Citadel Protocol Specification

Dusk Network

June 1, 2023

Contents

1 General Overview

1.1 What is Citadel

A Self-Sovereign Identity (SSI) protocol serves the purpose of allowing users of a given service to manage their identities in a fully transparent manner. In other words, every user can know which information about them is shared with other parties, and accept or deny any request for personal information.

`Citadel` is a SSI protocol built on top of Dusk Network. Users of a service can get a *license*, which represents their *right* to use such a service. In particular, `Citadel` allows for the following properties:

- **Proof of Ownership:** users can prove ownership of a license that allows them to use a given service.
- **Proof of Validity:** users can prove that a license has not been revoked and hence, it is a valid license.
- **Unlinkability:** activities from the same user in the network cannot be linked to each other by other parties.
- **Decentralized Session Opening:** when users use a license to open a session to use a service, everyone in the network learns that this happened, so it cannot be used again.
- **Attribute Blinding:** users can decide what information they want to share, hiding any other sensitive information and providing only the desired one.

1.2 Document Organization

In Section 2, we define all the object types and entities involved in the protocol. In Section 3, we roll out the protocol with full details.

2 Definitions

2.1 The Roles Involved

- **User:** an entity that interacts with the wallet to request licenses and prove ownership of those.
- **License Provider (LP):** an entity that receives requests for licenses, and upon acceptance, issues them.
- **Service Provider (SP):** the entity that provides a service upon verification that a service request is correct. The SP may be the same as the LP entity or a different one.

2.2 The Elements Involved

- **Request:** a request includes the encryption of a stealth address belonging to the user, where the license has to be sent to, and a symmetric key. The structure is as follows:

Element	Type	Info.
$(\mathsf{rpk}, R_{\mathsf{req}})$	StealthAddress	It is a request stealth address for the LP.
enc	PoseidonCipher[6]	It is the encryption of a license stealth address for the user and a symmetric key.
nonce	BlsScalar	Randomness needed to compute enc.

- **License:** a license is an asset that represents the right of a user to use a given service. The structure is as follows:

Element	Type	Info.
$(\mathsf{lpk}, R_{\mathsf{lic}})$	StealthAddress	It is a license stealth address of the user.
enc	PoseidonCipher[4]	It is the encryption of some user attributes and the signature of these attributes.
nonce	BlsScalar	Randomness needed to compute enc.
pos	BlsScalar	It is the position of the license in the Merkle tree of licenses.

- **LicenseProverParameters:** a prover needs some auxiliary parameters to compute the proof that proves the ownership of a license. Some of the items of this table are related to the session and session cookie elements. The structure is as follows:

Element	Type	Info.
lpk	JubJubAffine	The license public key of the user.
lpk'	JubJubAffine	A variation of the license public key of the user computed with a different generator.
$\mathsf{sig}_{\mathsf{lic}}$	Signature	The signature of the license attributes.
com_0^{hash}	BlsScalar	A hash of the public key of the LP.
com_1	JubJubExtended	A Pedersen commitment of the attributes.
com_2	JubJubExtended	A Pedersen commitment of the c value.
session_hash	BlsScalar	The hash of the public key of the SP together with some randomness.
sig_session_hash	dusk_schnorr::Proof	The signature of the session hash signed by the user.
merkle_proof	PoseidonBranch	Membership proof of the license in the Merkle tree of licenses.

- **Session:** a session is a public struct known by all the validators. The structure is as follows:

Element	Type	Info.
session_hash	BlsScalar	The hash of the public key of the SP together with some randomness.
session_id	BlsScalar	The id of a session open using a given license.
com_0^{hash}	BlsScalar	A hash of the public key of the LP.
com_1	JubJubExtended	A Pedersen commitment of the attributes.
com_2	JubJubExtended	A Pedersen commitment of the c value.

- **SessionCookie:** a session cookie is a secret value known only by the user and the SP. It contains a set of openings to a given set of commitments. The structure is as follows:

Element	Type	Info.
$\mathsf{pk}_{\mathsf{SP}}$	JubJubAffine	The public key of the SP.
r_{session}	BlsScalar	Randomness for computing the session hash.
session_id	BlsScalar	The id of a session open using a given license.
$\mathsf{pk}_{\mathsf{LP}}$	JubJubAffine	The public key of the LP.
attr	JubJubScalar	The attributes of the user.
c	JubJubScalar	The challenge value.
s_0	JubJubScalar	Randomness used to compute com_0^{hash}.
s_1	BlsScalar	Randomness used to compute com_1.
s_2	BlsScalar	Randomness used to compute com_2.

3 Protocol Workflow

In Citadel, each party involved in the protocol keeps static keys, as we detail now. Let $G, G' \leftarrow \mathbb{J}$ be two generators for the subgroup $\mathbb{J}$ of order t of the Jubjub elliptic curve. The keys of each party are the following.

- *Secret key:* $\mathsf{sk} = (a, b)$, where $a, b \leftarrow \mathbb{F}_t$.
- *Public key:* $\mathsf{pk} = (A, B)$, where $A = aG$ and $B = bG$.

The workflow of the Citadel protocol is depicted in Figure 1, and described with full details as follows.

1. (**user**) request_license : compute a license stealth address $(\mathsf{lpk}, R_{\mathsf{lic}})$ belonging to the user, using the user's own public key, as follows.

   (a) Sample r uniformly at random from $\mathbb{F}_t$.

   (b) Compute a symmetric Diffie–Hellman key $\mathsf{k} = rA_{\mathsf{user}}$.

   (c) Compute a one-time public key $\mathsf{lpk} = H^{\mathsf{BLAKE2b}}(\mathsf{k})G + B_{\mathsf{user}}$.

   (d) Compute $R_{\mathsf{lic}} = rG$.

   Compute also an additional key $\mathsf{k}_{\mathsf{lic}} = H^{\mathsf{Poseidon}}(\mathsf{lsk})G$, by computing first the license secret key $\mathsf{lsk} = H^{\mathsf{BLAKE2b}}(\mathsf{k}) + b_{\mathsf{user}}$. Then, compute the request stealth address $(\mathsf{rpk}, R_{\mathsf{req}})$ using the LP's public key, as follows.

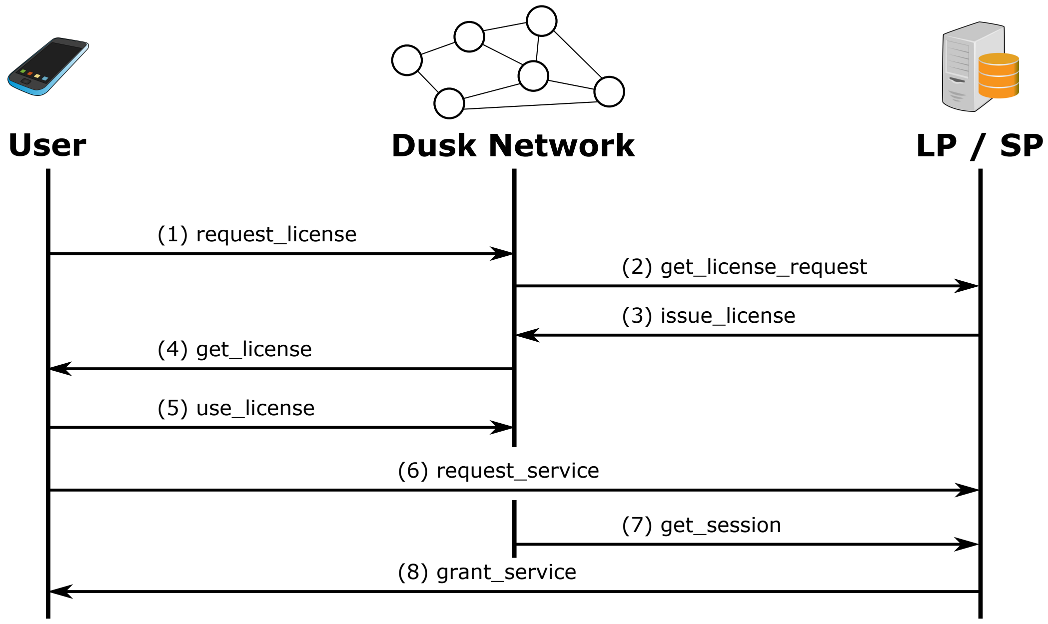

Figure 1: Overview of the protocol messages exchanged between the user, the Dusk Network, and the SP.

(a) Sample r uniformly at random from $\mathbb{F}_t$.

(b) Compute a symmetric Diffie–Hellman key $\mathsf{k}_{\mathsf{req}} = rA_{\mathsf{LP}}$.

(c) Compute a one-time public key $\mathsf{rpk} = H^{\mathsf{BLAKE2b}}(\mathsf{k}_{\mathsf{req}})G + B_{\mathsf{LP}}$.

(d) Compute $R_{\mathsf{req}} = rG$.

And finally send the following request to the network:

$$\mathsf{req} = ((\mathsf{rpk}, R_{\mathsf{req}}), \mathsf{enc}, \mathsf{nonce}),$$

where

$$\mathsf{enc} = \mathsf{Enc}_{\mathsf{k}_{\mathsf{req}}}((\mathsf{lpk}, R_{\mathsf{lic}})||\mathsf{k}_{\mathsf{lic}}; \mathsf{nonce}).$$

2. (**LP**) get_license_request : continuously check the network for incoming license requests, by checking if $\mathsf{rpk} \stackrel{?}{=} H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{req}})G + B_{\mathsf{LP}}$, where $\tilde{\mathsf{k}}_{\mathsf{req}} = a_{\mathsf{LP}} R_{\mathsf{req}}$.

3. (**LP**) issue_license : upon receiving a request from a user, define a set of attributes attr representing the license, and compute a digital signature as follows:

$$\mathsf{sig}_{\mathsf{lic}} = \mathsf{sign\_single\_key}_{\mathsf{sk}_{\mathsf{SP}}}(\mathsf{lpk}, \mathsf{attr}).$$

Then, send the following license to the network:

$$\mathsf{lic} = ((\mathsf{lpk}, R_{\mathsf{lic}}), \mathsf{enc}, \mathsf{nonce}, \mathsf{pos}),$$

where

$$\mathsf{enc} = \mathsf{Enc}_{\mathsf{k}_{\mathsf{lic}}}(\mathsf{sig}_{\mathsf{lic}}||\mathsf{attr}; \mathsf{nonce}).$$

4. (**user**) get_license : receive the license by scanning the incoming transactions, and checking if $\mathsf{lpk} \stackrel{?}{=} H^{\mathsf{BLAKE2b}}(\tilde{\mathsf{k}}_{\mathsf{lic}})G + B_{\mathsf{user}}$, where $\tilde{\mathsf{k}}_{\mathsf{lic}} = H^{\mathsf{BLAKE2b}}(\mathsf{lsk})G$.

5. (**user**) use_license : when using the license, open a session with a specific SP by executing a call to the license contract. The following steps are performed:

   - The user issues a transaction that calls the license contract, which includes a ZKP that is computed out of the gadget depicted in Figure 2. Notice that here, the user signs session_hash using lsk. Likewise, the user here will need to compute $\mathsf{lpk}' = \mathsf{lsk}G'$.

- The network validators will execute the smart contract, which verifies the proof. Upon success, the following session will be added to a shared list of sessions:

$$\mathsf{session} = \{\mathsf{session\_hash}, \mathsf{session\_id}, \mathsf{com}_0^{hash}, \mathsf{com}_1, \mathsf{com}_2\},$$

  where $\mathsf{session\_hash} = H^{\mathsf{Poseidon}}(\mathsf{pk_{SP}}||r_{\mathsf{session}})$, and r_{session} is sampled uniformly at random from $\mathbb{F}_t$.

6. (**user**) $\mathsf{request\_service}$: request the service to the SP, establishing communication using a secure channel, and providing the session cookie that follows.

$$\mathsf{sc} = \{\mathsf{pk_{SP}}, r_{\mathsf{session}}, \mathsf{session\_id}, \mathsf{pk_{LP}}, \mathsf{attr}, c, \mathsf{s}_0, \mathsf{s}_1, \mathsf{s}_2\}$$

7. (**SSP**) $\mathsf{get\_session}$: receive a $\mathsf{session}$ from the list of sessions, where $\mathsf{session.session\_id} = \mathsf{sc.session\_id}$.

8. (**SSP**) $\mathsf{grant\_service}$: grant or deny the service upon verification of the following steps:

   - Check whether the values $(\mathsf{attr}, \mathsf{pk_{LP}}, c)$ included in the sc are correct.
   - Check whether the opening $(\mathsf{pk_{SP}}, r_{\mathsf{session}})$ included in the sc matches the $\mathsf{session\_hash}$ found in the $\mathsf{session}$.
   - Check whether the openings $((\mathsf{pk_{LP}}, \mathsf{s}_0), (\mathsf{attr}, \mathsf{s}_1), (c, \mathsf{s}_2))$ included in the sc match the commitments $(\mathsf{com}_0^{hash}, \mathsf{com}_1, \mathsf{com}_2)$ found in the $\mathsf{session}$.

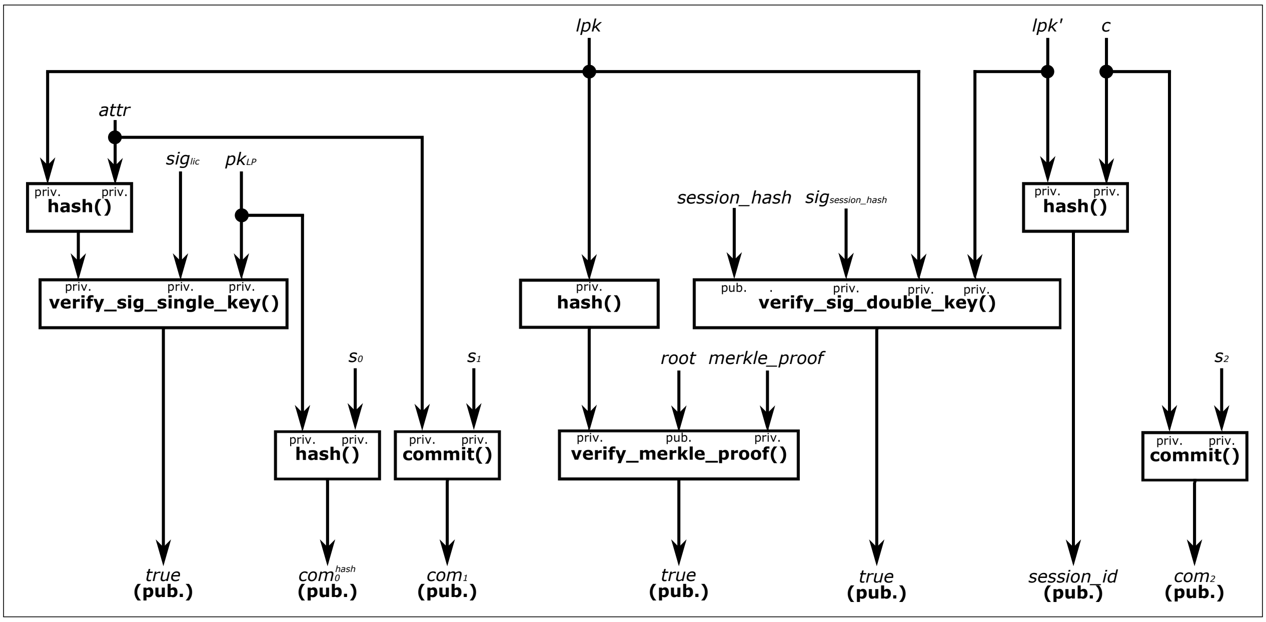

Figure 2: Arithmetic circuit for proving a license's ownership.

Furthermore, the SP might want to prevent the user from using the license more than once (e.g. this is a single-use license, like entering a concert). This is done through the computation of $\mathsf{session\_id}$. The deployment of this part of the circuit has two different possibilities:

- If we set $c = 0$ (or directly remove this input from the circuit), the license can be used only once.
- If the SP requests the user to set a custom value for c (e.g. the date of an event), the license can be reused only under certain conditions.

4 Protocol Implementation

4.1 Participants

Protocol implementation involves realization of the protocol building blocks as well as providing means of data communication between them. Building blocks are placed at the following locations, which correspond to protocol participants:

- User software.
- License Provider software.
- Service Provider software.
- Citadel contract.

Data communication between protocol participants is realized in the following communication modes:

- Transactions which change the contract state.
- Queries which do not change the contract state.
- Storing data directly in blockchain by issuing transactions.
- Retrieving data from blockchain by scanning transactions.
- Delegating ZK proof calculation.
- Off-chain calls.

The following diagram illustrates an interaction between protocol participants and indicates the communication means used.

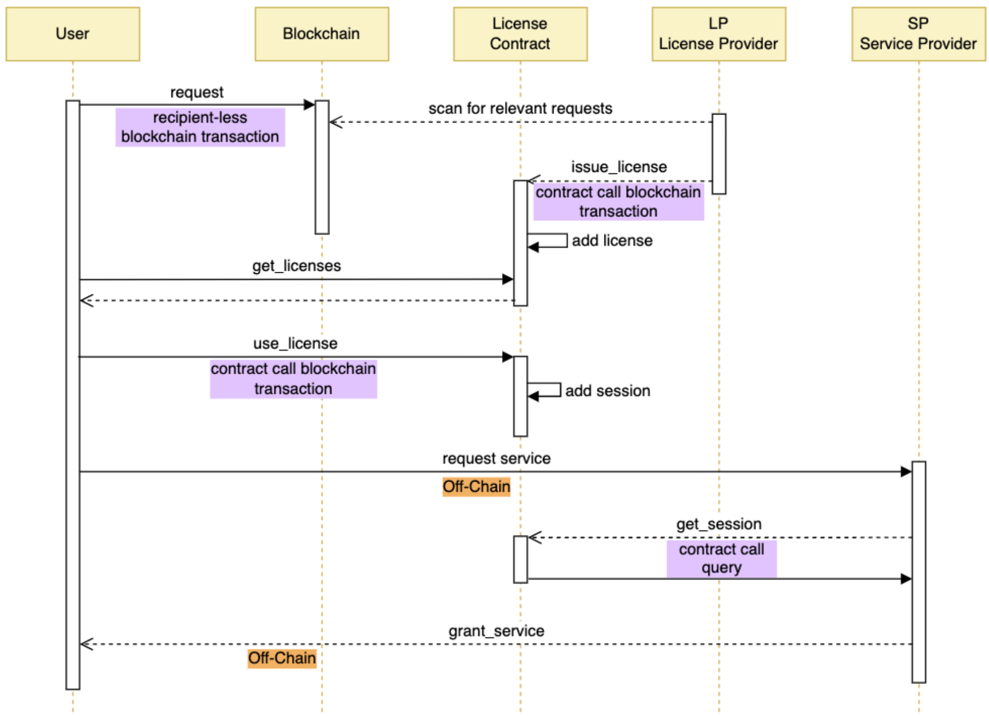

Figure 3: Interaction between protocol participants

On the diagram, we can see various communication modes being used. Initially, user submits recipient-less transaction which contains a request as payload into the blockchain. Subsequently, License Provider can scan the blockchain for transactions containing relevant requests. License Provider can obtain requests via other routes as well, for example via http or email, passing requests on blockchain is only one of possible ways. Once License Provider gets a hold of a request, it can perform appropriate verification and issue a license. Issuing a license involves another mode of communication - a smart contract transaction call. Smart contract transaction call is also a blockchain transaction, yet to not to confuse the reader, we show it on the diagram while omitting the detail of blockchain involvement. User obtains licenses using a mode of communication based on a contract query. For privacy reasons, user obtains a bulk of information not pertaining exclusively to her, and she filters it out by herself. To economize the volume of data transfers, block-height range is also passed to allow the transfer to include only a subset of available records. All modes of communication used so far were on-chain. Communication between user and ServiceProvider, on the other hand, is off-chain. When Service Provider wants to establish a session, it calls a contract to obtain session for a given session id.

The diagram illustrates the following flow of data and interactions between participants:

- User submits request to a License Provider by issuing blockchain transaction.
- License provider scans blockchain for requests and obtains the request.
- License provider, upon necessary verification, issues a license.
- License provider sends the license to License Contract via a smart contract call transaction.
- User obtains licenses for a given block-height range.
- User filters out licenses addressed to her/him.
- User calculates a proof (the proof calculation might be delegated).
- User calls *use-license* to redeem a license, via a smart contract call.
- License Contract attempts to verify the proof and, if verified, adds a new session to a list of sessions.
- User requests a service from a Service Provider (off-chain).
- Service Provider asks contract for a session.
- Service Provider grants service to the user (off-chain).

4.2 License Contract

License contract maintains state consisting of the following data:

- List of sessions.
- Map of licenses and their positions in the Merkle tree.
- Merkle tree of license hashes.

Contract provides the following methods:

- *issue-license*
- *get-licenses*
- *use-license*
- *get-session*

issue-license adds a license to a Merkle tree of licenses. *get-licenses* provides a list of new licenses added in a given block-height range *use-license* attempts to verify the proof and, if verified, adds a new session to a list of sessions, nullifies the license in the Merkle tree. *get-session* finds a session in a list of sessions and returns it to the caller.

4.3 License ZK Proof Calculation

License ZK proof calculation will either be performed by user or delegated to node. At the time of writing the details of delegation are not known yet, but they will be filled in here once the information becomes available.