Security

Note: Please check the security documentation for the features that Kafka supports today. This page is no longer maintained, but it is kept for historical reasons.

Overview

The following is a proposal for securing Apache Kafka. It is based on the following goals:

1. support authentication of client (i.e. consumer & producer) connections to brokers
2. support authorization of the assorted operations that can take place over those connections
3. support encrypting those connections
4. support security principals representing interactive users, user groups, and long-running services
5. security should be optional; installations that don't want the above features shouldn't have to pay for them
6. preserve backward compatibility; in particular, extant third-party clients should still work

Current implementation efforts are tracked in KAFKA-1682.

Features In Scope

- Authentication via SSL & Kerberos through SASL
- Auditing
- Authorization through Unix-like users, permissions and ACLs
- Encryption over the wire (optional)
- It should be easy to enforce the use of security at a given site

We think all this can probably be done in a backwards compatible manner and without significant performance degradation for non-secure users.

We plan to only augment the new producer & consumer implementations, and not the 0.8 implementations. This will hopefully drive adoption of the new implementations, as well as mitigate risk.

Details on these items are given below.

Features Out Of Scope (For Now)

- Encryption/security of data at rest (can be addressed for now by encrypting individual fields in the message & filesystem security features)
- Encryption/security of configuration files (can be addressed by filesystem security featuers)
- Per-column encryption/security
- Non-repudiation
- Zookeeper operations & any add-on metrics
- Provisioning of security credentials

Authentication

We need to support several methods of authentication:

- SSL for access from applications (must)
- Kerberos for access on behalf of people (must)
- Hadoop delegation tokens to enable MapReduce, Samza, or other frameworks running in the Hadoop environment to access Kafka (nice-to-have)
- LDAP username/password (nice-to-have)
- All connections that have not yet been authenticated will be assigned a fake user ("nobody" or "josephk" or something). <- Note, admins should be able to disable fake users - auditors hate those.

Kerberos is the most commonly requested authentication mechanism for human usage. SSL is more common for applications that access Kafka. We would like a given broker to be capable of supporting both of these, as well as unauthenticated connections, at the same time (configurable, of course). We envision adding additional SASL mechanisms in the future (Hadoop, e.g.)

The LinkedIn security team (Arvind M & Michael H) suggest allocating one port to on each broker for incoming SSL connections, one for all authentication mechanisms over SASL, and optionally a third for open, or unauthenticated incoming connections.

A port dedicated to SSL connections obviates the need for any Kafka-specific protocol signalling that authentication is beginning or negotiating an authentication mechanism (since this is all implicit in the fact that the client is connecting on that port). Clients simply begin the session by sending the standard SSL CLIENT-HELLO message. This has two advantages:

1. the SSL handshake provides message integrity
2. clients can use standard SSL libraries to establish the connection

A dedicated SASL port will, however, require a new Kafka request/response pair, as the mechanism for negotiating the particular mechanism is application-specific. This opens up the possibility of downgrade attacks (wherein an attacker could intercept the first message to the server requesting one authentication mechanism, and modify the message to request another, weaker mechanism). We can protect against that by designing the protocol to request a single authentication mechanism on the part of the client (that way, an attempted downgrade attack will result in handshake failure downstream).

Through this protocol, we could even support unauthenticated connections on the SASL port, as well.

A quick sketch:

1. Client connects on the SASL port
2. Server accepts, registers for reads on the new connection
3. Client sends a (newly-defined) Authentication Request message containing an int indicating the desired mechanism, along with an optional initial SASL response packet
4. Server can reject the request if it's not configured to use the requested mechanism, but if it does, it responds with with the SASL challenge data
5. Client replies with SASL response data

N.B. This document originally stated "We will use SASL for Kerberos and LDAP.", but AFAICT there is no SASL mechanism covering LDAP (and the Java SASL library doesn't support it, at any rate).

Administrators should be able to disable any authentication protocol in configuration. Presumably this would need to be maintained in the cluster metadata so clients can choose to connect to the appropriate port.

This feature requires some co-operation between the socket server and the API layer. The API layer will handle the authenticate request, but the username will be associated with the connection. One approach to implementing this would be to add the concept of a Session object that is maintained with the connection and contains the username. The session would be stored in the context for the socket in socket server and destroyed as part of socket close. The session would be passed down to the API layer with each request and we would have something like session.authenticatedAs() to get the username to use for authorization purposes. We will also record in the session information about the security level of the connection (does it use encryption? integrity checks?) for use in authorization.

All future checks for authorization will just check this session information.

Authorization

N.B. This is still under discussion; I've tried to pull together the current consensus here.

Regardless of the mechanism by which you connect and authenticate, the mechanism by which we check your permissions should be the same. The side effect of a successful connection via SSL with a client certificate or a successful authentication request by some other means will be that we store the user information along with that connection. The user will be based along with the request object to KafkaApis on each subsequent request.

Security has to be controllable on a per-topic basis with some kind of granular authorization. In the absence of this you will end up with one Kafka cluster per application which defeats the purpose of a central message brokering cluster. Hence you need permissions and a manageable way to assign these in a large organization.

The plan will be to support unix-like permissions on a per-topic level.

Authorization will be done in the "business logic" layer in Kafka (aka KafkaApis). The API can be something like

```
PermissionManager.isPermitted(Subject subject, InetAddress ip, Permissions permission, String resource)
```

For example doing a produce request you would likely check something like the following:

```
PermissionManager.isPermitted(session.subject(), session.peerIpAddress(), Permissions.WRITE, topicName)
```

This check will obviously have to be quite quick as it will be done on every request so the necessary metadata will need to be cached.

The subject is basically the "user name" or identify of the person trying to take some action. This will be established via whatever authentication mechanism. The action is basically a list of things you may be permitted to do (e.g. read, write, etc).

The IP address of the originating connection is is passed as it may be useful in certain authorization situations (whitelisting, or being more generous when the request originates on the loopback address, e.g.)

The PermissionManager will both check whether access was permitted and also log the attempt for audit purposes.

The resource will generally be based on the topic name but there could be other resources we want to secure so we can just treat it as an arbitrary string.

I could imagine the following permissions:

READ - Permission to fetch data from the topic
WRITE - Permission to publish data to the topic
DELETE - Permission to delete the topic
CREATE - Permission to create the topic
CONFIGURE - Permission to change the configuration for the topic
DESCRIBE - Permission to fetch metadata on the topic
REPLICATE - Permission to participate as a replica (i.e. issue a fetch request with a non-negative node id). This is different from READ in that it has implications for when a write request is committed.

Permission are not hierarchical since topics are not hierarchical. So a user will have a default value for these (a kind of umask) as well as a potential override on a per-topic basis. Note that CREATE and DESCRIBE permission primarily makes sense at the default level.

Implementing the PermissionManager

The above just gives a high-level API for checking if a particular user is allowed to do a particular thing. How permissions are stored, and how users are grouped together is going to need to be pluggable.

There are several scenarios we have considered:

- Some users may want to pick up and run Kafka without much in the way of external dependencies. These users will want a simple way to maintain permissions that works well out of the box.
- Hortonworks and Cloudera each have separate nascent attempts at securing the larger Hadoop ecosystem across multiple services. As these mature the best way to integrate into the larger ecosystem for their users will be to use either Sentry (Cloudera) or Argus (Hortonworks) depending on the Hadoop distribution the particular organization has.
- Large organizations often have very particular ways of managing security, auditing access, or implementing groups. There are various theories on the best way to manage the assignment of permissions to users (i.e. via roles, groups, acls, etc.).

Unfortunately, there is no single implementation that can satisfy all these cases. Instead we can make the PermissionsManager interface pluggable at run time so that users can specify their implementation in config.

We will try to provide something simple out of the box.

Administrators may disable authentication in configuration (giving an "audit-only" mode).

Deriving a Principal Name from Authentication Credentials

If we are to make the authorization library independent of the authentication mechanism, then we need to map each mechanism's credentials to the principal abstraction to be used in the authorization API. LinkedIn security proposes the following:

The principal is just a user name (i.e. a `String`).

When the client authenticates using SSL, the user name will be the first element in the Subject Alternate Name field of the client certificate.

When the client authenticates using Kerberos, the user name will be the fully-qualified Kerberos principal name. Admins can modify this through configuration using the standard Kerberos auth_to_local mechanism (cf. here).

When the client does not authenticate, the user name will be "nobody".

Auditing

All authentication operations will be logged to file by the Kafka code (i.e. this will not be pluggable). The implementation should use a dedicated logger so as to 1) segregate security logging & 2) support keeping the audit log in a separate (presumably secured) location.

Encryption

For performance reasons, we propose making encryption optional. When using Kerberos (via SASL & GSS-API), there are explicit parameters through which clients can signal their interest in encryption (similarly for SSL).

Sequencing

Here is a proposed sequence of work

Phase 1: Prep

- Add session as communication mechanism between socket server and kafka api layer.
- Add SSL port to metadata request

Phase 2: Authentication

- Allow disabling sendfile for reads that need encryption or other integrity checks added
- Implement SSL
- Implement SASL

Phase 3: Authorization

- Implement PermissionManager interface and implement the "out of the box" implementation.

Open Questions

Do we need to separately model hosts? i.e. in addition to user do we need to pass into the authorization layer information about what host the access is coming from.

Likely we need a way to specify the minimum encryption/integrity level of a client that is allowed to read data. Likely we should define something generic like NONE < INTEGRITY < ENCRYPTED and allow the user to set a minimum level for each topic so you can guarantee a particular data stream never goes in the clear.

Out-of-scope Features

On disk and per-field encryption

This is very important and something that can be facilitated within the wire protocol. It requires an additional map data structure for the "encrypted [data encryption key]". With this map (either in your object or in the wire protocol) you can store the dynamically generated symmetric key (for each message) and then encrypt the data using that dynamically generated key. You then encrypt the encryption key using each public key for whom is expected to be able to decrypt the encryption key to then decrypt the message. For each public key encrypted symmetric key (which is now the "encrypted [data encryption key]" along with which public key it was encrypted with for (so a map of [publicKey] = encryptedDataEncryptionKey) as a chain. Other patterns can be implemented but this is a pretty standard digital enveloping [0] pattern with only 1 field added. Other patterns should be able to use that field to-do their implementation too.

Non-repudiation and long term non-repudiation

Non-repudiation is proving data hasn't changed. This is often (if not always) done with x509 public certificates (chained to a certificate authority).
Long term non-repudiation is what happens when the certificates of the certificate authority are expired (or revoked) and everything ever signed (ever) with that certificate's public key then becomes "no longer provable as ever being authentic". That is where RFC3126 [1] and RFC3161 [2] come in (or worm drives [hardware], etc).
For either (or both) of these it is an operation of the encryptor to sign/hash the data (with or without third party trusted timestap of the signing event) and encrypt that with their own private key and distribute the results (before and after encrypting if required) along with their public key. This structure is a bit more complex but feasible, it is a map of digital signature formats and the chain of dig sig attestations. The map's key being the method (i.e. CRC32, PKCS7 [3], XmlDigSig [4]) and then a list of map where that key is "purpose" of signature (what your attesting too). As a sibling field to the list another field for "the attester" as bytes (e.g. their PKCS12 [5] for the map of PKCS7 signatures).
[0] http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/what-is-a-digital-envelope.htm
[1] http://tools.ietf.org/html/rfc3126
[2] http://tools.ietf.org/html/rfc3161
[3] http://www.emc.com/emc-plus/rsa-labs/standards-initiatives/pkcs-7-cryptographic-message-syntax-standar.htm
[4] http://en.wikipedia.org/wiki/XML_Signature
[5] http://en.wikipedia.org/wiki/PKCS_12