (in the same finance-ns namespace)
Access to john's pre-populated secrets **john-secret-1, john-secret-2, etc** in namespace **finance-ns**	update	get	none	none	none	none	get, update
Access to pod account **finance-ns/john-sa**	get	N/A	none	none	none	none	none
Access to jerry's pre-populated secrets **jerry-secret-1, jerry-secret-2, etc** in namespace **finance-ns**	none	none	get	get	none	none	get, update
Access to pod account **finance-ns/jerry-sa**	none	none	get	N/A	none	none	none
Access to tom's pre-populated secret **tom-secret** for renew service keytab in case the renew service uses the same namespace **finance-ns**	none	none	none	none	none	Use	get
Access to pod account **finance-ns/renew-sa**	none	none	none	none	none	get	N/A

4. A namespace per user group. Label secrets per user, assuming authorization plugin can select labeled secrets.

This is a variation of (3). The authorization plugin is capable of selecting a list of resource instances dynamically based on a user-specified predicate such as a K8s label:

- Read access to **all** current secrets in the namespace **foo** that has label **owner=john**

So, instead of a list of pre-populated secrets used in (3), the job client will create secrets and add labels to them indicating the intended pod accounts. The The cluster admin sets up ACLs using the labels.

Like (3), the renew service does not need to run in a separate namespace. The table below assumes the renew service uses the same namespace. Also we assumes the renew service keytab secret is pre-populated.

	Job owner in the finance group, **john**	Job owner's pod accounts **john-sa**	Job owner in the finance group, **jerry**	Job owner's pod accounts **jerry-sa**	Users in other groups	renew service owner, **tom**	Renew service pod account **renew-sa** (in the same finance-ns namespace)
Access to **all** secrets in namespace **finance-ns**	create	none	create	none	none	create	none
Access to secrets with label **owner=john**, in namespace **finance-ns**	none	get	none	none	none	none	get, update
Access to pod account **finance-ns/john-sa**	Use	N/A	none	none	none	none	none
Access to secrets with label **owner=jerry** in namespace	none	none	none	get	none	none	get, update

finance-ns							
Access to tom's pre-populated secret **tom-secret** for renew service keytab in case the renew service uses the same namespace **finance-ns**	none	none	none	none	none	none	get
Access to pod account **finance-ns/renew-sa**	none	none	none	none	none	get	N/A

5. A namespace per user group. Label secrets per user assuming a controller pod will creates label-based ACLs on the fly.

This is a variation of (4). The authorization mechanism is not capable of selecting labeled secrets dynamically. (There is a list of open issues -- #44703, #40403, #816) Instead, we run a controller pod that will discover those labeled secrets and create the desired ACLs on the fly.

Note the controller pod will have to use a particular access control mechanism such as RBAC to create the ACLs, preferably using plugin. So this approach will be biased toward specific authorization mechanisms. The controller pod will need the capability of creating ACLs. In addition, the secret labels will have to be well defined to allow programmatic creation of ACLs. For instance, a secret would need multiple labels, one for each different account, say owner=john, reader=john-sa, and updater=renew-sa.

It is possible to merge the controller code into the renew service pod. We don't draw the ACL table since it will be very similar to that of (4).

Implementation

This system requires a variety of Kubernetes Primitives including: Secrets, RBAC, and Watchers in addition to those provided by the Spark-on-K8S code.
Secret will be used by:

- The Service Pod to hold the **keytab** and **principal**
- The Driver Pod to hold the **DT**
- The Executor Pod to hold the **DT**

RBAC will be used by:

// TODO

Watchers will be used by:

- The Service Pod to watch for newly created and recently destroyed secrets

Phases:

Ilan [Bloomberg]

- Integration Testing Environment for E2E (here)
- Submission Client
- Later (Service Pod)

Kimoon [PepperData]

- Driver Pod
- Executor Pod
- Later (Service Pod)

Step-by-Step Requests

Start of Short / Long Running Job

Step 0. [if Long Running Job]

User (client) will launch Service Pod + ensure spark Conf includes long running job flag

Step 1.

User (Client) will $kinit to store **TGT** in **LTC**

Step 2.

Submission Client will request **HDFS SVC** with **TGT** from Kerberos **TGS**

Step 3.

Submission Client will request **DT** with **HDFS SVC** from NameNode

Step 4.

Submission Client will store mount **DT** in Namespace secret

Step 5.

Driver pod and executor pods will be launched reading from that secret

End of Short Running Job

Step 6. [if Long Running Job]

Service Pod will refresh DT for the job and update the corresponding secret, which is discovered by using **Secret Discovery** to book-keep newly created and recently destroyed secrets

Step 7. [if Long Running Job]

The Driver pod and the executors will use their expiration date to re-pull from the secrets, which will be automatically updated by the Service Pod.

End of Long Running Job

Discussions

1. Who should create the DT and the secret? The job client or the renew service pod?

[Ilan] Upon initial retrieval of the first DT the submission client will mount the DT as a secret within the namespace using the Job User's principle. After that point, the service pod will be updating that secret with a different principle passed in upon the creation of the service pod. The service pod will function as a superuser that has access to all namespaces.

[Kimoon] I agree with Ilan's approach above. I.e. The job client creates the DT and the secret, both for the short job and the long-running job. IMO, this has a few nice benefits:

A. The initial DT will be created under the job user's principal using Kerberos authentication. Upon the presence of the initial DT with a proper label, the service pod can trust it was created passing the authn. (If we let service pod create the initial DT, we have to have our own authn code when a bad job asks DTs for other users.) This serves as implicit registration that tells the service pod that a particular app/job needs secured HDFS access for prolonged period. See discussion item 4 below that talks about the discovery mechanism.
B. The service pod can briefly go down for various reasons without blocking jobs from launching. (If initial DTs are created by the service pod, this brief outage can block a lot of jobs from starting.)
C. The short job design and long-running job design converges for the job bootstrap part. In particular, the Spark code will be pretty similar between short and long-running job. The long-running job will add extra thread or something for checking replacement DTs, but that can be added as extension on top of the initial bootstrapping code.

2. How does access to the DT secrets get restricted to the driver, executor or the renew service pod?

[Kimoon] We are relying on k8s RBAC on this. See the secret access control section above. But, according to Matt's comment, custom authentication provider might not implement RBAC at all and use alternative mechanism. So we shouldn't be hostile to such setups.

3. Who should cancel the DT and the secret when the job is done? The driver or the renew service pod?

[Kimoon] We should check if canceling the DT may require a valid Kerberos TGT. (Canceling DT allows the namenode to remove the entry from its memory) The job driver does not have a valid TGT. If it does not require TGT, I prefer the job driver to revoke the job DT and delete the secret for both short jobs and long-running jobs. So that the implementation will be simpler. There is one worry that buggy jobs may fail to clean up. Note that in Yarn, it's always the RM that does the clean up, perhaps for this concern. If TGT is required, then this will be reason to use the service pod for both short and long running jobs.

4. How does the renew service pod find DT secrets to renew? Should the driver send an explicit RPC request for registration or should we let the service pod discover the secrets that has well known labels?

[Kimoon] I'm imagining the service pod can discover on its own which job secrets to renew. I.e. Avoid using explicit registration RPCs. Maybe jobs can add labels to DT secrets to help the discovery. There are two nice things about this approach, IMO.

A. it's easy for apps (possibly beyond Spark) to start adopting the renewal service since there is not a lot of coding work to do.
B. When the service pod briefly failed and is recovering, it can reconstruct the state by doing the discovery again. If we have explicit registration protocols, apps may have to re-register with the service pod which means more code on the app sides to get right. What do you think?

This works together well when the initial DT is created by the job client under authenticated Kerberos principal. See discussion item (1) for details.

5. Which namespaces' DT secrets should the renew service pod be in charge of?

[Kimoon] It should be a launch option. It can be one of the following options

A. Just the namespace of the service pod itself
B. A list of user-specified namespaces.
C. All namespaces. Ilan mentioned this to be an important use case in a comment.

Option (B) and (C) assumes that the service pod can upate secrets in namespaces other than the one it is running on, which should be possible using ClusterRole and ClusterRoleBinding of K8s RBAC.

6. How does access to the keytab secret get restricted to the renew service pod(s)?

[Kimoon] Again, we assume we can rely on RBAC. Note access to the keytab secret will be authorized only to the service pod(s). Spark job driver/executor pods will never be allowed to access the keytab secret.

7. What should be impact of the renew service pod outage? Should the job tolerate the outage or should it fail?

[Kimoon] Yes, the job should be able to tolerate a brief service pod outage. To this end, the renew/reissue effort for an expiring DT should start sufficiently earlier than the expire time and continuously retried to ensure long window for success. The service pod can probably use a K8s Deployment to automate the pod restart.