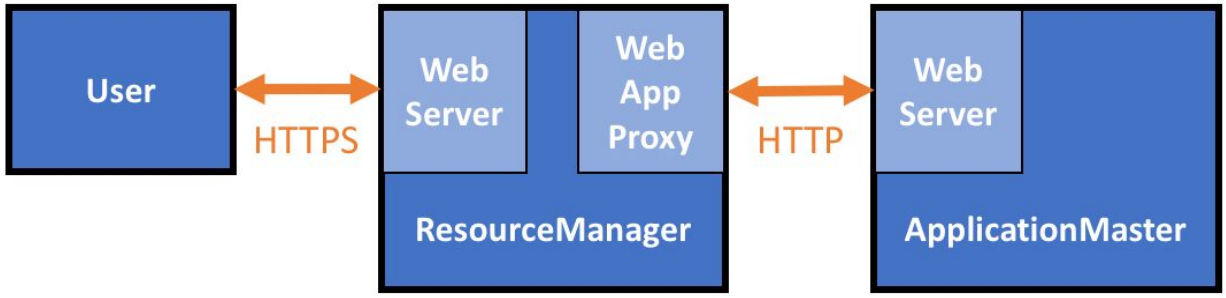
AM Web UI HTTPS Design Document

Robert Kanter

Background

Today, when you enable SSL/TLS/HTTPS on your Hadoop cluster's Web UIs, it only affects daemon processes like the ResourceManager, NameNode, Job History Server, etc. The one place where it's missing is the ApplicationMaster.

The reason nobody has really noticed this, and why it's a less severe issue, is that we typically proxy the connection through the WebApplicationProxy, which lives in the RM. So when a user goes to the AM's webpage, they're actually going through the RM, and the page appears to be encrypted over HTTPS with the RM's certificate. However, in the backend (which should at least be in an internal network), the WebApplicationProxy is talking to the AM's Web Server over HTTP. This is shown in the following diagram:



Why Solving this is Complicated

The main issue is that the AM is considered untrusted user code. We can't simply give it the same keystore as the RM, and we don't want to give some shared "real" certificate (e.g. from a real CA or from a company-wide CA) because the user running the AM could steal it and now has a seemingly valid certificate that they could do something malicious with.

That leaves us with the option of requiring users to bring their own certificate, either a "real" one or a self-signed one; that way, there's no issue of stealing. However, doing this is hard and we can't rely on the user to be able to do this on their own. Plus, it would be best if the WebApplicationProxy (i.e. the RM) could (safely) trust the certificate provided by the AM; which would be problematic if the user provided their own.

High Level Proposed Solution

The RM will act as a CA, using a self-signed CA certificate that it generates. When launching an AM, it will generate a child certificate (from of the CA certificate) for that Application, and securely pass it along to the Application, the same way it does today with delegation tokens. The AM can then use that certificate when talking to the WebApplicationProxy, which will accept it because it was the one who signed it. We can also allow for mutual authentication so that the AM can verify that it's the WebApplicationProxy who's talking to it.

Design Considerations

These are some goals that were kept in mind during the design:

- This is more about providing wire encryption than it is about proving identify, because the receiver is untrusted anyway.
- This is for internal Yarn communication, so a user's browser does not need to (easily) accept the certificate; in fact, we likely don't want it to.
- It should be easy for the cluster admin to configure/setup.
- This should be as easy as possible for the Application to take advantage of. It doesn't need to be completely free, but it shouldn't require extensive refactoring or complicated setup to use.
- The RM already tracks a lot of information per Application (which goes into the RMStateStore). We should try to add as little new info as possible - in other words, we should not add a new piece of data per Application to keep track of.
- Most users run the WebApplicationProxy within the RM. We don't need to worry about users who run it standalone.
- It should be flexible to still allow for AMs that don't want SSL, or that want to use a real certificate, etc

Details for Proposed Solution

The design is actually pretty straightforward, but requires changes in a lot of areas so we'll break this down into different steps.

Step 1 - The RM is Now a Certificate Authority

On startup, the RM will generate a self-signed certificate to use for signing certificates destined for AMs. It will act as a CA in this way. CA certificates typically last between 10 and 30 years, and are often capped at 2037 (to avoid any complications with the Y2K38 problem); so we can simply hardcode the end date to sometime in 2037 (~19 years as of now).

Certificates have a distinguished name that's currently defined in RFC 2253. For the RM CA certificate, we'll set the "OU" field to some unique value that also identifies it as YARN: "YARN-<random UUID>". This will ensure that each YARN cluster has a unique name.

The other important part of generating a certificate is the public and private keys. The RM will randomly generate 2048 bit RSA keys and sign certificates with SHA-512.

To keep things simpler, we won't be rolling the CA certificate. If desired, that can be done as a follow up JIRA. Note that rolling complicates things because of the mutual authentication whereby the AM needs to be able to verify the RM's certificate (see Step 4).

When RM HA is enabled (or work-preserving RM restart), the RM will store the CA certificate, CA public key, and CA private key in the RMStateStore. When a failover occurs, this will allow the new active RM to have the same CA certificate, CA public key, and CA private key as before. This is important to ensure that it can verify previously generated certificates and for mutual authentication.

Step 2 - Generating Child Certificates

When starting an AM, the RM will use its CA certificate, CA public key, and CA private key to generate a child certificate for that AM. We can limit the lifetime of these certificates to the max lifetime of the RM Delegation Token (default is 7 days). However, we can make the expiration date of the generated certificates configurable in order to handle Applications like Spark Streaming that can run longer.

Most certificates have a field in their distinguished name called "CN" which is set to the hostname so that clients/browsers can verify it. In our use case, we don't care about specific nodes - we care about Application IDs. So we're going to set the CN to the Application ID. The WebApplicationProxy can then verify that the Application ID, as reported by the CN field in the certificate of the Web Server it contacted, matches the Application ID that it's expecting to contact. This ensures that a user can't re-use Certificates generated by the RM CA for other Applications (the IDs won't match).

As with the CA certificate, we'll use randomly generated 2048 bit RSA public and private keys for the child certificate. The child certificate will be signed using the CA's private key - this ensures that nobody can tamper with it, and that it can be verified with the CA's public key. The child certificate, as well as the CAs certificate, will then be placed into a keystore. The CAs certificate needs to be included because it's part of the certificate chain. The child's private key will be associated with the child certificate in the keystore. Once this is prepared, the RM doesn't need the child public or private keys anymore, so there's no need to store them anywhere.

Step 3 - Distributing Child Certificates

The RM needs to provide the following items to the AM:

- Keystore file
- Truststore file
- Keystore password
- Truststore password

Preparation of the keystore was already described in Step 2. For the truststore, we need to provide the CA's certificate. Later on (see Step 4), the AM will need this in order to trust the RM's certificate in mutual authentication.

To securely distribute the keystore and truststore files from the RM to the AM, we can utilize the Credentials infrastructure already in place. The Credentials are where delegation tokens and other secrets are passed from the RM to the AM via the NodeManager. By re-using this infrastructure, we don't have to reinvent the wheel here. The PoC does this by writing the bytes for the keystore and truststore files as secret keys in the Credentials, but in the final implementation, we can add new fields to Credentials instead. The keystore and truststore should be about 3kb of data total, which is relatively small.

The passwords for the keystore and truststore are needed in order for the AM to actually open the files. We can randomly generate these and pass them along the same way we do for the keystore and truststore (as secrets or as new fields).

In order to simplify things for the AM, the NodeManager can take the keystore and truststore data out of the Credential, and write it to an actual file (with proper permissions, of course). It can also inject environment variables KEYSTORE_FILE_LOCATION and TRUSTSTORE_FILE_LOCATION. The NM already does something like this today for delegation tokens. The NM can do something similar with the keystore and truststore passwords: KEYSTORE_PASSWORD and TRUSTSTORE_PASSWORD.

Step 4 - Using HTTPS

The results of Step 3 are that the AM is now provided environment variables for the keystore and truststore files and passwords. It is up to the developer writing the AM to actually use these; however, that's pretty straightforward. We'll add detail to the docs and implement this for the MR AM.

The AM can also optionally enable mutual authentication. This is simply a boolean config in the AM's web server that we'll also document. This allows the AM to be sure that the request is coming from the RM by cross-checking with the truststore (that's done automatically by the web server as long as the truststore is configured properly). Today, the MR AM uses an ip filter to

ensure that requests only come from the RM; however, this is easily fooled by spoofing your IP or by simply making the request from the RM host. Using mutual authentication is much more robust for this purpose.

The other step that the AM developer needs to do is set the tracking URL to an HTTPS address, which is done as part of the AM registration process. Again, this will be documented.

As mentioned in Step 2, the CN on the child certificates will be set to the Application ID. We'd modify the TrustManager used by the WebApplicationProxy in the RM to verify that the Application ID is as expected. We'll also have the TrustManager check that the certificate is properly signed using the CA's public key (that's why it's important that we signed the certificates using the CA's private key). And for mutual authentication, the WebApplicationProxy will present the CA certificate as the client.

Configuration

This feature should be entirely opt-in. That means we need to be flexible in the configuration and usage:

- It is up to the AM to set the tracking URL to an https address. If set to an http address, the RM won't bother generating and providing the keystore and truststore.
    - In the case where an AM is providing its own certificate (so it sets the tracking URL to https), the RM would still end up generating and providing a keystore, but the AM can simply ignore it.
- The RM will have a new config which specifies the enforcement of HTTPS for the AM proxying. The modes are:
    - OFF - The RM will do nothing special (i.e. what we do today)
    - OPTIONAL - The RM will generate and provide a keystore and truststore when an AM gives it an https tracking URL. It will still accept http URLs though. This will be the default.
    - REQUIRED - The RM will always generate and provide a keystore and truststore and require that the tracking URL for all applications is https.
- Mutual authentication is a server setting (in our scenario, the AM is the server and the RM proxy is the client), so enforcement is up to the AM. The RM will always provide the truststore when an https tracking URL is provided, so the AM will always have what it needs for this, if it opts to use it.
- We'll add a new MR config that will opt in to this feature on for the MR AM, including mutual authentication. It will be off by default.
- As mentioned in Step 2, we'll also add a configuration to the RM for it to set the expiration date of the child certificates. The default value will be the same length as the max lifetime of the RM Delegation Tokens: yarn.resourcemanager.delegation.token.max-lifetime (default is 7 days), but it can also be set to an arbitrary number of days in order to handle longer running Applications.

Proof of Concept

Attached to the JIRA is a PoC patch. Its purpose was to verify that the above design does in fact work successfully, given how non-trivial it is to configure and setup SSL certificates, let alone in the way that we're proposing. It hardcodes a few things, is missing some things (e.g. no RM HA), has no real unit tests, and is not flexible (e.g. you have to only use SSL); but the final version of this will do all of this properly, as per this design document.