Timeline Server - Generalized ACLs

Last Modified Date: August 4 2014

Background and Problem Statement

Timeline server stores application specific information in terms of a number of entities and events. An entity can be an arbitrary concept related the application, while an event represents what happens to the entity over time.

Since entities and events may contain confidential information of an application, we need to have ACLs to control the access to this data. Currently, timeline ACLs generally work as follows:

- The creator of an entity is defined as the owner of the event
- Only owner and admins can update and access the entities; and
- The access control is at the granularity of a single entity.

However, the current ACLs are not satisfactory enough to serve our use cases. As of today,

- We don't distinguished reading and writing operations. We may want to allow some users to put and modify entities & events, while allowing others to query them.
- Users may want to put multiple related entities, for example a number of jobs under a workflow, and organize the entities in the way that all the job entities are related to the workflow entity as their parent. In this case, they want to prevent others from linking a malicious job entity to the workflow entity. With current entity-level ACLs, we overlook the entity relationship, and open a security hole for attackers to inject unrelated timeline data into users' workspace as the related entities.

Therefore, we need to figure out a generalized solution to control users' access to the timeline data, and take the entity relationship into account.

Solution

Two types of ACLs

We need to make two primary changes on the current timeline ACLs. First of all we need to differentiate reading and writing ACLs:

- *Reading*: with reading ACL, the user can query and retrieve entities and events;
- *Writing*: with writing ACL, the user can put and update entities and events.

Namespace and Namespace ACLs

Secondly, we need to give up current entity-level ACLs. Instead, we should define a timeline **namesapce**, which serves as an umbrella of all the entities in it. In general, the application (or the user) need to define a namespace, and specify who are users that can read or write timeline entities in it. Later on, when the application put a timeline entity, it can specify the namespace to put. Below are some details about the timeline namespace.

- The namespace is a new data model, whose schema can be as follows:

  TimelineNamespace {
  - ID: A user defined identifier, which must be unique across the whole store.
  - Name/Description (optional): Additional description about the namespace.
  - Owner: The user who creates the namespace.
  - Readers List: The users who can read the entities in this namespace, by default, the owner only.
  - Writers List: The users who can write the entities in this namespace, by default, the owner only.
  - Created time: the timestamp when the namespace is created.
  - Modified time: the timestamp when the namespace is modified (updating the ACLs).

  }

  It is possible that multiple tenants on the same YARN cluster may conflict each other with TimelineNamespace ID. Assume the authorized users on the cluster are reasonable and cooperative, we require them to come up a meaningful and specific enough ID, such as <APPLICATION_TYPE>_<JOB_NAME>_<Number>, to avoid the conflicts. In the same namespace, each entity's identifier is supposed to be unique, while different namespaces are allowed to both contain the entities having the same identifier.
  In the data model of TimelineEntity, we need to add one more field to record which TimelineNamespace it is put into.

  TimelineEntity {
  - …
  - namespaceId: the ID of the namespace which contains this entity.

  }

- A user can create a namespace, and he is going to be the owner automatically. The user, which is the owner of the namespace, is allowed to modify the ACLs fields in the future. To allow users to create the namespace, and to get back the namespace they have created before, we should provide additional APIs as follows:
  - POST /timeline/namespace: Post an namespace.
  - GET /timeline/namespace: Get all the namespaces belonging to the user of the request.

    - GET /timeline/namespace/{id}: Get a specific namespace.

  Please note that

  The namespace should be persisted into the timeline store as well in the similar way we persist the entity, but in a different schema for namespace only.
- As other YARN components, the timeline server should be able to support the admin ACL. We allow the system-wide admin to modify the ACLs setting, as well as read/write the entities of any entity.
- When the timeline ACLs is not enabled, we need to make anything work seamlessly. To do this, the system need to have a default namespace, whose reader and writer list are set to anybody. Therefore, all the entities are put into the default namespace when the namespace is not specified. Then, anybody can update or access theirs and others' entities.

  When the timeline ACLs is enabled, users should take the following additional steps. The user needs to create a TimelineNamespace first. Once the timeline namespace is created, the user can put the entities into it by specifying the namespace ID. However, it is possible that the application forgets to or intentionally doesn't specify the namespace ID. There are two options to handle this case:
    - The uploaded entities are going to be rejected, and the put response is going to notify the user that the namespace is missing.
    - It is also possible to handle it as the ACLs is not enabled. The entity is automatically falling into the default namespace.

  When using the GET API to retrieve the entities, only the entities belonging to the namespace where the user has reader access will be returned.
- TimelineClient should provide corresponding method to post the namespace. When the namespace is created successfully, one optional thing is to cache the namespace, whenever the user post an entity later, the namespace will automatically be loaded in the entity post, thus relieving user from always editing the namespace field every time.
- An application has multiple stakeholders that can upload the timeline data to the timeline server, i.e., the client, the AM and containers. Therefore, all the stakeholders should be coordinate on the common namespace to be used. For example, in a typical process, if the client creates a namespace upfront, it can put this namespace ID into the environment of the CLC of the app submission context, such that the AM is able to have the ID as well. Later on, the AM can put this ID into the CLC to start the containers. Moreover, if multiple applications of a single user want to share a common namespace, these applications can source a common conf file for the namespace ID, or get the ID passed by the guidance app. Anyway, this is the userland work:
    - It's up to users to define the logic to make different pieces in an application can get the same namespace.
    - It's up to users to define how many applications should share a common namespace, or how many namespaces an application want to use.

Use Case

Let's just talk about a typical MR job first.

- The user can specify the namespace ID in mapred-site.xml or even parse it via CLI arg (in the latter way, we can easily make each individual MR job to have a different namespace ID). When the client is up, it contacts the timeline server to create a namespace according to supplied ID, and it can either reuse MR job ACLs or use the dedicated ACLs config for the timeline data for the namespace's ACLs. After that, the MR client can upload the timeline data into the namespace. It may have several scenarios here:
    - The given namespace has been created before, and then the MR job can simply reuse it.
    - The given namespace hasn't been created before, and then the MR job client creates one, and use it.
    - The namespace ID is not given, and then the MR job is going to directly put the timeline data to the system default namespace (or being rejected).
- To make MR AM be able to put the timeline data into the same namesapce, when client's submitting the job to YARN cluster, in the container launch context of app submission master, the client needs to pass in the namespace ID as an env var. Then, when the MR AM is up, it could read the evn var, and setup a TimelineClient to post the timeline data to this namespace as well.
- Furthermore, if MR job wants to post the timeline data from containers, this namespace ID should be passed to the container via CLC when AM starts a container. This has to be taken care of by AM as it is in charge of the container life cycle.

Let's take an Oozie workflow of multiple MR jobs as an example. It's pretty much like the single MR job example, but the high-level controller wants to distribute the namespace ID among different jobs.

- Before the Oozie client starts any MR jobs, and it can create a namespace for this workflow.
- Let's assume Oozie client is going to launch two Hive queries, the Oozie client needs to be modified to provide the namespace information to the Hive client to run the queries (perhaps adding the namespaces as the CLI param when invoking the Hive client).
- Consequently, when the Hive query is translated into Tez/MR jobs, the namespace is going to be passed to the AM via the app submission context.
- All these stakeholders can then put entities into the common namespace.
- Later on, the user can use TimelineClient to update the ACLs of the namespace to add some other users to the reader list, such that the can query the timeline data in this domain to analyze the workflow.

Last but not least, let's look at a sample query with ACLs control enabled:

```
GET http://ts-host:8188/ws/v1/timeline/TEST_ENTITY_TYPE
```

And we have the timeline data stored as follows:

```
Namspace1 {
```

- Entities: Entity1 { type = TEST_ENTITY_TYPE},
- Readers: User1

}

Namspace2 {

- Entities: Entity2 { type = TEST_ENTITY_TYPE},
- Readers: User1

}

Namspace3 {

- Entities: Entity3 { type = TEST_ENTITY_TYPE},
- Readers: User2

}

If the aforementioned sample query is run by User1, the expected result set is going to contain Entity1 and Entity2, but not Entity3. The query will run across the namespaces, but only return the entities that is visible to the user.

Implementation

We need to do the following steps to implemented the aforementioned generalized ACLs:

- Define the POJO class TimelineNamspace, using it as the payload to communicate the ACLs setup between the client and the timeline server. In TimelineNamspace, we need to add one more field to record the ID of the namespace where the entity will be put.
- The timeline services need to open two more APIs:
  - *putNamespace*: the client can create a new namespace or update an existing one;
  - *getNamespace*: the client can get a list of namespaces of the current user, or a particular namespace if the namespace ID is specified.
- TimelineStore needs to have two more APIs accordingly to query and persist the namespace information, and the Leveldb implementation and the Hbase implementation (in the future) need to implement the two APIs.
- At TimelineClient, we need to provide the method to put the namespace, which wraps over the REST API.

Other questions/concerns

- Deleting entities and namespace as end-user APIs (for retention, the old entities entities will be automatically discarded. After this, some old namespaces will be empty, and can be deleted).
  - Only owner or admin can delete a timeline namespace. It is possible that the namespace is not empty when the application wants to delete it. There are two options here as well:

- A timeline namespace should not be allowed to delete unless the namespace is empty;
- When the user forces to delete the namespace, the entities and the events in this namespace will be deleted consequently.