Encryption Library in

Hyperledger Fabric v1.1

(JIRA FAB - 5727)

# Motivation

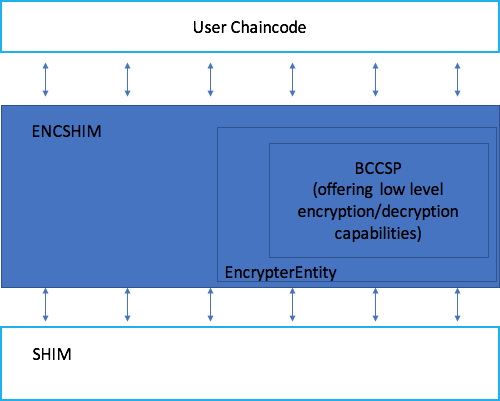
In Hyperledger Fabric, updates to a channel’s ledger are visible to all the parties who have read-access to that channel’s transactions. That is, as soon as a proposal is processed by an endorser, the respective endorsement (and included state updates) is included by the client in a transaction, submitted to the ordering service. It is henceforth clear, that the transaction itself, and included state updates will be visible to all the ordering nodes, as well as to the application network of the channel the transaction was included in.

However, the data a chaincode processes (i.e., proposal input), and/or state updates upon a chaincode’s invocation can be strongly bound to a specific business logic and are therefore required to remain confidential.

One step towards making this easier, is the offering encryption capabilities at the chaincode level, using client-side provided keys.

# Architecture

Encryption capabilities on the chaincode are facilitated by ENCSHIM, i.e., an encryption layer on top of chaincode SHIM, whose job is to encrypt the data before updating a state value (on a PUT) or decrypt encrypted parts of the state after fetching them by the shim (on a GET). As depicted in Figure 1, to perform the encryption/decryption operations, ENCSHIM is loaded with a (symmetric) key by the chaincode (represented by EncryptionEntity), and performs calls to BCCSP, the blockchain crypto service provider of Fabric that performs low level encryption/decryption operations.



More specifically ENCSHIM operates on the basis of a cryptographic material called “Entity”. An entity can represent an encryption key-pair, or a symmetric encryption key, or a signature key-pair and is referenced using an identifier. The interface of Entity is as described below:

// Entity is the basic interface for all crypto entities

// that are used by the library to obtain cc-level encryption

**type** Entity **interface** {

// ID returns an identifier for the entity;

// the identifier can be set arbitrarily by

// the entity's constructor in a manner that

// is relevant for its usage at the cc-level

ID() string

// Equals compares this entity with the supplied

// one and returns a boolean that is true if the

// two entities are identical. This includes any

// and all key material that the entity uses

Equals(Entity) bool

// Public returns the public version of this entity

// in case asymmetric cryptography is used. If not,

// Public returns itself

Public() (Entity, error)

}

EncrypterEntity is a special case of entity that performs encryption/decryption operations on a set of bytes (message). As such, it extends Entity with an Encrypter interface as specified below:

// Encrypter is an interface that provides basic encrypt/decrypt capabilities

**type** Encrypter **interface** {

// Encrypt returns the ciphertext for the supplied plaintext message

Encrypt(plaintext []byte) (ciphertext []byte, err error)

// Decrypt returns the plaintext for the supplied ciphertext message

Decrypt(ciphertext []byte) (plaintext []byte, err error)

}

// Encrypter entity is an entity which is capable of performing encryption

**type** EncrypterEntity **interface** {

Entity

Encrypter

}

ENCSHIM interface is depicted below. Each time a key is to be put to the ledger, ENCSHIM is loaded with the proper cryptographic material (EncrypterEntity), using “With” function. After that it is ready to accept GetState(...) and PutState(...) requests.

// EncShim is an interface that supports and facilitates

// chaincode-level encryption. It's expected to be used as follows

//

// shimInstance.With(entity).GetState(key)

//

// or

//

// shimInstance.With(entity).PutState(key, value)

//

// where GetState and PutState retrieve data from a backing key-value

// store (e.g. the ledger through the chaincode shim) and perform

// on-the-fly encryption and decryption appropriately according to

// the entity that is specified through the With function

**type** EncShim **interface** {

// GetState returns the value associated to the supplied key

// after decryption using the entity specified through a

// previous call to With

GetState(key string) ([]byte, error)

// PutState encrypts the specified value using the entity

// specified through a previous call to With and associates

// it to the supplied key string

PutState(key string, value []byte) error

// With supplies the Encrypter Entity that shall later perform

// encryption/decryption call required by calls to GetState or PutState

With(e entities.Encrypter) EncShim

}

Upon a GetState request, ENCSHIM would trigget GetState(...) from the SHIM, and then uses the EncrypterEntity.Decrypt() function to decrypt the received’s key-value, value. Upon PutState, ENCSHIM would use EncrypterEntity.Encrypt to encrypt the received entry’s value, and invoke SHIM’s PutState using the resulting ciphertext.

Encryption and decryption takes place in an almost transparent way to the chaincode developer, as the latter only needs to retrieve the

Information to setup an EncryptionEntityImpl (struct that implements EncryptionEntity) can be passed by the client to the chaincode via proposal transient data. An example of how transient data can be used by the chaincode is provided here: /fabric/core/chaincode/shim/ext/encshim/eccc\_example.go