KIP-317: Add end-to-end data encryption functionality to Apache Kafka

Status

Current state: *Under Discussion*

Discussion thread:

Previous discussion: https://www.mail-archive.com/dev@kafka.apache.org/msg88969.html

JIRA:

Released:

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Many people using Kafka have a need to encrypt the data that Kafka persists to disk, especially when personal data of customers is involved. After the GDPR became effective in May 2018 discussions around data security have been ever-present, and while the GDPR does not require encryption it sure does recommend it. Looking at the financial sector there are a few even stricter regulations that move beyond a simple recommendation and request at-rest encryption "unless technically not feasible". Discussions around this with large corporate compliance departments can become quite heated and/or tedious.

Kafka does not currently offer functionality to accomplish this task, so users who need this feature are usually pointed towards technologies like LUKS or dm-crypt to accomplish this. In combination with on-the-wire encryption via TLS, data is secured at all times. However, there are a few drawbacks to this approach:

- TLS encryption requires the broker to decrypt and encrypt every message which prohibits it from using zero-copy-transfer and causes a somewhat larger overhead
- Volume encryption works well as a safeguard against lost disks, but less so for people who already have access to the system (rogue admin problem)
- With volume encryption, it is not readily possible to encrypt only specific topics or encrypt topics with unique keys

I think it would be beneficial for Kafka to implement functionality providing end to end encryption of messages - without the broker being able to decrypt messages in the middle. This is similar in scope to HDFS's transparent data encryption feature which allows specifying directories as encryption zones, which causes HDFS clients to transparently encrypt on write and decrypt on read all access to those directories. For Kafka the equivalent idea would be to enable users to specify a topic as encrypted which would cause producers to encrypt all data written to and consumers to decrypt all data read from that topic.

Originally I intended this to be an all-encompassing KIP for the encryption functionality, however, I now think that this may have been too ambitious and would like to stagger the functionality a little bit.

Aside from the actual crypto-implementation, the main issue that will need to be solved for this is where to obtain keys, when to roll keys, and synchronization of configuration between client and brokers I think. Since there are a lot of possible ways to provide crypto keys the implementation will be designed to be pluggable so that users have the option of implementing their own key retrieval processes to hook into an existing key infrastructure.

Some examples of these solutions are:

- Vault
- Consul
- Amazon KMS
- CloudHSM
- KeyWhiz

Phase 1 (this KIP)

In this phase, I'll add the base functionality to encrypt messages on the producer side and decrypt them on the consumer. All necessary configuration will be added directly to the clients. Key management will be pluggable, but in this phase only a basic implementation using local Keystores will be provided. This means that it will be the duty of the end user to roll out synchronized keystores to the producers and consumers

Phase 2 (KIP not created yet)

This phase will concentrate on server-side configuration of encryption. Topic settings will be added that allow the specification of encryption settings that consumers and producers should use. Producers and Consumers will be enabled to fetch these settings and use them for encryption without the end-user having to configure anything in addition.

Brokers will be extended with pluggable Key Managers that will allow for automatic key rotation later on. A basic, keystore based implementation will be created.

Phase 3 (KIP not created yet)

Phase 3 will build on the foundations laid in the previous two phases and create a proper implementation of transparent end-to-end encryption with envelope encryption for keys

Proposed Changes

Producer Changes

Configuration

The following new configuration settings will be added to the Producer. This will seem like a very short list, as it is mostly just two classes.

Since the entire process is designed to be pluggable, all actual config is part of the actual implementations of these classes. Please see section xxx for details on the implementations that are part of this KIP.

encryption.keyprovider.class

Class implementing the KeyProvider interface that offers methods to retrieve an actual key by reference. This is a mandatory configuration if encryption is to be used.

encryption.keymanager.class

A class providing an implementation of the KeyManager interface. The purpose of this class is to determine which key should be used to encrypt a given record. A common scenario would be to use a key per topic or partition. But this could also be higher-level functions like using customer-specific keys depending on data inside the record.

It is not mandatory to provide this setting, if a KeyManager is not specified then records will not be encrypted, unless a key is explicitly specified.

encryption.algorithms

A whitelist of acceptable encryption algorithms to use.

encryption.encrypt.keys

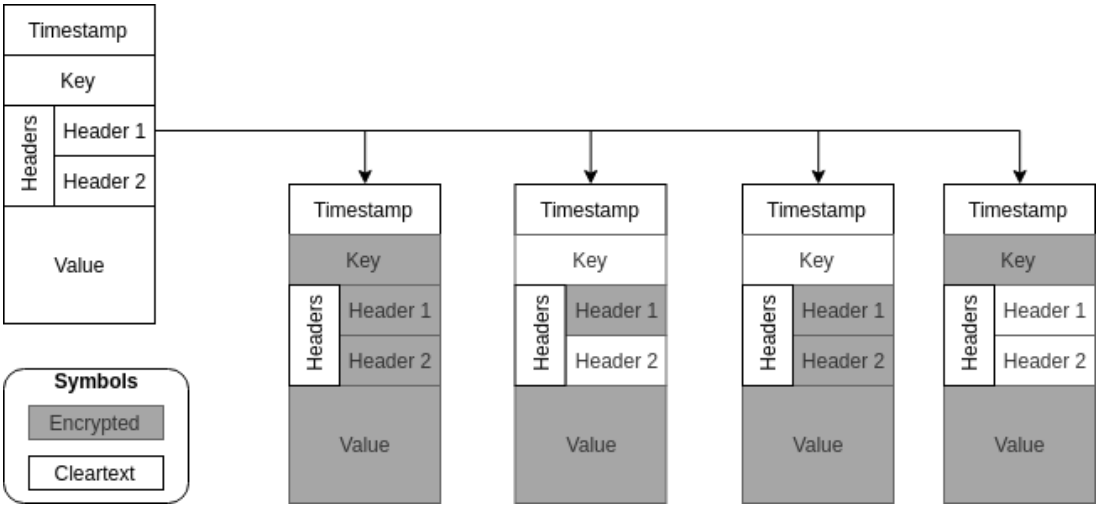
Should the keys of records be encrypted?

Will default to true.

encryption.encrypt.headers

A pattern that will match header field names that should be encrypted. Will default to .* to encrypt all headers.

These configs allow for various scenarios of which data is encrypted within a record and which isn't, as shown in the figure below.



What this would not allow is to programmatically change between these different configurations. For example to explicitly specify that a header field is to be encrypted when adding this header, not sure if we need this functionality (see section on items under discussion).

Additionally, any setting prefixed with *encryption.keyprovider.* will be passed through to the instantiated KeyProvider for configuration. This will allow creating a more useful implementation that obtains keys from remote vaults like CloudHSM, Keyvault, etc.

The API will not provide any means for the users to actually obtain the key itself, but simply allow identifying a key that is to be used for encryption. This is however more security by obscurity than any actual security as it would be easy to grab the key from the clients JVM with a debugger or run changed code to return the key itself.

API Changes

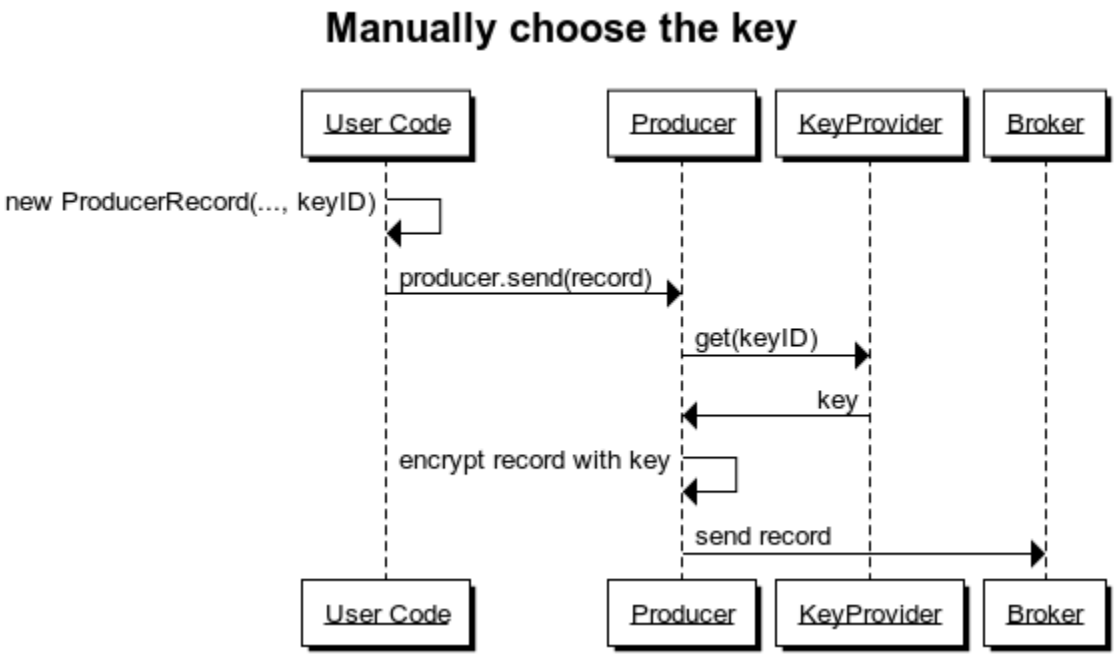
Encrypting messages can be done in two different ways, either implicitly by configuring a KeyManager, or explicitly by referencing a key when creating the ProducerRecord.

To pass a KeyReference to a ProducerRecord, the current constructors will be extended. The downside of this is, that the number of available Constructors will significantly increase (double in fact).

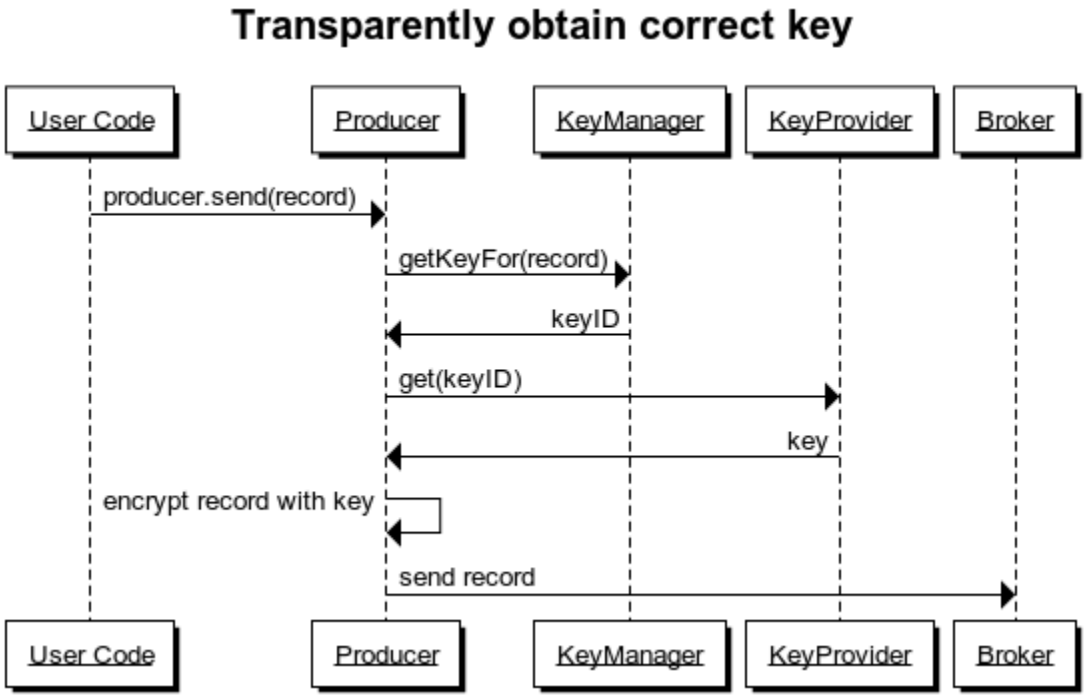
An example might look as follows:

Creating a ProducerRecord with a KeyReference

```
KeyReference encryptionKey = new DefaultKeyReference("key-2020");
ProducerRecord<String, String> data = new ProducerRecord("topic", "key", "value", encryptionKey);
```



When using a KeyManager instance, the user code is unchanged, as encryption is handled transparently in the background. The data flow within the producer is shown in the following figure.



Implementation

The producer will encrypt records based on the settings explained after partitioner and all interceptors have run, right before the record is added to a batch for sending.

This will allow spreading encryption effort out a little over time, batch.size and linger.ms permitting when compared to encrypting an entire batch, for which we'd have to wait for the batch to be complete before starting encryption.

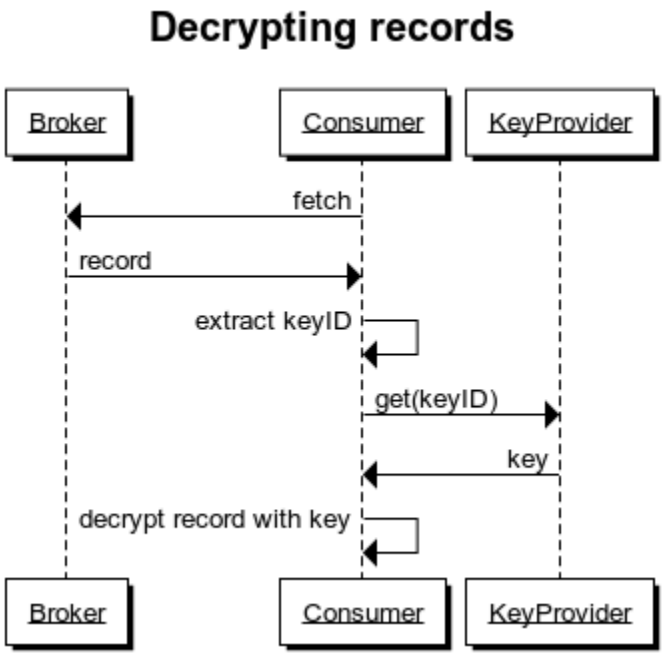
After encrypting the payload, the key reference for the key that was used and information on which fields were encrypted need to be added to the record.

Consumer Changes

The consumer will be extended to understand a subset of the configuration options that were explained in the producer section:

- encryption.keyprovider.class

When receiving an encrypted record, the consumer will retrieve the key reference from the record and use this to obtain a copy of the actual key from its configured KeyProvider.



New Interfaces

KeyProvider

The KeyProvider is instantiated by the Kafka clients and used to retrieve the appropriate key to encrypt & decrypt messages.

Keys are referenced by a KeyReference object, which the provider will use to determine the correct underlying key.

KeyProviders will support two main methods for getting a key:

getKey(KeyReference key) - obtain an exact version of a key, this is mostly used by the Consumer to get a key for decrypting records

getCurrentKey(KeyReference key) - get the current version for a key, this is to accommodate rollover functionality for keys in later versions. As the KeyManager should not be concerned with key versions it will simply return a reference without a version and leave it to the KeyProvider implementation to return the correct version of that key.

KeyManager

The KeyManager will determine which key is appropriate to encrypt a given record.

It will be passed the ProducerRecord object and be able to use any of the fields within it for this determination.

KeyReference

A very simple interface, similar to KafkaPrincipal, which is used to refer to keys. The reference implementation will simply contain a String and a version, but this can be extended as necessary by users to accommodate proprietary key scenarios that exist for corporate customers.

Proposed Public Interface Change

Binary Protocol

I've left this section empty for now, as I would like to gather feedback on the general principle first, before hammering out the details.

Consumer & Producer Config

The consumer config will receive the following new properties:

Option Name	Description
encryption.keyprovider-class	The class to use as KeyProvider.
encryption.keyprovider.	Prefix to symbolize KeyProvider config. Anything with this prefix will be passed through to the KeyProvider upon initialization.

The producer config will receive the following new properties:

Option Name	Description
encryption.keymanager-class	The class to use as KeyProvider.
encryption.keymanager.	Prefix to symbolize KeyProvider config. Anything with this prefix will be passed through to the KeyProvider upon initialization.
encryption.keyprovider-class	The class to use as KeyProvider.
encryption.keyprovider.	Prefix to symbolize KeyProvider config. Anything with this prefix will be passed through to the KeyProvider upon initialization.

Initial KeyProvider & KeyManager implementations

As part of this KIP two implementations will be provided, one KeyManager and one KeyProvider. Both will be fairly simple, but fully usable.

Full details on these to follow.

KeyProvider

The Keyprovider will read keys from a local file (optionally encrypted) file.

KeyManager

The KeyManager will be based on topic names and allow configuring keys for patterns of topic names, a default key if no pattern matches and a pattern for unencrypted topics.

Discussion points

As mentioned in the introduction, please keep the discussion on the mailing list instead of the wiki, however, I felt that it would be beneficial to track points that are still in need of clarification here, as I suspect discussion might be somewhat extensive around this KIP.

Encryption mechanism

In order for this implementation to interfere with Kafka internals as little as possible, I have opted for the simplest possible way of going about this, which is to encrypt the individual fields of a record individually.

This gives a large degree of flexibility in what to encrypt, we could potentially even use different keys for header fields and record key and value, to allow intermediate jobs access to routing information, but not the message content.

It comes at the cost of performance and message size however, encrypting the entire message, or even an entire batch will most probably be a lot faster than encrypting individual fields. We'd need to generate a new IV for every field for starters ..

Plus, we will have to pad every field to the required block size for a chosen cipher.

For example a record with 5 headers of 10 bytes, a 100 byte key and a 300 byte value would incur the following overhead for a 16 byte block size (headers are counted double, 5 bytes for field name, 5 for content):

Headers: 10 * (16 - 10) = 60 bytes

Key: (16 * 7) - 100 = 12 bytes

Value: (16 * 19) - 300 = 4 bytes

Total overhead: 76 bytes

Total overhead when encrypting entire record: 512 - (10 * 10 + 100 + 300) = 12 bytes

This is more of an illustration than an accurate calculation of course, but the principle stands.

What to encrypt?

Records don't just contain the value but can (or will) also have:

- Headers
- Timestamp
- Keys

Depending on the method of encryption that we choose for this KIP it might be possible to exclude some of these from encryption, which might be useful for compaction, tracing tools that relay on header values, and similar use cases.

Timestamps

The timestamp is a bit of a special case as this needs to be a valid timestamp and is used by the brokers for retention, we probably shouldn't fiddle with that too much.

Keys

For normal operations, I would say that we can safely encrypt the key as well, as we'd assign the partition in the producer before encryption. (Side note: is this a potential security risk because we can deduct information on the cleartext from this?)

But for compacted topics, we need the key to stay the same, which it wouldn't, even when encrypted with the same key, let alone after key rollover.

Headers

Headers are not used for anything internally, so should be safe to encrypt. However, they are outside of the actual message by design, as this makes them much easier to use for routing and tracing purposes. In a trusted environment there might be value in having some sort of unencrypted id in a header field that can be used to reference this message across topics or systems.

I propose that we leave the timestamp unencrypted and by default encrypt key and header fields. But we give users the option of leaving the key and/or individual header fields unencrypted if necessary.

Allow encrypting compressed content?

As was pointed out on the mailing list, compression is a difficult topic in the context of encryption. Encrypted content doesn't compress well as it is random by design, but encrypting compressed content has security implications that may weaken the security of the encrypted content (see CRIME & BREACH). TLS has afaik removed compression in version 1.3 for this very reason (reference missing, I couldn't find any tbh).

I propose to take an opinionated approach here and disable the parallel use of compression and encryption. But I could absolutely understand if people have other views on this and I am happy to discuss, especially in light of Kafka arguably already supporting this when compression is enabled for data transmitted to an SSL listener.

Support multiple different KeyProviders in a single Producer / Consumer

I am doubtful, whether we need to support more than one KeyProvider for a Client at this point.

I am leaning towards no, should a requirement to do this exist for a user, there is always the option of just having additional clients to accommodate this.

Whitelist unencrypted topics?

Messages might accidentally be sent unencrypted if the user simply forgets to specify a default encryption key in the producer config. To avoid this we could add a setting "encryption.unencrypted.topics" or similar that whitelists topics that are ok to receive cleartext data (or the other way around, blacklist topics that cannot receive cleartext data).

Alternatively, we might consider providing a set of options to specify default keys per topic(-pattern), but this might be taking it a step too far I think.

What do you think, should we include this?

Configuration vs Programmatic control

For the current iteration of this KIP, I have mostly chosen to provide configuration options to control the behavior of encryption functionality. The one exception being the possibility to specify a key when creating a ProducerRecord.

This has the benefit of working with 3rd party tools that just use Kafka client libraries without these tools having to change their code - at the cost of some flexibility.

The alternative / additional option would be to allow programmatic control. For example:

Instead of providing a setting that specifies a whitelist of header fields that should be unencrypted we extend the RecordHeader class to include information on whether to encrypt this header and with which key.

Should we make the actual encryption pluggable?

Or is that taking it a step too far?

Compatibility, Deprecation, and Migration Plan

Test Plan

TBD

Rejected Alternatives

Encrypting RecordBatches

It would probably be much more efficient to encrypt the payload of an entire RecordBatch, which is many records appended to each other, instead of encrypting every single message individually, however, there are a few drawbacks to encrypting a full batch:

- The broker cannot perform any validation on the record batch before appending it to its log.
- The broker cannot see control messages contained in the batch
- We'd lose the ability to use different keys for different records within a batch
- Compaction essentially becomes impossible with batches like this

We may choose to revisit Batch-Encryption at a later time, but at the moment per-record encryption seems to be the better choice. In case this is added at a later time it should become a second mode of operation instead of replacing the existing functionality.

Coding the encryption without a library

I considered not using a higher-level encryption library for the actual encryption and relying on core Java functionality. In the interest of security I decided against this and with Google Tink chose an opinionated framework that will try and keep the user (clueless me) as far away from potential mistakes as possible.