Membership Services in Hyperledger Fabric

How the Current Architecture Supports the Various Scenarios

WEDNESDAY, JULY 13, 2016

DARKMATTER

IBM

CONTENTS

▶01

Role-based Access Control

ROLE-BASED ACCESS CONTROL

Authorized auditors have automatic access to transaction participant(s)' attributes through TCA or other authorized key distribution entity (long-term keys) – Requires no action by or interaction with transaction participants or validators *(so as to thwart audit bypass, minimize transaction awareness & interference, and reduce compliance checking overhead)*

Does not, however, preclude the admission of auditors as a transaction participant type

Automatic access can be revoked/limited via key versioning

This User was granted limited- (i.e., transaction- specific-) access by transaction creator

This User's access may include transaction-specific data that is not directly available to auditors, i.e., without transaction-specific authorized communication: consistent with graduated release of information as access need and eligibility dictate

▶02

Asset Transfer

ASSET TRANSFER USE CASE

- Setup by Users: The auditable generation of an ECert by ECA and TCerts by TCA, where TCert key pairs are formally constructed from ECert key pairs
  - ECert: The ECert public key is associated with UserID and Affiliation upon proof of their ownership verified by RA during User registration
  - TCert template: Attributes are collected into an encrypted template within addressable ACA database for targeted accessibility to TCA(s) upon proof of Attributes ownership verified by ACA. Each such template forms the basis of one or more batches of TCerts, each of which includes a uniquely encrypted representation of each of the Attributes in that template
    - A batch of TCerts may include a sub-batch of Key Agreement TCerts. Each Key Agreement TCert includes a key agreement TCert public key (rather than a signature verification TCert public key), and its included encrypted Attributes may be limited to UserID and Affiliation
- A User draws an asset from an account in order to transfer the asset to another account (that belongs to that User or another User)
  - Unlike Bitcoin, the certificate-based system can safely retain a single signature verification capability even if a single transaction draws from multiple accounts (via TCerts that may span multiple TCAs)

ASSET TRANSFER BASIC TRANSACTION STRUCTURE EXAMPLE (1 0F 2)

- Pre-requisite: User acquires a TCert that includes the source accountID as an encrypted Attribute, a self-owned key agreement TCert, and a key agreement TCert and destination accountID from the intended asset transferee. Each Attribute in the TCert is uniquely encrypted using a TCert- and Attribute- specific symmetric key K_attrib, which is derived from a TCert-unique key K_TCert that is delivered to the User with the TCert (and that is passively accessible to authorized auditors). The User as Transaction Creator releases the K_attrib that corresponds to the source accountID (as well as any additional K_attrib values required by the asset transfer policy).

- Let Key_V denote an AES key derived from a one-pass elliptic curve Diffie-Hellman (ECDH) operation, where the static key pair (VPrivKey, VPubKey) is associated with a Validator group and the ephemeral key pair for the transaction (TxnPrivKey, TxnPubKey) is generated by the transaction creator. Similarly for Key_TC and Key_AT, for Transaction Creator and Asset Transferee.

- VPubKey (or the identifier of the Validator group or identifier of a certificate that includes VPubKey) may be included within the TCert by the issuing TCA if the TCA knows which Validator group to designate based on the Attributes. This measure could be used to enforce/audit compliance by the User as Transaction Creator.

ASSET TRANSFER BASIC TRANSACTION STRUCTURE EXAMPLE (2 0F 2)

Transaction =

Fresh TCert of Transaction Creator || Transaction Signature over Text generated using TCert private key || Text = TxnPubKey || key agreement TCert_AT || key agreement TCert_TC || AES_Encrypt $_{Key\_V}$ (K_attrib(s) of Transaction Creator || Asset || accountID of Asset Transferee) || AES_Encrypt $_{Key\_AT}$ (Asset || accountID of Asset Transferee*) || AES_Encrypt $_{Key\_TC}$ (TxnPrivKey)

- *One or more of K_attrib(s) may also be made available to Asset Transferee
- This structure is extensible to a case where one or more of the K_attrib(s) need to be released “publicly” – In that case, such K_attrib(s) may appear in the clear within Text. This may be important where there are “fabric-layer” roles instead of or in addition to “application-layer” roles, and where such “fabric-layer” roles need to be externally visible.
- State is checked as indicating that accountID of Transaction Creator does indeed show Asset is available (such as available funds if Asset is a dollar amount). Upon successful validation and consensus, state is updated to show the Asset has been transferred from accountID of Transaction Creator to accountID of Asset Transferee

DECENTRALIZATION AND SCALABILITY EXTENSION: TCA HIERARCHY

Rather than having a TCA generate a batch of TCerts directly from a TCert template, it can be advantageous to utilize Primary TCA(s) and Subordinate TCA(s)

- A Primary TCA is accountable for verifying the provenance of a TCert template retrieved from the ACA database, e.g., by verifying timely signed assertions that were generated by Attribute Authorities and provided to the ACA via the User. Such TCert template can be independently verified via multiple Primary TCA components that collectively generate a TemplateTCert (e.g., via threshold signatures that are verifiable using a single certified Primary TCA public key) that is made available to Subordinate TCAs

- A TemplateTCert is usable by any Subordinate TCA that is authorized to access and derive K_TCert values for the Affiliation within the TemplateTCert, so that it can generate (operational) TCerts to permission transaction creation (as well as key agreement TCerts for read access). A Subordinate TCA that violates the conditions imposed by a TemplateTCert can have its Subordinate TCA public key certificate revoked

ADDING CAPABILITY TO LIMIT ATTRIBUTE ACCESS TO VERIFY-ONLY MODE

Verify-Only Mode provides a means for a Transaction Creator to enable a Validator group to corroborate what they already expect an Attribute value to be

- Rather than straight encryption of an Attribute using a K_attrib derived from K_TCert, K_TCert can be used to derive attribute-specific Masking Keys that are used in turn to derive attribute-specific Encryption Keys
- Each Encrypted&Masked Attribute that is selectively released can be released for use either in Decrypt Mode or Verify-Only Mode
- Decrypt Mode: Given a Masking Key, one can generate the Mask and the Encryption Key, so that the Mask is exclusive-or’ed to the Encrypted&Masked Attribute and the resultant value is decrypted using the Encryption Key
- Verify-Only Mode: Given an Encryption Key, one can encrypt a presumed Attribute value, exclusive-or the result to the Encrypted&Masked Attribute, and then process the result to check whether it matches against the received Encryption Key

▶03

Structure and Utilization of TCerts (Detailed)

STRUCTURE OF TCERTS: SECURE* ATTRIBUTES

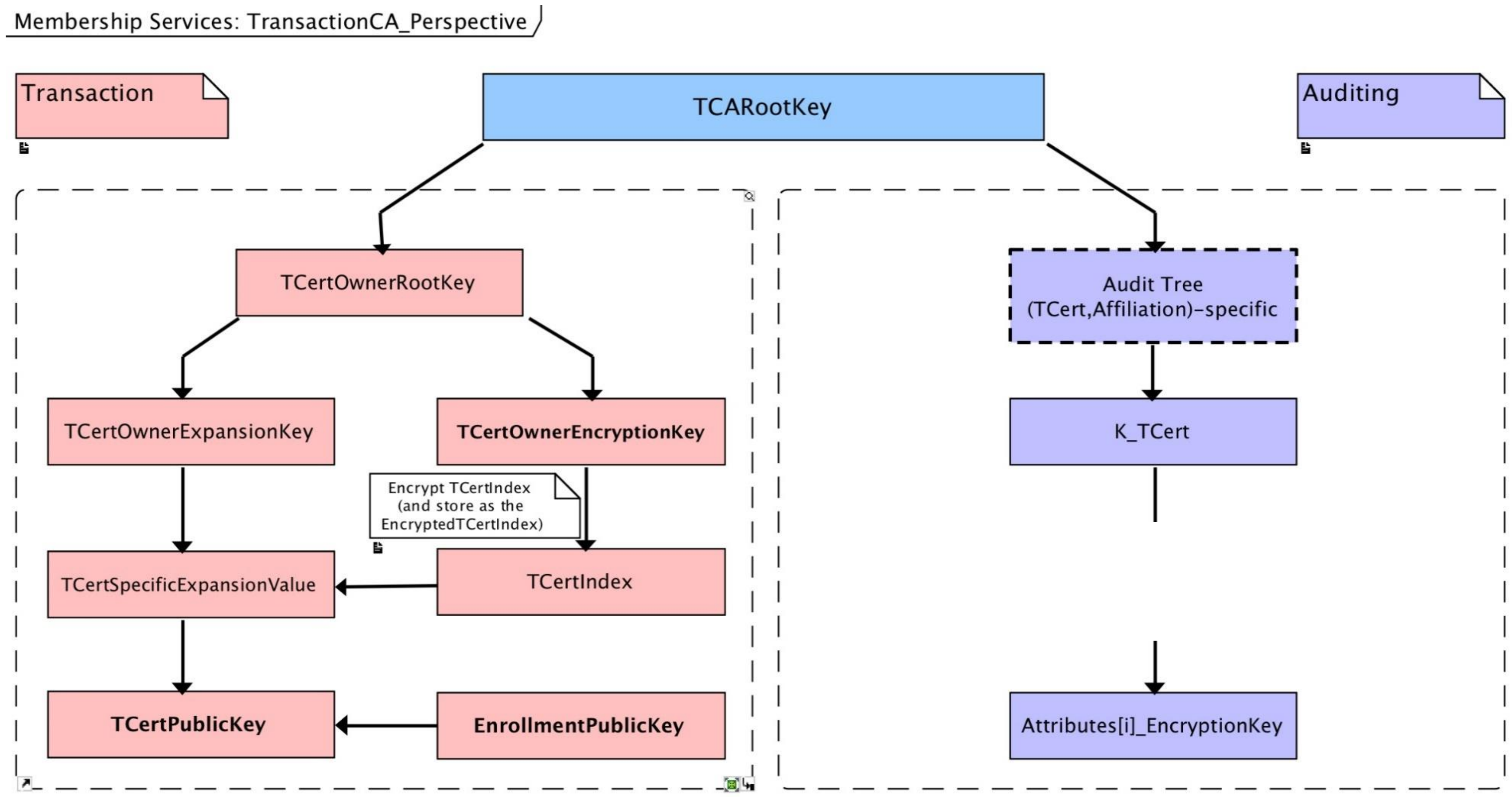
- Encryption of attributes using symmetric keys with the following properties:
  - Different per TCert (to ensure unlinkability, which encryption with a reused key would not)
  - Different per attribute (to enable User-initiated selective release, so as to make the acquisition of TCerts and their use in transactions efficiently asynchronous)
  - Independently accessible to only those auditors whose (hierarchical) authorization includes the Affiliation of that User as extracted by the TCA from an Enrollment Certificate used to authenticate the request for issuance of the TCert
- Mandatory attributes include (primary) Affiliation and UserID [mandatory for inclusion within TCerts for auditability, but not necessarily released during transactions]
- The TCert-specific symmetric key from which the attribute-specific symmetric keys above are derived is securely provided to the User with the TCert. As an aid in resilience against device crash, the key need no longer be retained once the TCert is used in a transaction

* **SECURE in the sense of CONFIDENTIAL (& PRIVACY PRESERVING)**

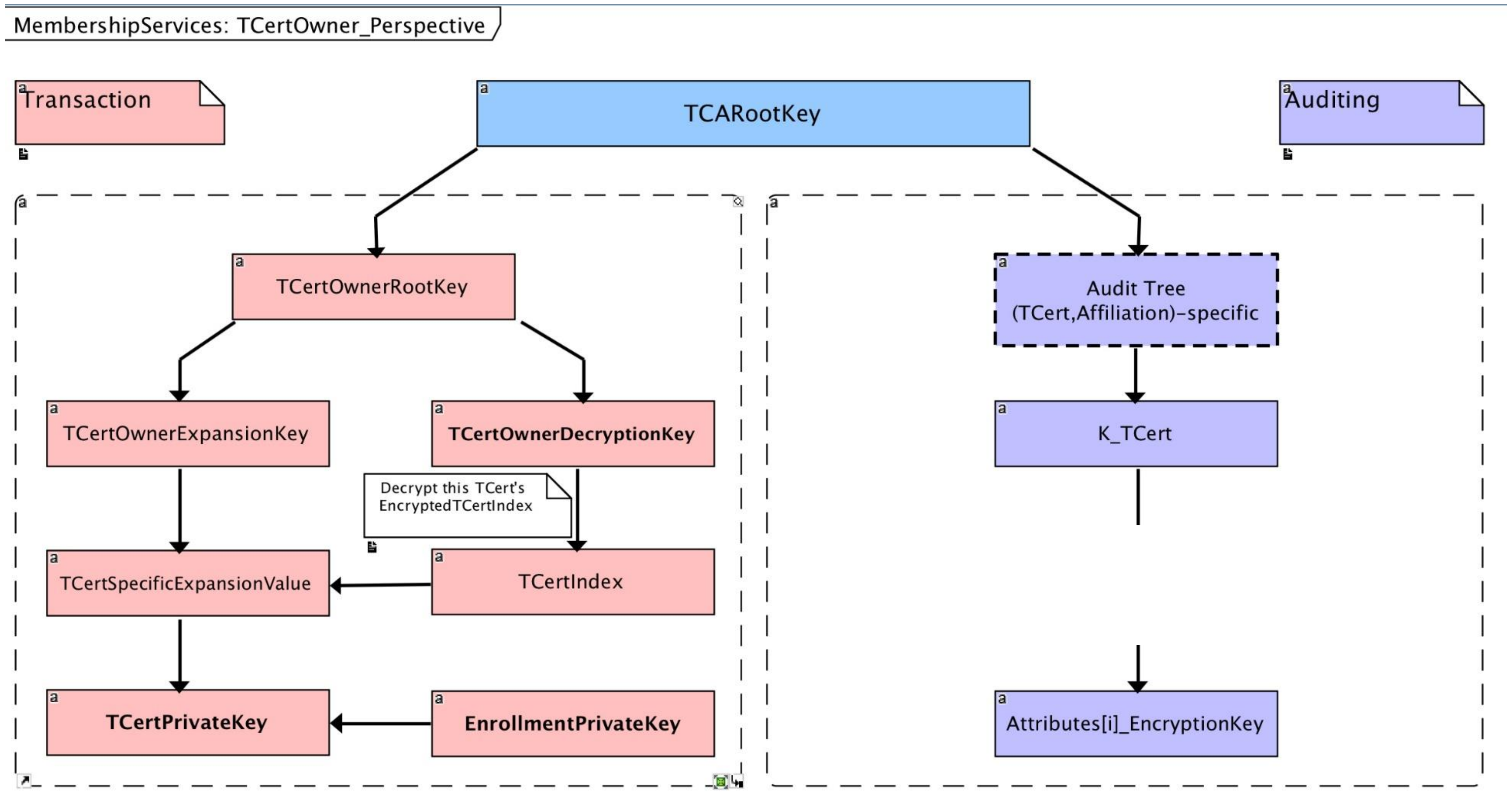
STRUCTURE OF TCERTS: SECURE MAPPING OF LONG-TERM KEY TO SHORT-TERM KEYS

- *Expansion function:* Encryption of a TCert-unique parameter (TCertIndex) that is used by the TCA for construction of the TCert public key from the ECert's public key and for recovery by the specific User of the TCert private key from the private key corresponding to the ECert's public key
- The TCert private key is used to sign a transaction that is permissioned via the TCert that accompanies the transaction; the TCert public key is used to verify the authenticity and integrity of the transaction
  - Extension: Additional TCerts could be included to corroborate one or more attributes that span TCAs or to handle disjoint attributes the collection of which requires input from multiple TCAs.
- Enables bandwidth- and computationally- efficient request and production of TCerts
  - Request does not include candidate TCert public keys or proof of knowledge of TCert private keys
- Enables portable security across User's devices without retention/synchronization of TCert-specific data, as optimized for blockchain-based access to unencrypted TCerts within past transactions
  - Conducive to secure instantiation of complex chained/sequenced transactions

KEY MANAGEMENT - TCA PERSPECTIVE


Membership Services: TransactionCA_Perspective
Transaction
TCARootKey
Auditing
TCertOwnerRootKey
TCertOwnerExpansionKey
TCertOwnerEncryptionKey
Encrypt TCertIndex (and store as the EncryptedTCertIndex)
TCertSpecificExpansionValue
TCertIndex
TCertPublicKey
EnrollmentPublicKey
Audit Tree (TCert,Affiliation)-specific
K_TCert
Attributes[i]_EncryptionKey

KEY MANAGEMENT – USER PERSPECTIVE


MembershipServices: TCertOwner_Perspective
Transaction
TCARootKey
Auditing
TCertOwnerRootKey
Audit Tree (TCert,Affiliation)-specific
TCertOwnerExpansionKey
TCertOwnerDecryptionKey
K_TCert
Decrypt this TCert's EncryptedTCertIndex
TCertSpecificExpansionValue
TCertIndex
TCertPrivateKey
EnrollmentPrivateKey
Attributes[i]_EncryptionKey

AUDIT SUB-TREE – KEY HIERARCHY (EXAMPLE)

Provides appropriate auditability independently of transaction creation and validation -- via PreKeys

- PreK_ABC [(3)] : available to TCA and auditors of {Automobile, Banking, Construction}
- PreK_A [(2)] = HMAC(PreK_ABC [(3)], "Automobile"): available to auditors of {Automobile}
- PreK_B [(2)] = HMAC(PreK_ABC [(3)], "Banking"): available to auditors of {Banking}
- PreK_C [(2)] = HMAC(PreK_ABC [(3)], "Construction"): available to auditors of {Construction}
- PreK_BofZ [(1)] = HMAC(PreK_B [(2)], "BofZ"): available to auditors of Bank of Z, an element of {Banking}
- PreK_[BofZ, xyz] [(0)] = HMAC(PreK_BofZ [(1)], TCertID="xyz") = K_TCert: available to User as owner of TCert with Affiliation (extracted by TCA from ECert) = BofZ and TCertID = "xyz"

PreK_ABC [(3)] is derived (possibly after multiple iterations, depending on the depth of this node in the full Audit Tree) from PreK_Root

To handle renewability (in response to revoked auditor(s) or to otherwise limit access), PreK_Root can be set as PreK_Root = HMAC(TCARootKey, KeyVersion). KeyVersion can be included within TCerts to facilitate the audit process. An auditor of {Banking}, say, is securely provisioned with PreK_B [(2)] for one or more values of KeyVersion

TRANSACTION CERTIFICATES (TCERTS) – BASIC STRUCTURE

- **TCertID** (unique per TCert)
- **Encrypted Attribute(s)** [= AES_CBC_Encrypt $_{\text{K\_attrib}}$ (Attribute || Constant Pad)] (where encryption key K_attrib is unique per attribute, and K_attrib is derived from the TCert-specific key, K_TCert, provided to the TCert owner). Attributes can be as finely-granulated as pertaining to a single user. Includes two mandatory attributes, i.e., UserID and Affiliation
  - Constant Pad thwarts attribute substitution attack by owner of TCert
- **TCert Public Key** (for verifying the signatures on transactions)
- **Encrypted TCertIndex** [= AES_CBC_Encrypt $_{\text{TCertOwnerEncryptionKey}}$ (TCertIndex || Constant Pad)]

(where TCertOwnerEncryptionKey is derived from a key, TCertOwnerRootKey, supplied to the TCert owner with the TCert batch (constant for all batches of that TCert owner))

TRANSACTION CERTIFICATES (TCERTS) - REVOCATION & EXPIRATION

Revocation Field [= Truncated_Hash(TCertID || Truncated_RevocK)]

- For TCA_Revoke_Key available to the TCA, let RevocK = HMAC(TCA_Revoke_K, Attribute(s) || ValidityPeriod) --- where Attribute(s) and ValidityPeriod correspond to the particular batch of TCerts. RevocK may be truncated to yield Truncated_RevocK
- Let the TCA-signed CRL (or delta CRL) include entries of the form ValidityPeriod || Truncated_RevocK(s)
- The revocation test for a given TCert entails checking all RevocK(s) (if any) in a CRL that correspond to the same ValidityPeriod as that in the TCert being tested

ValidityPeriod (may be common to all TCerts within a batch (as well as other unrelated batches))
[Not Valid Before xxxx, Not Valid After yyyy]

KEY MANAGEMENT SPECIFICS (1 0F 2)

384-bit TCARootKey generated by / available to TCA (may be made available to most powerful auditor(s) relative to that TCA)

384-bit TCertOwnerRootKey = HMAC(TCARootKey, EnrollmentPublicKey)
- This key is delivered to TCertOwner's Client by TCA within Batch Response

256-bit TCertOwnerEncryptionKey = $[\text{HMAC(TCertOwnerRootKey, “1”)}]_{\text{256-bit truncation}}$

384-bit TCertOwnerExpansionKey = HMAC(TCertOwnerRootKey, “2”)

384-bit TCertSpecificExpansionValue = HMAC(TCertOwnerExpansionKey, TCertIndex)

KEY MANAGEMENT SPECIFICS (2 OF 2)

TCertPublicKey and TCertPrivateKey derivation (using P-384):

- Computable by TCA / most powerful auditor(s) using stored TCARootKey, and EnrollmentPublicKey extracted from TCertOwner's enrollment certificate:

  TCertPublicKey = EnrollmentPublicKey + TCertSpecificExpansionValue *G*

- Computable by TCertOwner using TCertOwnerRootKey retrieved from Batch Response, and generated/stored EnrollmentPrivateKey:

  TCertPrivateKey = (EnrollmentPrivateKey + TCertSpecificExpansionValue) modulo *n*
  - *n* per NIST FIPS PUB 186-4
  - Note that the TCertOwner need not retain TCertPrivateKey, since it is recomputable from the Tcert

TCertPublicKey derivation process is auditable (given access to TCARootKey or TCertOwnerRootKey). For such audit purpose, a key, TCertsRoot, can be placed between TCARootKey and TCertOwnerRootKey, namely, HMAC(TCARootKey, KeyVersion) which can coincide with PreK_Root, or for greater granularity PreK_Root can be set as HMAC(TCARootKey, "1" KeyVersion) and TCertsRoot can be set as HMAC(TCARootKey, "2" KeyVersion). Providing TCertOwnerRootKey(s) to an auditor (without audit tree access) restricts user identification of transactions to those specific users

ADDITIONAL TCERT TYPE: KEY AGREEMENT TCERTS

- Incorporate a key agreement public key (in place of signature verification TCert public key) into a streamlined variant of TCerts. Such key agreement TCerts, when one or more of these are included within a transaction, allow TCert owner(s) to independently determine whether or not they are a participant of the particular transaction, and if so, to retrieve a per-transaction decryption key by recovering the key agreement private key corresponding to the key agreement public key within the TCert. They also allow an authorized auditor to determine the identities of the transaction participants intended by the transaction creator

USING KEY AGREEMENT FOR SELECTIVE RELEASE OF TRANSACTION CREATOR'S ATTRIBUTES

- Use Validator group key-agreement public key
- Use intended recipients' key-agreement public keys
- Reflexive case: use transaction creator key-agreement public key
- This mechanism also enables targeted plaintext-access to chain-code and/or other data/metadata that may be included as ciphertext within the transaction
- Intended transaction participants are not necessarily granted equal access relative to one another or to the particular Validator group or to the transaction creator

KEY AGREEMENT

Key agreement between transaction creator and Validator group:

a. Transaction creator: uses a Validator group public key and a transaction creator random private key

b. Validator group: uses a Validator group private key and a transaction creator random public key

The random public key from (b) is included in the clear in the transaction. This applies for the rest of the cases mentioned below

Key agreement between transaction creator and intended asset recipient:

c. Transaction creator: uses intended asset recipient's key agreement TCert public key & transaction creator random private key from (a)

d. Intended asset recipient: uses key agreement TCert private key & transaction creator random public key from (b)

The intended asset recipient's key agreement TCert is included in the clear in the transaction

Key agreement between transaction creator and the transaction creator:

e. Transaction creator: uses transaction creator key agreement TCert public key & transaction creator random private key from (a)

f. Transaction creator: uses key agreement TCert private key & transaction creator random public key from (b)

The transaction creator's key agreement TCert is included in the clear in the transaction

Shared secret of (e) and (f) can be used to encrypt the transaction creator random private key in particular, for inclusion in ciphertext:

This enables recovery by transaction creator of whatever was encrypted for use by Validator group and intended asset recipients

TWO WAYS TO HANDLE TRANSACTION PARTICIPANT CONTINUITY: ACCESS CONTROL LIST (ACL) OF TCERTS VS. USER-UNIQUE ATTRIBUTES

- In order to securely determine whether a follow-on transaction is created only by an appropriate User, an intended transaction participant can be designated by one of their TCerts or by one of their attributes. It is easier to retain privacy if TCerts are not included within the ACL, since then a follow-on transaction need involve only a single TCert owned by the transaction creator (rather than an encrypted proof of ownership of one of the TCerts that were included in the ACL as well as (cleartext) inclusion of a previously-unused TCert)

- In a given transaction sequence, an attribute may at first be non- User-unique, and then migrate to a User-unique attribute once the User is "identified": e.g., a request is made for the services of a User possessing a certain skill/license/certification; a respondent proves possession of that requested non- User-unique attribute as well as possession of a User-unique attribute (such as a license number). Future transactions in the sequence entail that User-unique attribute

TWO WAYS TO HANDLE TRANSACTION PARTICIPANT ACCESS: KEY AGREEMENT TCERT VS. QUERY TRANSACTION

- If a (new type of) query transaction is used by intended transaction participants to retrieve data/keys that the transaction creator authorized them to access, then such data/keys need be encrypted within the transaction itself for access by only the appropriate Validator group (where encryption involves a Validator group key agreement public key that is available to the transaction creator via a certificate or other reliable means). When used in combination with attribute-based (vs. TCert-based) ACL, access to past transactions can be maintained even if the user's original ECert public key has been superseded (via ECert revocation and reissuance)

- If key agreement TCert(s) of intended participants are available to and used by the transaction creator, then query transactions can be limited in scope (such as to acquire resultant state values). Use of key agreement TCert(s) of intended participant(s) enables additional feature of independent/automatic access by auditors authorized to decrypt the encrypted UserID and Affiliation fields (by virtue of the fact such Affiliation is included within such an auditor's jurisdiction). Note, however, that revocation of a transaction participant's ECert (e.g., due to suspected compromise of the ECert private key) does not prevent access to past transactions. ECert status can be disseminated via CRL or OCSP/OCSP-stapling

TWO WAYS TO IDENTIFY TRANSACTIONS CREATED BY A SPECIFIC USER: GENERATING A LIST OR CHECKING A PARTICULAR TRANSACTION

- For use by Users and/or by authorized auditors
- The User-specific key needed to (re-)generate such a list is securely provided to the User with each successful request for one or more TCerts, where a successful request requires use of the private key corresponding to the ECert's public key. Such user-specific key may be securely provided out-of-band to an auditor that is authorized to access that User's TCert information (or the key from which the User-specific key is derived may be provided to a more powerful auditor that is authorized to access information for all Users)
- Each entry of the list is comprised of the encrypted TCertIndex field
- The TCertIndex is encrypted along with a known/constant pad, so that a User can check this field in any particular transaction to see if the pad is present upon decryption with that User's User-specific key. An auditor that is authorized for the Affiliation to which the User belongs can check the encrypted UserID attribute field instead

▶04

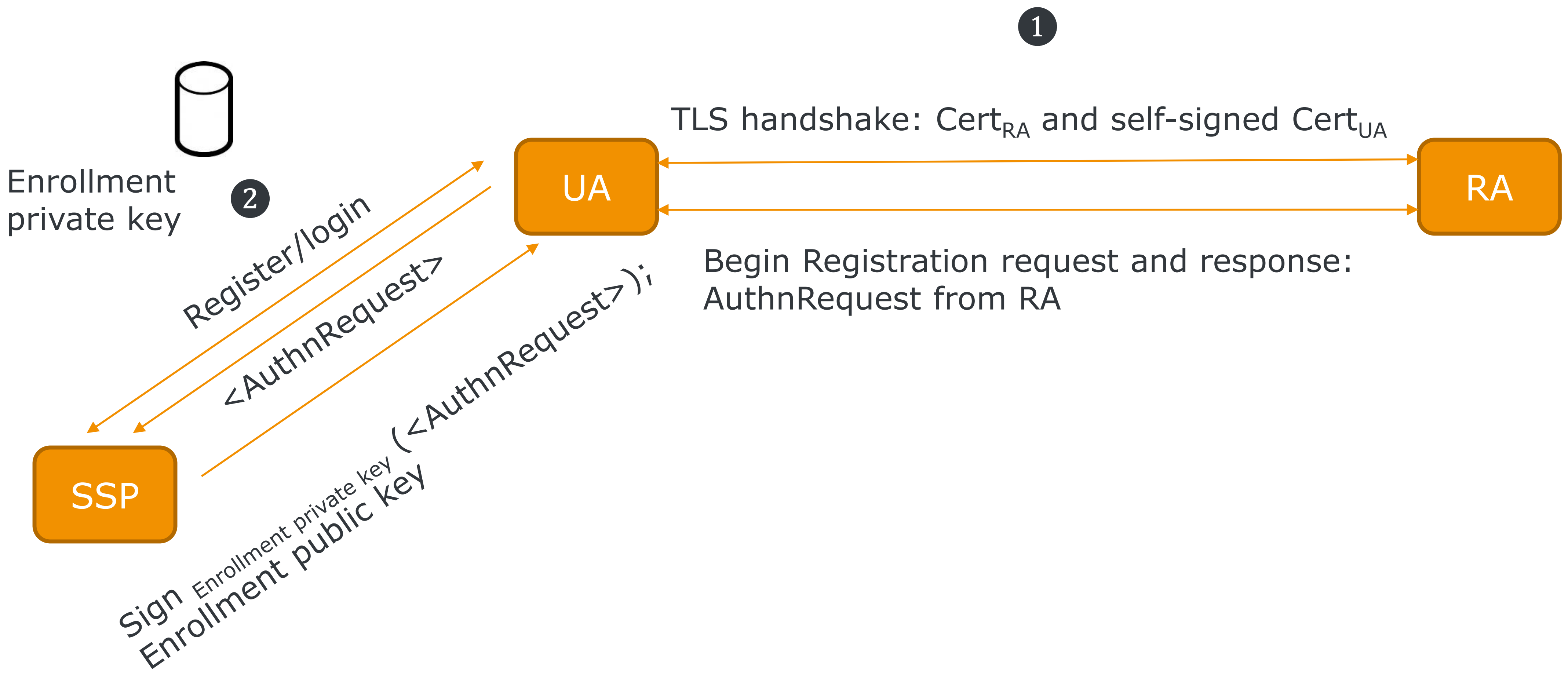
Potential Scenarios

A USER REGISTRATION SCENARIO (1 OF 6)

- User Agent (UA) and Registration Authority (RA) perform mutually authenticated TLS handshake protocol, where UA generates an ECDSA key pair and corresponding self-signed certificate Cert_{UA}. Subsequent communications between UA and RA are TLS-protected.
- RA extracts Cert_{UA} from TLS handshake, and issues signed authentication request to UA: AuthnRequest = <AuthnRequest(RA_ID, Nonce_{RA}, $\text{Timestamp}_{\text{RA}}$, Cert_{UA})>; $\text{Sign}_{\text{RA private key}}$ (<AuthnRequest>). Timestamps here and elsewhere may be included for use during audit.
- User/UA registers with / logs into Signature Service Provider (SSP), and UA passes <AuthnRequest>. SSP generates (ECDSA) Enrollment private key – Enrollment public key key-pair, and provides $\text{Sign}_{\text{Enrollment private key}}$ (<AuthnRequest>) and Enrollment public key to UA. SSP retains Enrollment private key as associated with User login credential(s).

NOTE: Here and subsequently, use of an SSP as an entity that is disjoint from UA is optional, and its presence or absence does not differentially affect the system flow.

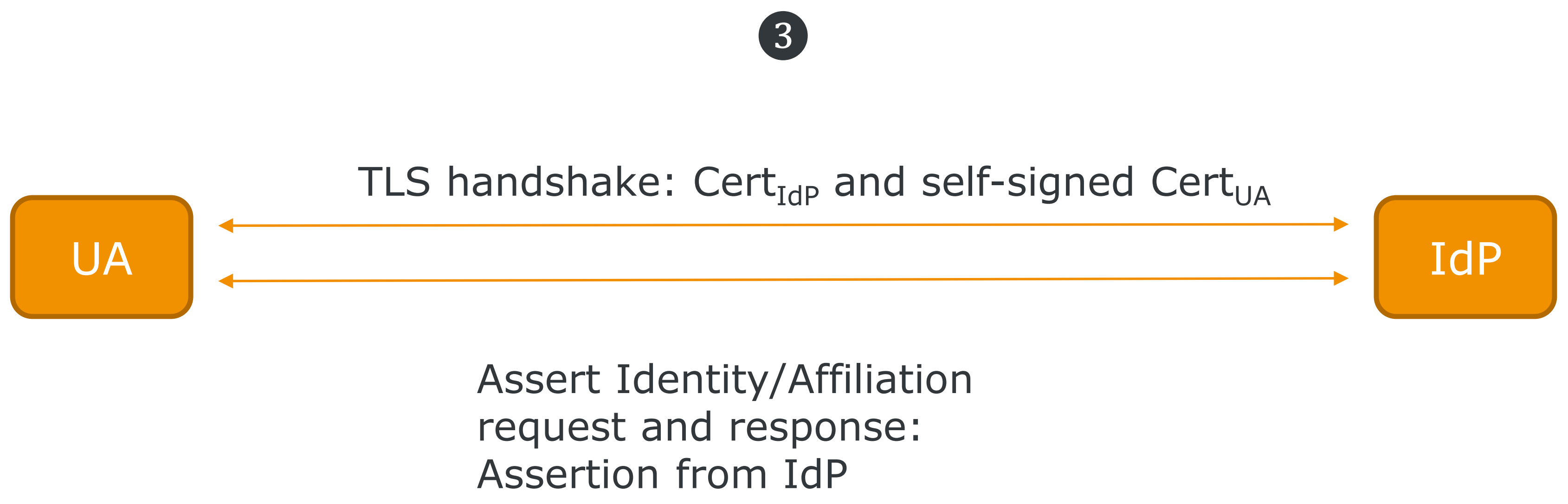
A USER REGISTRATION SCENARIO (2 OF 6)


1
TLS handshake: Cert$_{RA}$ and self-signed Cert$_{UA}$
UA
RA
Begin Registration request and response:
AuthnRequest from RA
Enrollment
private key
2
Register/login
<AuthnRequest>
Sign $_{\text{Enrollment private key}}$ (<AuthnRequest>);
Enrollment public key
SSP

A USER REGISTRATION SCENARIO (3 OF 6)

- UA communicates with (internal or external) Identity Provider IdP. Communications are TLS-protected under mutual authentication (with self-signed $Cert_{UA}$ (same as within AuthnRequest) provided by UA). IdP extracts $Cert_{UA}$ from TLS handshake, and incorporates it into the Assertion it issues.
  - UA provides AuthnRequest, Enrollment public key, $Sign_{\text{Enrollment private key}}$ (<AuthnRequest>).
  - User/UA provides proof of ownership of UserID and Affiliation, as required by IdP. Such proof may involve the use of pre-existing private key(s) held by the UA. For example, a pre-existing public key may be included within a certificate that establishes the private key owner's UserID and/or Affiliation, where such certificate was issued by an entity recognized by the IdP to have such authority. Such entity may be the IdP itself.
  - IdP issues signed assertion to UA: Assertion = <Assertion(IdP_ID, $Nonce_{RA}$, $Timestamp_{IdP}$, $Cert_{UA}$, UserID, Affiliation, AuthnRequest, Enrollment public key, $Sign_{\text{Enrollment private key}}$ (<AuthnRequest>))>; $Sign_{\text{IdP private key}}$ (<Assertion>).

A USER REGISTRATION SCENARIO (4 OF 6)

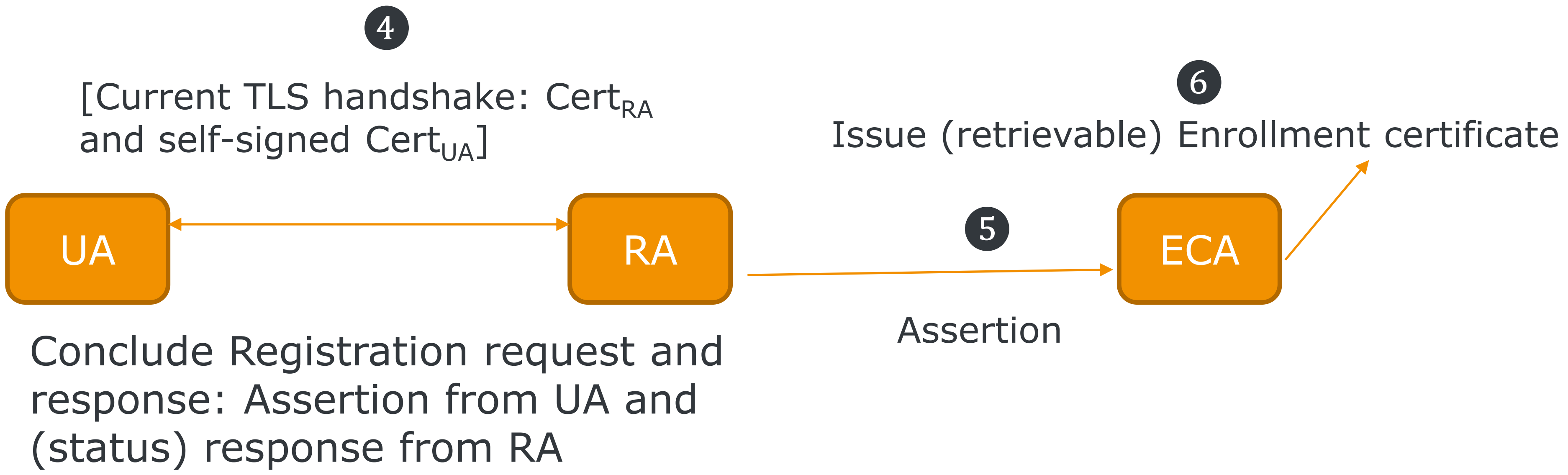

❸
TLS handshake: Cert_{IdP} and self-signed Cert_{UA}
UA
IdP
Assert Identity/Affiliation request and response: Assertion from IdP

A USER REGISTRATION SCENARIO (5 OF 6)

- UA provides Assertion to RA.
- RA makes Assertion available to ECA if it is properly formulated, where, in particular, signatures verify, Nonce_{RA} is current and Cert_{UA} field within Assertion and Cert_{UA} field within included AuthnRequest match Cert_{UA} extracted from current TLS handshake.
- ECA generates Enrollment certificate (ECert), as made available (e.g., via LDAP server or TLS communications) to the ECA, the ACA, TCA, and authorized Auditors upon presentation of UserID and Affiliation. ECA may make use of threshold cryptography for digital signature generation. If so, each signer unit may have independent secure access to RA-approved Assertions (so as not to have an indirect trust model, even if Assertions are not RA-signed).

NOTE: A rogue ECA cannot undetectably substitute another Enrollment public key for which it knows the corresponding Enrollment private key, since the ECA cannot generate an acceptable Assertion without knowledge of an IdP private key.

A USER REGISTRATION SCENARIO (6 OF 6)


4
[Current TLS handshake: Cert_{RA} and self-signed Cert_{UA}]
UA
RA
Conclude Registration request and response: Assertion from UA and (status) response from RA
5
Assertion
ECA
6
Issue (retrievable) Enrollment certificate

A SCENARIO FOR USER-DRIVEN REFRESH OF ATTRIBUTE CERTIFICATE AUTHORITY (ACA) DATABASE (1 OF 6)

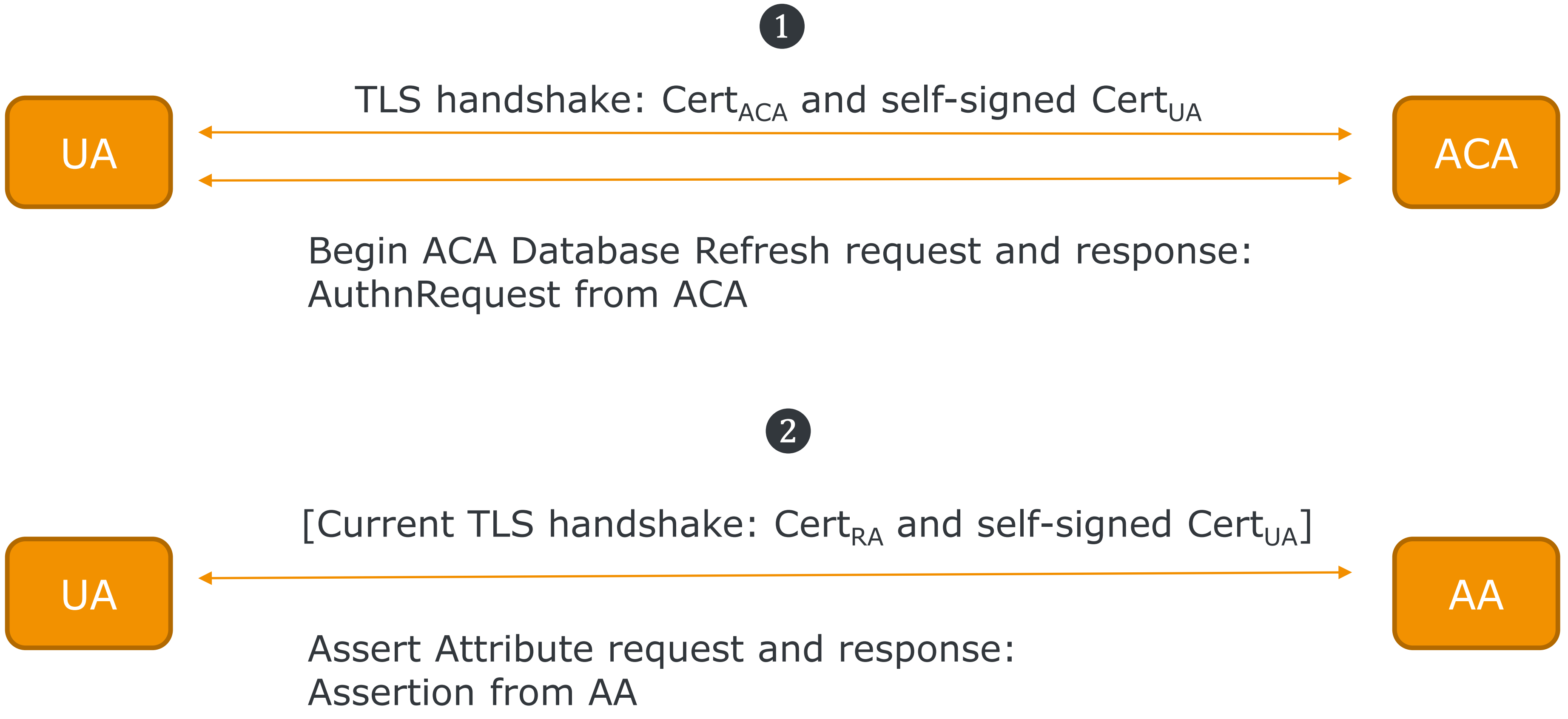
- User Agent (UA) and ACA perform mutually authenticated TLS handshake protocol, where UA generates an ECDSA key pair and corresponding self-signed certificate Cert_{UA}. Subsequent communications between UA and ACA are TLS-protected.
- ACA extracts Cert_{UA} from TLS handshake, and issues signed authentication request to UA: AuthnRequest = <AuthnRequest(ACA, $\text{Nonce}_{\text{ACA}}$, $\text{Timestamp}_{\text{ACA}}$, Cert_{UA})>; $\text{Sign}_{\text{ACA private key}}$ (<AuthnRequest>).
- The step below is iterated, as necessary, across multiple (internal or external) attribute authorities (AA), with disjoint or overlapping attribute(s): UA communicates with AA. Communications are TLS-protected under mutual authentication (with Cert_{UA} (same as within AuthnRequest) provided by UA).

A SCENARIO FOR USER-DRIVEN REFRESH OF ATTRIBUTE CERTIFICATE AUTHORITY (ACA) DATABASE (2 OF 6)

- UA provides AuthnRequest and UserID*.
- User/UA provides proof of ownership of UserID and attribute(s), as required by AA. Such proof may involve the use of pre-existing private key(s) held by the UA. For example, a pre-existing public key may be included within a certificate that is referenced by an attribute/authorization certificate or SAML assertion, where such certificate(s)/assertions were issued by entity(ies) recognized by the AA to have the relevant authority. Such entities may overlap with the AA itself.
- AA issues signed assertion: Assertion = <Assertion(AA_ID, $\text{Nonce}_{\text{ACA}}$, $\text{Timestamp}_{\text{AA}}$, Cert_{UA}, UserID, attribute(s), attribute validity period(s))>; $\text{Sign}_{\text{AA private key}}$ (<Assertion>).

*UserID should be comparable (if not identical) to UserID within the User's Enrollment certificate.

A SCENARIO FOR USER-DRIVEN REFRESH OF ACA DATABASE (3 OF 6)


1
TLS handshake: Cert$_{ACA}$ and self-signed Cert$_{UA}$
UA
ACA
Begin ACA Database Refresh request and response:
AuthnRequest from ACA
2
[Current TLS handshake: Cert$_{RA}$ and self-signed Cert$_{UA}$]
UA
AA
Assert Attribute request and response:
Assertion from AA

A SCENARIO FOR USER-DRIVEN REFRESH OF ACA DATABASE (4 OF 6)

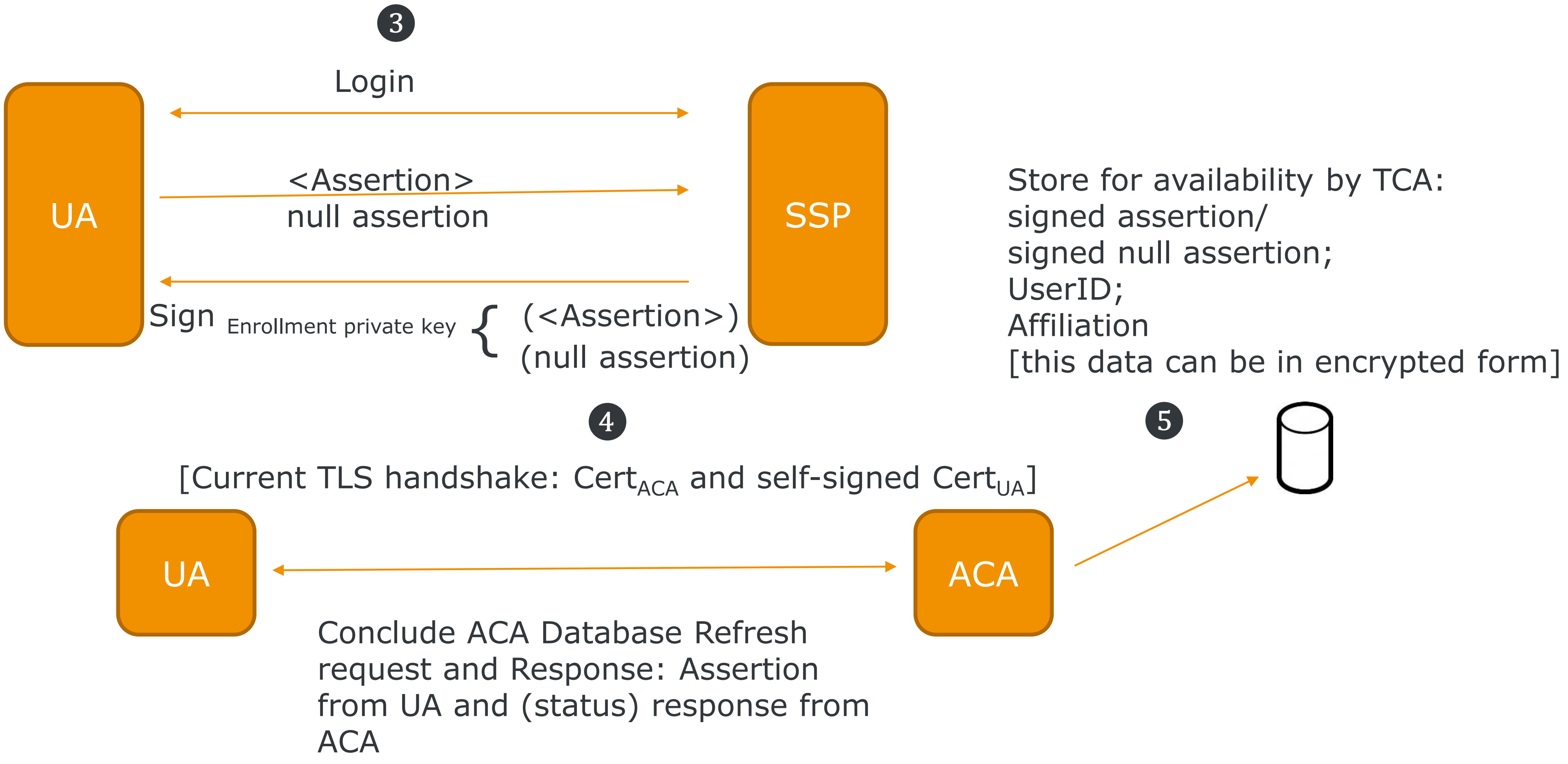
- UA logs into Signature Service Provider (SSP), and passes <Assertion>(s) (as well as UserID and Affiliation (from User registration) if not already associated at the SSP with the User’s account).
  - If a given attribute has a currently valid Assertion within the ACA database, then UA can pass a null assertion (that identifies the attribute) to SSP in place of an AA-issued <Assertion>. Such null assertion is signed by SSP in place of signing an AA-issued <Assertion>.
- SSP provides Sign $_{\text{Enrollment private key}}$ (<Assertion>) (s) and Sign $_{\text{Enrollment private key}}$ (null assertion) (s) to UA
- UA provides AA- and SSP- signed assertion(s), SSP- signed null assertions, UserID and Affiliation to ACA, where UserID and Affiliation should be identical to such fields within a previously issued Enrollment certificate (ECert)*. ACA generates and communicates Rand to UA if there is at least one freshly signed assertion (where Rand is retained by UA and used to refer to ACA database entry during TCert requests). UA is responsible for providing appropriate previously issued Rand value(s) to ACA as matched against SSP- signed null assertion(s), if any – so that ACA can append or point to these within the currently processed ACA database entry.

A SCENARIO FOR USER-DRIVEN REFRESH OF ACA DATABASE (5 OF 6)

- ACA verifies that AA- and SSP- signed assertion(s) and null assertions are properly formulated, where, in particular, signatures verify, $\text{Nonce}_{\text{ACA}}$ is current and Cert_{UA} field within Assertion and Cert_{UA} field within included AuthnRequest match Cert_{UA} extracted from current TLS handshake.
- ACA makes AA- and SSP- signed assertion(s) and SSP- signed null assertions, UserID and Affiliation available to TCA by appropriately updating ACA database (incorporating hash(Rand), and any previous hash(Rand) values as appropriate).
- ACA database encryption option: After verifying signatures (using AA public key(s) and Enrollment public key (per ECert that corresponds to UserID and Affiliation), the ACA can encrypt using one-pass ECDH with a locally generated ephemeral key pair and the current ECDH public key of the TCA (if such is securely available to the ACA via a certificate or other means). If the ephemeral is generated using a true random number generator (as opposed to wholly via a deterministic process), then the key agreement- based encryption will not be reversible by the ACA. The associated hash(Rand) values can remain in the database in the clear, so that the ACA can effectively handle null assertions.

*NOTE: ACA is responsible for determining if UserID included within an Assertion is comparable (if not identical) to the UserID associated with the Enrollment public key (via an ECert).

A SCENARIO FOR USER-DRIVEN REFRESH OF ACA DATABASE (6 OF 6)


3
Login
UA
<Assertion>
null assertion
SSP
Sign Enrollment private key { (<Assertion>) (null assertion)
Store for availability by TCA:
signed assertion/
signed null assertion;
UserID;
Affiliation
[this data can be in encrypted form]
4
5
[Current TLS handshake: Cert ACA and self-signed Cert UA]
UA
ACA
Conclude ACA Database Refresh request and Response: Assertion from UA and (status) response from ACA

ACA-ISSUED “CERTLETS”

What and How:

Since the ACA has access to AA-signed assertions provided to it via User Agents, the ACA has the opportunity to incorporate individual attributes (such as account numbers (each of which may or may not incorporate an indication of the issuing institution)) into streamlined certificates (denoted here as “certlets”) that are sanitized, in that they do not indicate ownership of such attributes. Such (preferably ACA-signed) certlets could be generated by the ACA prior to encryption, if any, of the incoming assertions that are incorporated into the ACA database (or otherwise made available to TCA(s)) after processing by the ACA. Each such certlet may include the validity period of that attribute (as attested to by an internal or external AA). If an attribute does not include self-contained indication of its origin or issuing institution, such may be included by the ACA within the certlet. Such indication of origin may include an ID of one or more AAs that issued relevant assertions made available to the ACA.

There are various options for making certlets available to relying parties. Whether held in plaintext or ciphertext form within the ACA database (or an associated database), these may be retrievable by relying parties. Additionally or internally, certlets may be incorporated into transactions on the blockchain itself (by the ACA or other parties).

Why:

Certlets can serve a useful function relative to one or more conditions that need to be satisfied when considering a transaction, such as an asset transfer, for incorporation into the blockchain. For example, an intended destination/recipient account ID (as designated by a transaction creator) can be checked for current legitimacy by transaction Validators.

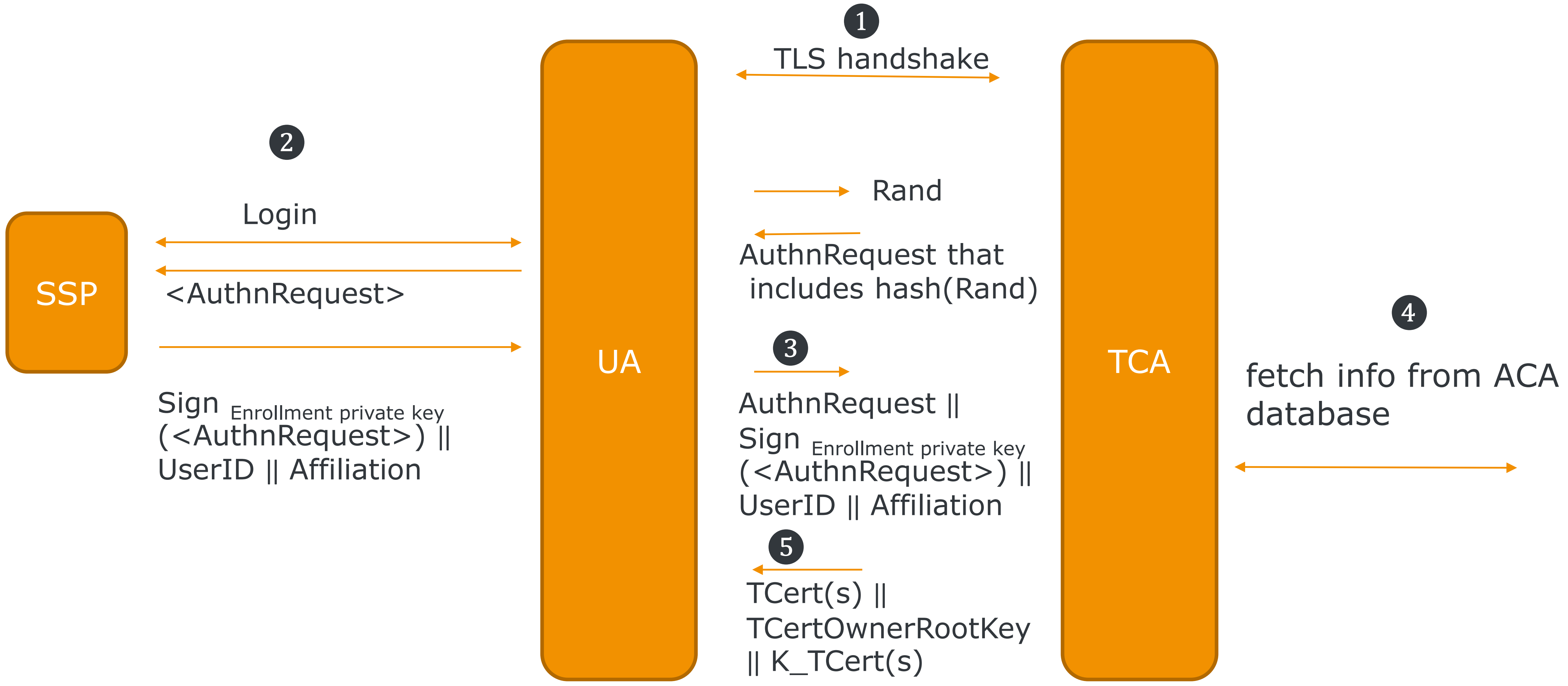
A SCENARIO FOR TCERT REQUEST AND TCERT RESPONSE (1 OF 3)

- User Agent (UA) and TCA perform mutually authenticated TLS handshake protocol, where UA generates an ECDSA key pair and corresponding self-signed certificate Cert_{UA}. Subsequent communications between UA and TCA are TLS-protected. UA provides Rand corresponding to the particular ACA entry/info for which one or more TCerts are requested.
- TCA extracts Cert_{UA} from TLS handshake, and issues signed authentication request: AuthnRequest = <AuthnRequest(TCA, Nonce_{TCA}, Timestamp_{TCA}, hash(Rand), Cert_{UA})>; Sign $_{\text{TCA private key}}$ (<AuthnRequest>).
- UA logs into Signature Service Provider (SSP), and passes <AuthnRequest>.
- SSP provides Sign $_{\text{Enrollment private key}}$ (<AuthnRequest>), UserID and Affiliation to UA.
- UA provides AuthnRequest, Sign $_{\text{Enrollment private key}}$ (<AuthnRequest>), UserID and Affiliation to TCA.

A SCENARIO FOR TCERT REQUEST AND TCERT RESPONSE (2 OF 3)

- TCA acquires the info from ACA database referenced by hash(Rand). TCA generates the TCert(s). [The TCA may generate one or more key agreement TCerts as well, if requested. Such key agreement TCert(s) can be provided to other users for potential incorporation into transactions.]
- TCA communicates the TCerts, [key agreement TCerts], TCertOwnerRootKey, and the (TCert- specific) K_TCert values to UA. K_TCert is used by UA during transaction creation, but is not passed by UA to SSP, if any.
- The TCA can potentially be configured to include hash(hash(K_TCert)) within the TCert, so that the UA can prove ownership of the TCert (by releasing hash(K_TCert, but not K_TCert)) to an entity authorized to handle transaction submission (in a way that is not available to an SSP). If this approach is taken, then transactions should not be constructed so as to make K_TCert available for retrieval (by the SSP) via a key agreement TCert owned by the transaction creator. Alternatively, the system can potentially enforce one-time- only submission of TCerts. If the UA and SSP functionalities are combined, this is not an issue.

A SCENARIO FOR TCERT REQUEST AND TCERT RESPONSE (3 OF 3)



SSP
UA
TCA
1
TLS handshake
2
Login
<AuthnRequest>
Sign Enrollment private key (<AuthnRequest>) || UserID || Affiliation
Rand
AuthnRequest that includes hash(Rand)
3
AuthnRequest || Sign Enrollment private key (<AuthnRequest>) || UserID || Affiliation
4
fetch info from ACA database
5
TCert(s) || TCertOwnerRootKey || K_TCert(s)

TRANSACTION PREPARATION AND SUBMISSION

- UA formulates transaction to be signed by SSP, and passes that as well as the TCert and TCertOwnerRootKey to SSP.
- SSP returns signature to UA. UA verifies against transaction using the TCert public key contained within the TCert. UA submits signed transaction to an authorized node for incorporation into blockchain.