Secure Data Sharing Protocol

DRAFT-v0.09, July 2020

David Broman
david.broman@consecio.com
Consecio AB, Sweden

Abstract

This document gives an overview of the Secure Data Sharing (SDS) protocol (DRAFT version). The objective of the document is to give a formal definition of the security model of the protocol, without describing low-level protocol and implementation details.

1 Introduction

The Secure Data Sharing (SDS) protocol enables different entities, such a private persons, enterprises, and systems to share sensitive information in a secure manner. In particular, the system is designed for scalability, without a single central entity for coordinating access control. The system is based on a *web of trust* model, where individual users connect to each other and share data without exposing their shared private data to anyone else. The model enables end-to-end security between different users, where the data is still backed up on managed servers. Note, however, that the data are encrypted such that no-one—including employees working with the servers—can read or alternate the data without users' explicit consent.

2 Notations and Definitions

This section introduces basic notations, cryptographic operations, and the main components and keys of the security model. This section should be used as a reference when reading the rest of the document.

2.1 Basic Notation

An *item* is either a data item or an opaque item. A *data item* (that contains a possibly empty sequence of bytes) is written in lower case letters, such a, b, or m. The infix operator $\|$ concatenates the bytes of two data items, producing a new data item. For instance, $a\|b$ is the concatenation of the two data items a and b. We use the symbol $\perp$ to denote an empty data item. An *opaque item* is used as a unique identifier, within a (possibly infinite) set of identifiers. For instance, we might say that $x \in X$ is a unique identifier within the set X. A set of items is denoted using capital letter, such A or B. If the set is a multiset, it is explicitly stated within its context. Notation $\mathcal{P}(A)$ is the powerset of A.

2.2 Cryptographic Operations

The model uses secure hash functions, asymmetric encryption, symmetric encryption, and a pseudo number generator. The following list defines the algorithms used throughout the text.

- $prng(k)$ is a secure pseudorandom number generator that produces k bits of random data.
- $hash(a)$ is the secure one-way hash of data a. The actual hash algorithm is configurable. The default algorithm is 256-bit SHA-2.
- $aead_e(m, k, n, h)$ is an authenticated encryption with associated data (AEAD) that combines confidentiality, integrity, and authenticity of data. m is the plain text, k the symmetric key, n the 96-bit nonce, and h a header that is not encrypted, but has authenticity protection. The function returns a tuple (c, t) with cipher text c and an authentication tag t. See RFC 7366 for encrypt-then-MAC and RFC 5116 for using AEAD. The nonce is divided into a 32-bit[1] counter field n_c, a 10-bit[2] device field n_d, and a 54-bit[3] user identifier field n_u. The symmetric algorithm is configurable. The default algorithm is 128-bit AES using GCM mode.
- $aead_d(c, k, n, t, h)$ is the AEAD decryption function, where c is the cipher text, k the symmetric key, n the 96-bit nonce, t the authentication tag, and h a header. The function returns the plain text m or an error $\perp$.
- $hmac(k, m)$ creates a hashed message authentication (MAC) data item a of message m using the secret key k. The default HMAC uses 256-bit SHA-2.

[1]According to RFC 5116, Section 3.2.
[2]We assume that the max number of devices per user is $1\,024$.
[3]This allocation of 54-bit for the user field assumes that 22 bits are allocated for the home server identification (max $4\,194\,300$ home servers) and 32 bits are used for the unique user per home server (max $4\,294\,967\,296$ users per home server)). This means that the unique userID consists of $32 + 22 = 54$ bits.

- $keygen_s()$ generates a public/private key pair $(k^{s:pub}, k^{s:priv})$ for digital *signatures* (hence the s subscript). Both RSA and Elliptic Curves (EC) can be used, but EC is the default.

- $sign(k^{s:priv}, m)$ creates a digital signature for plain text m using the private key $k^{s:priv}$. The returned signature, denoted g, can only be verified by using the corresponding public key $k^{s:pub}$.

- $verify(k^{s:pub}, m, g)$ verifies the digital signature g for plain text message m, using the public key $k^{s:pub}$. The function returns a boolean value.

- $keygen_x()$ generates a public/private key pair $(k^{x:pub}, k^{x:priv})$ used for secure key *eXchange* (hence the x subscript). The default algorithm is Elliptic-curve Diffie–Hellman (ECDH).

- $secret(k^{x:pub}, k^{x:priv})$ generates a shared secret based on the public/private key pair $(k^{x:pub}, k^{x:priv})$.

2.3 System Abstractions

The overall system consists of users, client devices, home servers, and data objects. A user can have one or more client devices. All devices for one specific user always connects to the same home server, but different users can connect to different home servers.

A *client device* can either be *stateful* (can have a local storage) or be stateless (can be connected and authenticated, but does not have a local storage). Example of stateful clients are mobile apps and desktop applications. Example of stateless clients are web clients.

A *home server* is the connection point for a client device. The home server is responsible for storing all data objects that the user is the owner of.

A *data object*, or just an *object*, consists of general data that may be shared among users. An object always has one specific owner. An object consists of a set of fields, where each field is a mapping between a unique *label* and a field *value*. Each field can be given different access rights. For more details about objects, see Section 4.

More formally, we use the following notation for the system abstractions:

- S is the set of home server identifiers.

- U is the set of users identifiers. $u \in U$ is globally unique (54-bit globally unique number, guaranteed by the server).

- D_u is the set of devices for a user u. We write $|D_u|$ to denote the number of devices for user u. The set of stateful devices D_u^s and the set of stateless devices $D_u^{\cdot}$ are distinct, that is $D_u^s \cup D_u^{\cdot} = D_u$ and $D_u^s \cap D_u^{\cdot} = \emptyset$. We write D for the set of all client devices in the system.

- O is the set of object identifiers. An object identifier $o \in O$ is guaranteed to be globally unique by using UUID generation or using user identifies as part of the identifer string.

- $l \in L_O$ is one label of the set of labels in object O. For an object $o \in O$, we write $o.l$ to denote the field value for label l in object o.

- B is the set of arbitrary binary data. We use this set to represent data where the structure is not described in this document.

2.4 Keys and Secrets

Within the security model, K is the set of all possible keys and secrets. The main keys and secrets within the model are summarized below.

- k_u^{master} is the 256-bit master key for user u. This is the main secret that is stored on all *stateful* devices D_u^s.

- $k_d^{session}$ is the session token that is generated by the home server when a client device d is logging in to the server. The session token is stored in a session cookie.

- k_{u_1,u_2}^{shared} is the shared key between users u_1 and u_2. This is a symmetric key, with the default size 128 bits.

- $k_{u_1,u_2}^{connect}$ is the connect key that is used for secure authentication between two users u_1 and u_2. The connect key is only valid until two users have been connected. The default size of the connection key is 128 bits.

- k_o^{obj} is an object key, specific to object o. If there are label-specific object keys, denoted $k_{o,l}^{obj}$, then the label-specific object key is used for label l. The object keys are used for symmetric encryption. See AEAD encryption.
- $k_u^{s:pub}, k_u^{s:priv} \in K$ is the public/private key pair for a user u that is used for *digital signatures* (hence the s before the colon). The private key is encoded using PKCS 8 and the public key is encoded using SPKI (Simple public-key infrastructure).
- $k_{u_1}^{x:pub}, k_{u_2}^{x:priv}, k_{u_2}^{x:pub}, k_{u_1}^{x:priv} \in K$ are the public/private keys used for *key exchange* between two users u_1 and u_2. User u_1 can create the shared secret using function $secret(k_{u_2}^{x:pub}, k_{u_1}^{x:priv})$ and user u_2 generates the same secret using $secret(k_{u_1}^{x:pub}, k_{u_2}^{x:priv})$.

2.5 Trusted Computing Base and Communication Channels

The overall objective of the SDS protocol is to enable secure object exchange between client devices. That is, within the CIA triad, confidentiality and integrity of object transfers must be ensured without the need to trust any other components than the client devices themselves. This means that the *trusted computing base (TCB)* for a user—regarding confidentiality and integrity of objects—is the user's client device. As a consequence, the SDS protocol does not concern security aspects of the operating system or hardware aspects of client devices. Any other entity in the overall system cannot be trusted. That is, even home servers do not need to be trusted regarding objects' confidentiality and integrity.

However, for *availability*, the home servers are critical. That is, the home servers need to be available and working correctly for client devices to be able to transfer objects between each other. Note, however, that if an attacker pretends to be another user, or if the attacker manages to get access to the home server, it must not affect the confidentiality and integrity of objects. This means that an attacker neither can read encrypted objects stored on home servers, nor can the attacker alter an object without being discovered by other client devices.

The different entities in SCS are modeled as a graph $G = (V, E)$, where the the set of vertices V consists of home severs and client devices, and where the set of edges E represents the communication channels between entities. We define $V = S \cup D$, where $S \cap D = \emptyset$. The graph represents both unidirectional communication channels (e.g., $(d_1, d_2) \in E$), or bidirectional channels (e.g., $\{(d_1, d_2), (d_2, d_1)\} \subseteq E$.

We assume the following secure communication channels:

- The bidirectional channel $\{(d, s), (s, d)\} \subseteq V$ between a client device d and a home server s is secured using TLS v1.3. The secure channel is terminated at the home server. After authentication, a session token $k_d^{session}$ is used when a client device reconnects to the home server.
- Each home server pair $(s_1, s_2) \in S$ can estamlish secure communication using TLS 1.3. That is, a home server s_1 can authenticate and connect to another server s_2 using a secure REST API over `https`. The authentication mechanism between home servers is not described in this document and it does not affect the overall confidentiality and integrity objectives of SDS.
- A client device $d_1 \in D$ can transfer secret plaintext information to another device $d_2 \in D$ using QR codes. Device d_1 generates a QR code with the secret plaintext data and user d_2 scans the data. In the SDS protocol, we assume that this is a trusted channel, which means that the QR code cannot be read by an attacker physically (scanning the QR code using shoulder surfing) or virtually (that the QR code is sent digitally over an insecure channel).

All other communication between client devices and home servers should be considered as insecure.

3 Registration

The protocol is using end-to-end encryption, which means that the client devices encrypt all data, but the data can still be shared between users. The data is stored on the home servers, but even if the home server is compromised, the data cannot be decrypted without the client keys.

Registration in end-to-end mode is only possible for stateful devices because the master secret must be stored by the client. The procedure is as follows:

1. The client device generates a master secret k^{master} by calling $prng(256)$. Note that the user device does not yet have a user ID.
2. The client generates a salt value $m_{salt} = prng(64)$. A computer generated password key p is generated using $hash(k^{master} \| m_{salt})$. The salt and password key are sent to the home server.

3. The server stores the password key and the salt.

4. The client is registered but not logged in.

Note that further verification of the user is possible, for instance to use SMS or email verification, to mitigate that an attacker overloads the server by creating massive amount of user accounts. However, these implementation details are outside the scope of this document.

4 Objects

An object identifier $o \in O$ represents a data object, but it does not state exactly what the content of the object is. In one specific point in physical (wall-clock) time, the actual data content of an object is allowed to be different for different clients and servers. The main reason for this is that clients can be in offline mode. However, the *eventual* data content is always consistently converging to one specific value. Each object always has *one* unique owner, and the consistent data of an object is always stored in the owner's home server.

4.1 Events

An object's data content (field values for certain labels, written as $o.l$, see Section 2.3) and its meta information (for instance, the owner of an object) are represented altogether as a sequence of *events*. For an object $o \in O$ the sequence of events is denoted as a set $E_o = \{e_1, e_2, e_5, e_7\}$, where the subscript denotes the *event number*. Notation $\lceil E_o \rceil$ denotes the largest event number of the object, whereas $|E_o|$ denotes the number of events. For instance, $\lceil \{e_1, e_2, e_5, e_7\} \rceil = 7$, whereas $|\{e_1, e_2, e_5, e_7\}| = 4$.

Note that events are ordered using event numbers, but under certain conditions, it is allowed that certain event numbers are missing. For instance, in the above example, there is no event with event number 6.

There are six *types of events*: *owner* (*o*), *reset* (*r*), *access* (*a*), *patch* (*p*), *delete* (*d*), and *transaction* (*t*). For a specific event e, we use the superscript to denote the type of the event. For instance, e^t_4 is a transaction event with event number 4, whereas e^p_7 is a patch event with event number 7.

Each event of a specific event type has certain number of *event values* associated to the event. We use the notation $e.\mathtt{x}$ to denote an event value with name $\mathtt{x}$ for event e. The following list describes all event values for the six different event types.

- **Event type: *owner* (*o*).** An owner event e^o states that an object has a new owner. The event values for owner events are as follows:
  - Event value: $e^o.\mathtt{userId}$. This event value is the user identifier $u \in U$ for the user who is the new owner of the object, from this event number and forward, until another *owner* event declares another owner.
  - Event value: $e^o.\mathtt{acount}$. An access counter number counting the number of *owner*, *reset*, and *access* events in a specific object. The first number is 1, and then incremented by one. The counter number is used when computing the nonce for an *access* event and as a way to guarantee the total order of access changes for an object. If the concurrent changes are conflicting, the home server rejects such access change. For instance, if two devices are offline and both make changes to access rights of the object, then the home server decides which change is accepted, and which is rejected after the two devices are online again. Note that the access count number is unique to an object, not to a user.
  - Event value: $e^o.\mathtt{signature}$. The digital signature is used to verify that the change of ownership is signed by the previous owner. Assume that the current event is e^o_k such that the new owner is $u_n = e^o_k.\mathtt{userId}$. The previous owner is $u_p = e^o_m.\mathtt{userId}$ where m is the largest event number representing an *owner* event, where $m \leq k$. Note that if $m = k$ then $u_n = u_p$, which means that this is the first owner of an object. Assume also that o_c is the current object identifier. The signature is computed as: $e^o_k.\mathtt{signature} = sign(k^{s:priv}_{u_p}, u_p \| u_n \| c_a \| o_c)$, where $c_a = e^o.\mathtt{acount}$. Hence, if it is the first owner of an object, the object is self signed and $e^o.\mathtt{acount} = 1$.
- **Event type: *reset* (*r*).** A reset event e^r removes all access rights for the whole object or for a specific object field. A reset event is always followed by an $access$ event that adds new access rights. This means that a new k^{obj}_o or $k^{obj}_{o,l}$ key needs to be generated, which is done as part of the *access* event that follows. The event values for *reset* events are as follows:
  - Event value: $e^r.\mathtt{userId}$. This event value specifies the user who performs the reset. Such a user must have `owner` or `admin` access rights to the object or field.

– Event value: e^r.label. The event value e^r.label contains the label l of the field for which the access is removed (field-level access). If this event value is empty $\perp$ then the reset of accesses is for object-level access.

– Event value: e^r.acount. An access counter number counting the number of *owner*, *reset*, and *access* events in a specific object. For more details, see the access counter event value description for the *owner* event.

– Event value: e^r.signature. In end-to-end mode, the digital signature is used to verify who have made the reset. The signature is computed as: $e^o_r.\texttt{signature} = sign(k_u^{s:priv}, u\|l\|c_a\|o_c)$, where $u = e^r.\texttt{userId}$, $l = e^r.\texttt{label}$, $c_a = e^r.\texttt{acount}$, and where o_c is the current object identifier.

- **Event type: *access* (*a*).** An access event e^a adds access rights for the whole object or for specific object fields.

  At the object level, there are four possible access levels: owner access (owner), admin access (admin), read & create access (rc), and read access (r). All except the owner access is handled by *access* events (in combination with *reset* events). An object can only have one owner, and it is assigned by the *owner* event. The access rights are defined to have monotonically increasing privileges r $<$ rc $<$ admin $<$ owner. For instance, the admin access level gives all access rights that are available for the rc access.

  It is possible to give specific access rights on separate fields. At the field level, there are three possible access levels: admin access (admin), read & write access (rw), and read access (r). If a user has object-level access and field-level access for a specific object, then the field level access can give higher access level, not lower. For instance, if a user has r access at the object level, and field access admin for a field with label l, then the user can read all fields in the whole object. Moreover, the user can also delete, change access rights for, and write to the field with label l. The rc access at the object-level makes it possible for a user to create a new field and then give himself arbitrary field-level access to the new field.

  The following table summarizes different actions that can be taken on objects, which events that are used, and what access levels that are required for taking specific actions.

		Object-level access				**Field-level access**		
Action	**Event**	owner	admin	rc	r	admin	rw	r
Change object owner	*owner*	x						
Create new field	*patch*	x	x	x				
Delete field	*delete*	x	x			x		
Change access rights	*access/reset*	x	x			x		
Write to field	*patch*	x	x			x	x	
Read from field	n/a	x	x	x	x	x	x	x

  The event values for access events are as follows:

  – Event value: e^a.label. The event value e^a.label contains the label l of the field that is given additional access rights (field-level access). If this event value is empty $\perp$, then the new access rights are given at the object level.

  – Event value: e^a.userId. The user identifier for the user who changes the access rights.

  – Event value: e^a.deviceNo. The unique device number for user e^a.userId.

  – Event value: e^a.acount. An access counter number counting the number of *owner*, *reset*, and *access* events in a specific object. For more details, see the access counter event value description for the *owner* event.

  – Event value: e^a.signature. In end-to-end mode, this digital signature makes sure that a user with admin or owner access rights have given this access. Let $l = e^a.\texttt{label}$, $u = e^a.\texttt{userId}$, $d = e^a.\texttt{deviceNo}$, and $c_a = e^a.\texttt{acount}$. Assume also that o_c is the current object identifier. Then the signature is computed as $e^a.\texttt{signature} = sign(k_u^{s:priv}, l\|u\|d\|c_a\|o_c\|e^a.\texttt{users})$.

  – Event value: e^a.users. This event value consists of a sequence of access rights tuples $(r_1, r_2, \ldots r_n)$. Each access right tuple r is a quadruple $r = (u, a, k, t)$, where $u \in U$ is the user identifier that is given access, a is the access level, k the encrypted key used for asymmetric encryption of the field(s), and t the authentication tag. If $e.\texttt{label} \neq \perp$, then this is a field-level access assignment and $a \in \{\texttt{admin}, \texttt{rw}, \texttt{r}\}$, else it is an object-level access assignment and $a \in \{\texttt{owner}, \texttt{admin}, \texttt{rc}, \texttt{r}\}$. We use the notation $e^a.\texttt{users}.u$ to denote the set of all user identifiers that are assigned new user access rights in event e^a. Note that an access event only states *additional* users who get access rights, or updates of access rights for a user. To compute the complete access rights for an object or for an object field, the whole sequence of access events needs to be considered. The actual field data of an object is encrypted using symmetric encryption. The key used for encryption/decryption of field data is either k_o^{obj} or $k_{o,l}^{obj}$. See the *patch* event for more details.

For a specific object o, we write $U^o_{\textit{obj}}$ for the set of users with object-level access. The sets $U^o_{\textit{obj}}$ is computed as follows. Let E_o be the set of events for object $o \in O$. Let n be the largest event number where $e^r_n \in E_o$ and $e^r_n.\texttt{label} = \bot$ or value 0 if such *reset* event does not exist. Then we have $U^o_{\textit{obj}} = \bigcup_{k=n}^{\infty} e^a_k.\texttt{users}.u$ where $e^a_k.\texttt{label} = \bot \wedge e^a_k \in E_o$.

Let us consider the different cases for an access event e^a_m where $m = \lceil E_o \rceil$.

1. Case $e^a_m.\texttt{label} = \bot$. This means that it is an object-level access event. In this case, all users $e^a_m.\texttt{users}.u$ shall be given access to k_o^{obj}. If k_o^{obj} does not exist, it is create by $e^a.\texttt{userId}$ (see below).
2. Case $e^a_m.\texttt{label} = l$ and $e^a_m.\texttt{users}.u \subseteq U^o_{\textit{obj}}$. This means that the field with label l uses the shared object key k_o^{obj} and all users $e^a_m.\texttt{users}.u$ has already access to k_o^{obj}. The access right tuples in $e^a.\texttt{users}$ assigns $\bot$ to key k and tag t.
3. Case $e^a_m.\texttt{label} = l$ and $e^a_m.\texttt{users}.u \nsubseteq U^o_{\textit{obj}}$. This means that we have an access update for a field where at least one user has access to the field, but not the rest of the object. For this case, there are two sub-cases:
   * Case $k_{o,l}^{obj}$ does not exist. In this case, the access event is legal only if $U^o_{\textit{obj}} \subset e^a_m.\texttt{users}.u$, that is, we encrypt keys for both field-level and object-level access users. Key $k_{o,l}^{obj}$ is create by $e^a.\texttt{userId}$ (see below).
   * Case $k_{o,l}^{obj}$ exists. $k_{o,l}^{obj}$ is used for giving access to users $e^a_m.\texttt{users}.u$.

When the access is changed, either the key k_o^{obj} or the key $k_{o,l}^{obj}$ is used. As described above, only one of these keys are selected for a specific label, and from now on we just use the notation k^{obj} to denote the selected key. If the key does not exist, a new symmetric key is generated using secure pseudorandom number generation $k^{obj} = prng(k)$, where the default key size k is 128-bit. Hence, for an access right quadruple (u, a, k, t), the encrypted key k and tag t is computed as $(k, t) = aead_e(k^{obj}, k^{shared}_{e^a.\texttt{userId},u}, n, \bot)$. This means that the key k^{obj} is encrypted and stored using the shared secret between the user who changed the access (an *owner* or a user with *admin* access) and the user u who gained access. Hence, user u can decrypt and obtain the key for encrypting/decrypting field data by performing $k' = aead_d(k, k^{shared}_{e^a.\texttt{userId},u}, n, t, \bot)$ where k' is the decrypted version that is equal to k^{obj}. The nonce is computed using $n_u = e^a.\texttt{userId}$, $n_d = e^a.\texttt{deviceNo}$, and $n_c = e^a.\texttt{acount}$[4].

* **Event type: *patch* (*p*)**. A *patch* event e^p creates or updates a field value. Several patch events can be created concurrently for the same object when the devices are offline. The final order of events is decided by the home server. The event values for *patch* events are as follows:
  - Event value: $e^p.\texttt{userId}$. This event value specifies the user who performs the patch.
  - Event value: $e^p.\texttt{deviceNo}$. The unique device number for user $e^p.\texttt{userId}$.
  - Event value: $e^p.\texttt{label}$. The event value $e^p.\texttt{label}$ contains the label l of the field that is either created (if it did not exist before) or updated (if existed).
  - Event value: $e^p.\texttt{pcount}$. A patch counter number counting the number of *patch* and *delete* events a user with a specific device has created for a specific object. Note that in contrast to access counting using `acount` (which is used in *owner*, *reset*, and *access* events), $e^p.\texttt{pcount}$ is not globally unique for an object. This uniqueness is for one object for a specific user on a specific device. This makes it possible for several users to create patches concurrently, even in offline mode. The event value starts with 1 and is after that incremented by the client device. The value is used for computing the nonce.
  - Event value: $e^p.\texttt{acount}$. The event value `acount` denotes the access count number that has the keys used for patch encryption. By introducing access count points in patches, we get an explict order relationship between patches and object keys. It also enables the home server to reject a patch to make it possible for the client to re-encrypt, if the encryption key has changed.

[4]Before the counter exceeds its max value of $2^{32} - 1$, the shared secret $k^{shared}_{e^a.\texttt{userId},u}$ needs to be renewed. The procedure for performing this update is outside the scope of this document.

  - Event value: e^p.cipherText. This event value is used for storing the cipher text after applying symmetric encryption using AEAD. The cipher text for the patch event is computed as follows: $(e^p.\texttt{cipherText}, t) = aead_e(m, k^{obj}, n, l \| c_a \| o_c)$, where m is the plaintext message of the field value, k^{obj} is the object key (either $k_{o,l}^{obj}$ or k_o^{obj} depending on the access configuration), n is the nonce, $l = e^p$.label, $c_a = e^p$.acount, and o_c the current object identifier. The nonce is computed using $n_u = e^p$.userId, $n_d = e^p$.deviceNo, and $n_c = e^a$.pcount (see Section 2.2). Note how the AEAD encrypt function also returned the authentication tag t. This tag is stored in the next event value, e^p.authTag.
  - Event value: e^p.authTag. This is the autentication tag generated when computing the cipher text value e^p.cipherText (see above). Both the cipher text $c = e^p$.cipherText and the authentication tag $t = e^p$.authTag are used when decrypting the patch. The plain text message m is decrypted as follows: $m = aead_d(c, k^{obj}, n, t, h)$. Note how the header h can be constructed by fields that are available in the patch, e.g., $h = l \| c_a \| o_c$, where $l = e^p$.label, $c_a = e^p$.acount, and o_c the current object identifier.
  - Event value: e^p.signature. This value can optionally be used to explicitly prove the origin of the patch. If the signature is not used, then e^a.signature $= \perp$. Let $l = e^p$.label, $u = e^p$.userId, $d = e^p$.deviceNo, $c_a = e^p$.acount, $c_p = e^p$.pcount, $c = e^p$.cipherText, and $t = e^p$.authTag. Assume also that o_c is the current object identifier. Then the signature is computed as e^p.signature $= sign(k_u^{s:priv}, l \| u \| d \| c_a \| c_p \| c \| t \| o_c)$. Note that this signature only needs to be used if the users who have access to an object field cannot trust each other.

  Note that labels, such as e^p.label, are not protected for confidentiality. If the application wants to protect the label itself, it can generate a unique label using e.g. UUIDv1, and then include the mapping between this unique label and the real label inside an encrypted field. Because this can be done on top of SDS, it is not included in this description.

- **Event type: *delete* (*d*).** A *delete* event e^d deletes a specific field. The event values for *delete* events are as follows:
  - Event value: e^d.userId. This event value specifies the user who deletes the field.
  - Event value: e^d.deviceNo. The unique device number for user e^d.userId.
  - Event value: e^d.label. The event value e^d.label contains the label l of the field that is deleted.
  - Event value: e^d.pcount. A patch counter number counting the number of *patch* and *delete* events a user with a specific device has created for a specific object. See the description of pcount in the *patch* event.
  - Event value: e^d.acount. The even value acount denotes the access count number that has the object key for this field.
  - Event value: e^d.signature. An optional value that explicitly proves the origin of the delete event. If the signature is not used, then e^d.signature $= \perp$. Let $l = e^d$.label, $u = e^d$.userId, $d = e^d$.deviceNo, $c_a = e^d$.acount, and $c_p = e^d$.pcount. Assume also that o_c is the current object identifier. Then the signature is computed as e^d.signature $= sign(k_u^{s:priv}, l \| u \| d \| c_a \| c_p \| o_c)$.
- **Event type: *transaction* (*t*).** A *transaction* event e^t is used for grouping together events sent by a client. It does not have any security implications in itself. It is used by a client device to keep track of events that have not yet been acknowledged by a home server. The event values for *transaction* events are as follows:
  - Event value: e^t.userId. The user who created the transaction.
  - Event value: e^t.deviceNo. The unique device number for user e^t.userId.
  - Event value: e^t.transactionNo. An integer number that is unique to a user on a specific device for a specific object.

5 Conclusions

This document presents a DRAFT of the Secure Data Sharing (SDS) protocol. The current version describes most of the aspects of the model. However, certain parts are not yet included in the description, including user authentication and user connect management.

Acknowledgements

Thanks to Jacob Wahlgren and Sam Hedin for their work on formalizing parts of the protocol in the Tamarin prover. The input from their work has improved the model and solved a few critical issues in early drafts of this document.

Revisions

- DRAFT-v0.01 Initial version with notations, definitions, and authenticaton.
- DRAFT-v0.02 New sub-section about trusted computing base and communication channels.
- DRAFT-v0.03 Initial parts of the object formalization.
- DRAFT-v0.04 Added *owner* and *access* events. Updated AEAD to support nonce. Minor updates in various places in the document.
- DRAFT-v0.05 Added the *reset* event.
- DRAFT-v0.06 Solved security issues in *owner* and *access* events. Added *patch*, *delete*, and *transaction* events.
- DRAFT-v0.07 Minor fixes.
- DRAFT-v0.08 Updated the name of the protocol and removed basic mode.
- DRAFT-v0.09 Minor edits before the first public release.