Identity Hub Data Encryption and Sharing

Motivation

Identity Hubs are designed to encrypt data in transit and at rest. Data in transit is encrypted with the hub's DID keys. Data at rest should be encrypted with keys under the control of the user.

The goal of this proposal is to encrypt all data stored in identity Hubs such that even the operator cannot access the data without user consent. At the same time, users must be able to share the data in their hub with 3rd parties.

This feature will help increase trust in identity hubs and help differentiate decentralized identity from existing identity solutions.

Introduction

We propose a design that allows users to store their objects encrypted in the hub. The design allows for users to share data with peers. Peers can also store encrypted objects in a user's hub.

The design will support the CRDT model meaning we need to encrypt the whole history of operations to an object.

Requirements

Requirement/assumptions	**Level**
Unless explicitly granted access, the identity hub operator cannot access user data.	High
User can selectively decide which object to encrypt in the hub.	High
The encryption scheme can be applied in the user agent transparent to user.	High
Users can share individual objects or all objects of a certain object type.	High
The operations on the user agent need to be designed with performance in mind.	High
Enterprise hubs will need escrow to access data of employees that left the company.	High
When access to data is revoked, the audience loses access to any future changes.	High

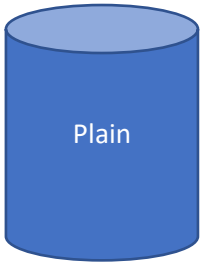
Hybrid audience scheme

First, we need to investigate how our hybrid scheme will be constructed.

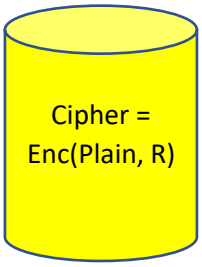
Hybrid Encryption

Plain	Data to encrypt
Cipher	Encrypted data
Aud	Public key of audience for the Plain data
Own	Private or public key of the owner of the data
R	Random key used to create cipher
Enc	The encrypt operation
Sig	The signature operation

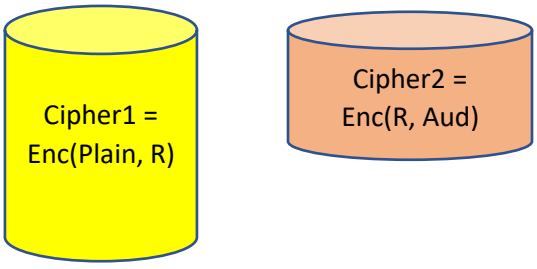
1. We take our plain data



2. Encrypt the plain data with our random key R



3. Encrypt R with the public key of audience

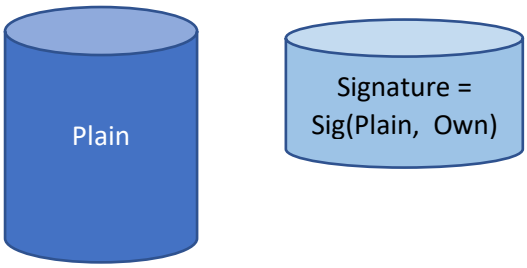


We can now add as many audiences as we want by just adding new records of encrypted R's with the public key of new audiences. The records will be stored in the Object Protections Elements.

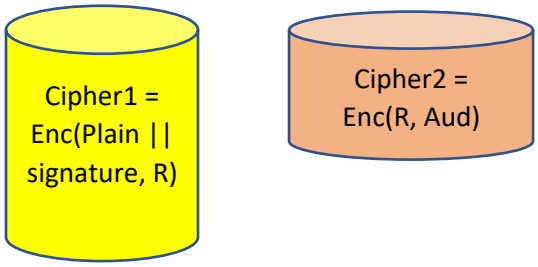
Hybrid encryption with authenticity

We want to make sure that the originator of data can always be identified. We will do this by adding a signature of the owner of the data.

1. Adding the signature on the plain data



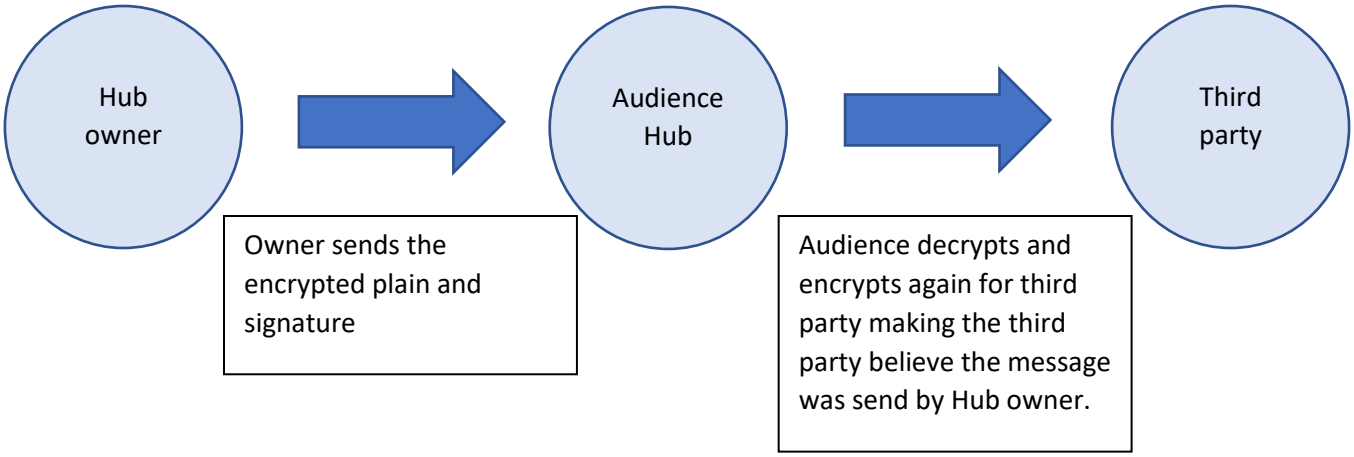
2. Next, we can encrypt Plain and signature with R and encrypt R with the public key of the audience.



The model of encrypting the signed plain data is the approach taken by many protocols including JOSE. The model has a known vulnerability called surreptitious forwarding.

Surreptitious forwarding

Encrypting the plain data and signature is known to have the surreptitious forwarding problem.



Question:

Is this a problem for our scenarios? Does any recipient of the original data need to understand the originator?

Solution

Protection against surreptitious forwarding will be more expensive. We need to bind the audience into the signature. This means we need to resign the object each time we add a new audience. Because audiences cannot be shared amongst each other, this also applies we will need to have different encrypted copies, one for each audience.

An optimization will be to add an additional record to the payload.

Intended audience = $Sign_{user}$(Signature on plain || audience).

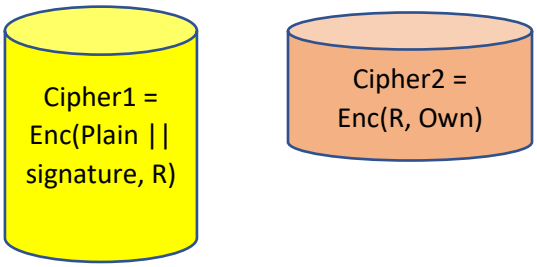
If the recipient wants to protect itself against surreptitious forwarding it needs to validate the 'intended audience' proof and check whether the recipients was intended to receive the object.

This scheme will allows us to encrypt the larger plain data only once and share it with multiple audiences. For each audience we need to create a protection element with the encrypted R and we need to add a signature on the plain data signature plus the audience.

Model for flexibly adding additional audiences

Storing a signed and encrypted object

We will need a performant way to add new audiences to an object. This can be accomplished by keeping elements of our protected object separate.



The plain data is first signed on the user agent with the owner's private key. Next, we encrypt the signed data with a random number R. This R is then encrypted with the public key of the owner so owner can always retrieve R.

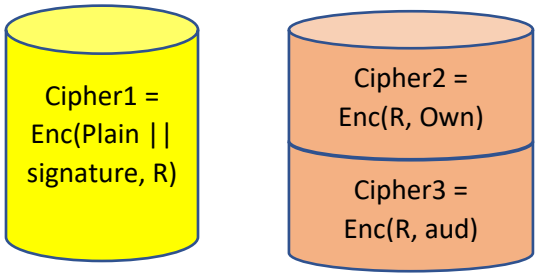
A new audience is added to read the object

A new audience (aud) requests read access to the object. The transaction involves the user agent.

1. User agent receives the read request from the audience
2. The user agent permits the audience to read the object
3. Add new audience record

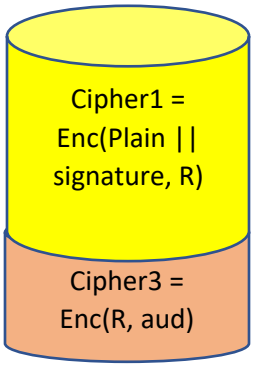
The user agent reads the audience records for the object and search for itself as the audience.

Next, the user agent will retrieve R by decrypting its audience record with it own private key. A new *audience record is now created for the new audience.*



When the actual object is transmitted to the new audience, the will reassemble the object so no other audiences appear in the object.

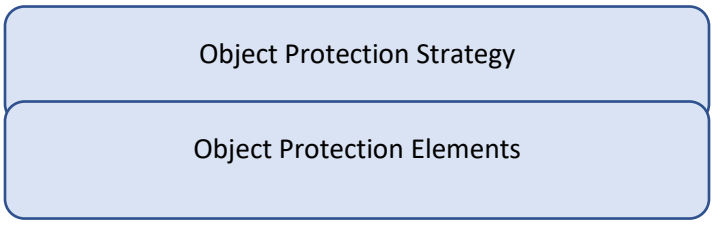
The audience will receive:



We can repeat this process for as many audiences we require.

Protection object

The Protection strategy object will store the object protection properties.



Object Protection Strategy

The object protection strategy will define the actual policy that will be applied to an object or to an object type.

The object protection policy contains the following properties:

Encryption Strategy

Define the encryption strategy to protect the confidentiality of the object. The strategy references an ID of a strategy object.

We can have many encryption strategy objects. Eventually a user could create its own strategy objects and protect certain object according to this strategy.

We propose the following default encryption strategies. The alg values are defined by https://datatracker.ietf.org/doc/rfc7518/?include_text=1.


```
{
   "id": "ea3c080e-deb8-4519-a954-58fba3f9cc3d",
   "partition_key": "did:test:hub.id:ea3c080e-deb8-4519-a954-58fba3f9cc3d/objects/partition-001",
   "name": "DefaultEncryptionEC",
   "@type": "EncryptionStrategy",
   "protocol": "jose",
   "keyEncryptionAlgorithm": {
      "alg": "ES256K"
   },
   "contentEncryptionAlgorithm": {
      "alg": "A256GCM"
   }
}

{
   "id": "93e1a42d-81dd-4722-97dd-0228220e9395",
   "partition_key": "did:test:hub.id:93e1a42d-81dd-4722-97dd-0228220e9395/objects/partition-001",
   "name": "DefaultEncryptionRSA",
   "@type": "EncryptionStrategy",
   "protocol": "jose",
   "keyEncryptionAlgorithm": {
      "alg": "RSA-OAEP",
      "length": 2048
   },
   "contentEncryptionAlgorithm": {
      "alg": "A128GCM"
   }
}
```


Objects can reference these strategy objects to define their encryption strategy.

Signature Strategy

Define the signature strategy to protect the authenticity of the object. The strategy references an ID of a strategy object.

We can have many signature strategy objects. Eventually a user could create its own strategy objects and protect certain object according to this strategy.

We propose the following default signature strategies:


```
{
   "id": "1cc2ce72-55dc-4324-b365-af3d302107cf",
```

```
  "partition_key": "did:test:hub.id:1cc2ce72-55dc-4324-b365-af3d302107cf/objects/partition-001",
  "name": "DefaultSignatureEC",
  "@type": "SignatureStrategy",
  "protocol": "jose",
  "algorithm": {
    "alg": "ES256K"
  }
}

{
  "id": "a26ed31e-a2ca-45b3-9ec7-9238beb3cca4",
  "partition_key": "did:test:hub.id:a26ed31e-a2ca-45b3-9ec7-9238beb3cca4/objects/partition-001",
  "name": "DefaultSignatureRSA",
  "@type": "SignatureStrategy",
  "protocol": "jose",
  "algorithm": {
    "alg": "RS256",
    "length": 2048
  }
}
```


Hardened Signature Strategy:

Algorithms used to add hardened authenticity to the object or type. Hardened authenticity provides protection against surreptitious forwarding.

TBD

Object Protection Elements

This part contains the encrypted audiences for the object or type (see Model for flexibly adding additional audiences).

Copies of objects?

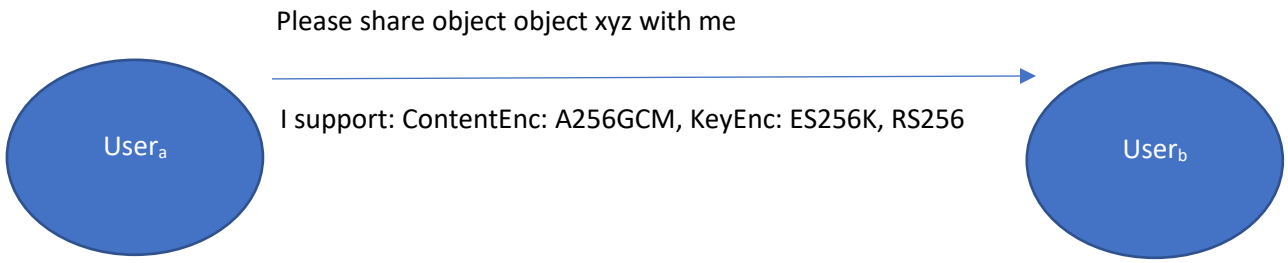
It is very desirable not to have the necessity to copy object for different audiences. The proposed scheme does not require us to copy the object for different audiences. This is under the assumptions that everybody standardizes on the same content encryption algorithm.

How can we upgrade every object to a new algorithm. We will need a negotiation protocol to do this. This needs to be defined.

Audiences using different key types

We can assume that not everybody will use the same key types for encryption. Some users will use RSA keys, other will use EC keys.

Also, this problem be solved by means of negations as explained in the following diagram.



The user agent of user$_a$ can now select one of the Key Encryption algorithms and prepare the audience for user$_b$.

Because the user agent prepares each audience protect element, we can use different algorithms for these elements depending on the requirements of the audience.

Storing of strategies

The payload of objects can be stored in JOSE. All commits on an object will be protected by the same R but require a separate signature if the signature strategy is applied. It is not necessary to store the audience with every commit, In some DB implementations commits and the initial object may be stored separately. We can store the strategies and the audiences with in the initial object.
We will add an object to reference the used strategies.

Example:

“protectionStrategies” : {

“encryption”: “93e1a42d-81dd-4722-97dd-0228220e9395”,

“signature”: “a26ed31e-a2ca-45b3-9ec7-9238beb3cca4”

}

We reference the strategy object by means of its id.

We will store the audiences as a property of the initial object. Strategies and audiences can be seen as metadata about an object. We are not really updating an object by adding an audience.

When sharing an object, we only leave the intended audience in the header. The other audiences will be stripped.

Writing data into the hub

User writes/updates data to the hub

A user updating data to the hub requires the following steps

1. The user agent reads the encrypted object from the hub. Indicate the DID used to decrypt it. If the DID corresponds to the owner of the object, all encrypted audiences will also be passed.
2. The user agent uses the private key associated with the DID to decrypt the object.
3. Update the object which results in a new transaction.
4. Generate new R to encrypt object