PRISM protocol v0.3

Atala PRISM team - IOG

Abstract

This document describes the current state of the protocol that supports PRISM. PRISM is a framework for the management of decentralized identifiers and verifiable credentials.

Contents

1 Introduction and protocol description

In this section we will informally describe a protocol to create and manage DIDs (**D**ecentralized **Id**entifiers), that also allows to manage the creation, revocation and presentation of verifiable credentials.

1.1 DIDs management

For simplicity, we will define a DID as a string identifier of the form

$$did{:}prism{:}\langle identifier\rangle$$

Each DID is associated to a list of public keys. The list can be updated over time by adding and revoking keys. Each key has an assigned role during its lifetime. The three possible roles we describe are:

issuing keys used to issue credentials on behalf of the DID.

revocation keys used to revoke credentials on behalf of the DID.

master keys used to add or revoke other keys associated to the DID.

Our construction for DID management relies on an underlying blockchain. The blockchain allows us to publish transactions with sufficiently large metadata. We use the blockchain for both; data distribution along parties, and for consensus related to the order of relevant events in our protocol. All the parties participating in our protocol are running a blockchain node that reads the metadata of the blockchain transactions. When they find a protocol event, they process it to construct their view of the system.

In the following sub-sections we describe the events in our protocol related to DID management.

1.1.1 DID creation

In order to create a DID, a user will follow the steps below:

- The user generates a desired list of public keys. He will associate each key to a key identifier and a role. There must be at least one key with the master role in this initial list.
- The resulting list is encoded and hashed producing an encoded *initial_state* and a *hash*.
- The *hash* is encoded in hexadecimal form producing an *encoded_hash*.
- The user's DID is constructed as *did:prism:encoded_hash*.
- The user will a signed *CreateDID* event in the transaction metadata that is submitted to a blockchain. The event contains the list of public keys with their identifiers and roles. The signature will be performed using the private key associated to any of the master keys in the initial list of keys.

Once the transaction is added to the blockchain with a sufficient number of confirmations d (i.e. the block containing the transaction has d blocks appended after it on the underlying chain), all the participants following our protocol will validate the signature of the *CreateDID* event. After verification, they will register the DID as *published* along with all the keys posted in the initial list. The keys will be considered valid since the time associated to the event carrier transaction. This is a timestamp generated by the blockchain.

Optionally, while waiting for blockchain confirmation, the user can also use the associated DID

$$did{:}prism{:}encoded\_hash{:}initial\_state$$

This DID, called *long form* or *unpublished* DID will not be recognised as *published* by other parties in the protocol. However, other users would be able to verify that the list of keys encoded in *initial_state* corresponds to the DID *did:prism:encoded_hash*. The recipient of an unpublished DID needs to query the state of the *short form* of the DID (the prefix before "*:initial_state*") to check for changes in the list of keys.

1.1.2 DID update

Updating a DID means that the user controlling a master key associated to it, will add new or revoke existing keys to the list associated to the DID. In order to update the list of keys, the user will:

- Create the list of key identifiers that the user wants to revoke from the current state of his DID.
- Create a list of keys he wants to add to the list of keys associated to his identifier. This keys must be associated to a role and a fresh key identifier.
- Create an *UpdateDID* event that contains the two previously generated lists. The event also carries a hash of the last *UpdateDID* performed over the DID (or the hash of the *CreateDID* if this is the first update)
- Sign the event with one of the *currently associated* master keys of the DID, and publish the event inside the metadata of a transaction.

When the event added to the blockchain has enough confirmations, all parties will process the event. This is, they will validate the signature, and update their internal knowledge of the updated DID. The newly added keys and the revoked keys will be timestamped using the time of the carrier transaction.

With these two events we have shown how to create and update DIDs.

1.1.3 Obtain the list of keys associated to a DID

Now, we can define the process that a protocol participant must follow to obtain the keys associated to a DID. We mentioned that a DID can be presented in long form (or as an unpublished DID), or in its short form. Hence, we will describe the process in the two cases.

In order to obtain the keys associated to a DID in short form, we simply read the state we have constructed so far by processing the *CreateDID* and *UpdateDID* events we read from the blockchain. The list of keys associated to the DID, is the one we see in our internal state. If we do not find the DID as a published one in our state, we return a *DID unknown* response.

Now, if the DID we receive is in long form, we first verify that $hash(initial\_state) = initial\_hash$. If that check fails we reply with an *Invalid DID* response. If the check passes, we extract the short form of this DID (the prefix that ends right before the last *":"*), and we check the list of keys as described in the previous paragraph. If the result of this process is a *DID unknown* response, then we decode the list from the *initial_state* suffix and return this list as a result.

With this process we complete the events and operations related to DIDs. Lets now move into the events related to issue and revoke credentials.

1.2 Verifiable credentials

Now that we have DIDs, we can proceed to the creation and revocation of credentials. A *credential* to us is a set of *claims*. Each claim is represented by a property-value pair, e.g. $(name, John)$. Credentials are created by *issuers* to *subjects*. Both issuers and subjects are represented with their corresponding DIDs.

For practical reasons, we assume that issuers want to issue credentials in batches. This is, an issuer would like to create multiple credentials at once. Below we describe the steps to issue a batch of credentials.

1.2.1 Credentials batch issuance

Lets us refine first what a credential is in the context of this document

Credential a credential is a JSON document that contains three key-value maps.

- the "issuer" key represents the issuer DID.
- the "keyId" key represents the key identifier associated to the issuer DID that was used to sign the credential.
- the "claims" key represents claims that the issuer makes about the subject. The claims are grouped in a JSON object. There is a key for each claim asserted by the issuer. One particular claim is the subject DID, that represents a DID controlled by the subject of the credential.

In the following steps, we assume the issuer and subjects have already established a secure communication channel. We also assume that the issuer has a published DID that will be used to represent him in credentials. In order to issue a batch of an arbitrary number N of credentials, the issuer will:

- Ask to each subject the DID they would like to use for the credential they will receive.
- The issuer creates the credentials for each subject. It uses his published DID and the key id of an *issuing key* to populate the credential corresponding fields. On each credential, it adds the corresponding subject DID as one of the claims.
- The issuer encodes each individual credential using a base64URL encoding.
- The issuer signs each individual encoded credential with an *issuing key* associated to his publicly known DID. The signing key corresponds to the key id in the credentials' claims.
- The issuer encodes each signature, and concatenates them to their corresponding credentials using a dot (".") as separator. This produces strings of the form $\langle encoded\_credentials \rangle.\langle encoded\_signature \rangle$. We call these strings *signed credentials*.
- Now, the issuer takes all the signed credentials, and computes a merkle root from them.
- The issuer creates an *IssueBatch* event that contains the merkle root and issuer DID, and signs the event with an *issuing key* associated to this DID. **Note**: Today we use the same issuing key for the event signature and the individual credentials signature, but those could be different keys.
- The issuer attaches the signed event to the metadata of a transaction and sends it to the blockchain.
- The issuer gives to each subject their corresponding signed credential along with their associated merkle inclusion proof.

Likewise previous protocol events, all parties will process the transaction once confirmed, validate the event signature and timestamp the merkle root with the time associated to the carrier transaction.

Note that, even though the issuer DID *must be published*, subjects' DIDs can remain unpublished.

1.2.2 Credentials revocation

In order to revoke a credentials, issuers have two alternatives:

1. They revoke all the credentials in a batch. This is done by signing a *RevokeBath* event with a *revocation* key associated to the issuer DID. The signed event contains the merkle root to revoke.
2. They revoke specific credentials associated to a batch. This is done by signing a *RevokeCredentials* event with a *revocation* key associated to the issuer DID. The signed event contains both the merkle root associated to the credentials to revoke, and the hashes of the specific credentials to revoke.

In both variants, the event is published on-chain and processed by all the participants. The participants will timestamp the new information with the carrier transaction time.

1.3 Credential presentation and verification

Once they receive credentials from issuers, subjects will present them to interested parties, called *verifiers*. For example, a student may receive a verifiable credential from a university, and would like to present his credential to a potential employer. In our setting, the verifier will be a party following our protocol events from the blockchain. The steps to present and verify a credential are the following:

- We assume a safe communication channel between the subject and verifier. We also assume that the channel can be identified by an identifier *ch*.
- The subject shares his credential and merkle inclusion proof to the verifier.
- The verifier then:
  - computes the merkle root from the inclusion proof and the credential hash;
  - extracts the issuer DID and signing key id from the credential claims,

- retrieves from the state he computed from the blockchain, the timestamps associated to the merkle root and the issuing key,
- checks if the merkle root or credential hash has been revoked
- validates the issuer signature on the credential, and determines if it was signed at a time when the issuer key was valid.

Note: The signature of the IssueBatch event has already been verified by the protocol participants at the time of batch publication.

- At this point, the verifier knows that the credential was properly signed. Now, he shares a nonce to the subject and asks him to sign it with a key associated to his DID.
- The subject signs the hash of $nonce||ch$ and returns the signature to the verifier.
- The verifier now checks the signature and concludes that the credential subject is indeed the person presenting it.

2 Further thoughts

2.1 Increasing scalability

In the previous sections we described a protocol that allows transparent management of decentralized identifiers and credentials. The protocol usage of a blockchain facilitates the solution to three points:

- Events ordering: All participants can agree on the order in which any pair of events occurred
- Fork protection (safety): There is no point in time where a valid sequence of processed protocol events could become invalid, nor replaced by another sequence
- Data transmission: All participants receive the same sequence of protocol events as they are transmitted through the blockchain

The second point has particular relevance to guarantee that the history of events can be trusted. For instance, imagine Alice controls a DID (its associated master keys) that represents a real world object, like a car. If Alice's wants to sell the car and transfer the corresponding DID, she would perform a DID update that revokes all the master keys controlled by her, and adds a new master key controlled by the new car owner. If Alice could later "undo" this update, the DID ownership transfer could not be trusted.

The properties brought by the blockchain come at a cost. Namely, the protocol throughput (number of events that participants can attach in transactions per unit of time) is bounded to the blockchain throughput. In order to scale the throughput, we have reviewed Sidetree's [1] approach. The high level idea behind Sidetree is to not publish individual events attached to blockchain transactions, but to publish a hash-link to an off-chain content addressable storage (CAS), like IPFS. The CAS will contain the file referenced by the hash-link, which contains a batch of many protocol events (allowing to transcend the size limits imposed by blockchain transactions).

We identify a non-ideal drawback about this approach, which is a loss of the safety property. In the setting defined by Sidetree, participants can control changes to the past of their identifiers. This is, if Alice desires, she could create a DID by posting a file, F_1 on the CAS, and its hash on-chain, F_1 would contain Alice $CreateDID$ event. Later, Alice can create a file F_2, containing an $UpdateDID$ event, post the hash of F_2 on-chain, but intentionally not posting F_2 on the CAS. Then, Alice can create a third file F_3 which contains an new $UpdateDID$ event that would be invalid if F_2 were revealed in the CAS (e.g. F_2 event could revoke the key that signs the update in F_3). However, Alice can post both F_3 in the CAS and the hash of F_3 on-chain. In Sidetree, this sequence of actions would lead all protocol participants to believe that Alice's DID is in certain state, produced by the $CreateDID$ of F_1 and the $UpdateDID$ from F_3.

Alice has the power to reveal at any point F_2, making the update from F_3 as invalid, and forcing all participants to update Alice's DID to the state reflected by F_2. This is known as **late publish attack**, and makes technically impossible to trust the past history history of events in Sidetree implementations. As an example consequence, it is not possible to transfer ownership of a DID when using Sidetree.

In order to avoid introducing this issue, we considered two options.

[1] https://identity.foundation/sidetree/spec/

The first one is to add a permissioned actor (or a federation of them) that are allowed to batch protocol events and publish both on-chain hash-links, and files in corresponding CAS. The special actor introduces a trust model that assumes that it will always reveal files. If the actor fails to reveal a file, the protocol participants stop processing further batches until the missing file is revealed. This is not possible in Sidetree because any participant would be able to freeze the system by not revealing a file.

This protocol variation would still allow users to publish events on-chain directly, leaving space for some decentralization for those participants that do not want to depend on the centralized batcher.

In order to use the batcher to publish the $UpdateDID$ events associated to a DID D, the owner of D will have to declare on-chain that future updates for D will be found on the off-chain batches. This public declaration is needed to avoid race conditions when a hash is on-chain but a file is not revealed.

A second option consist of a more decentralized variation of the previous approach. Namely, allow any participant to propose itself as an event batcher. The proposal is performed by submitting an on-chain event declaring the batcher DID. Every user that would like to batch its events off-chain should publicly notify the batcher they will use. Only one batcher can be assigned to each DID, this is, the $UpdateDID$ events associated to a DID will be published by at most one batcher. The objective is once again, to be able to identify missing files in a sequence, allowing participants to stop processing files out of order.

In any of the approaches, if the associated batcher stops publishing files, the DID registered to the batcher is "frozen" until the file is revealed. We could re-introduce some liveness protection to DID owners by requesting batches from a same batcher to be separated at least N underlying blockchain blocks, in order to provide that time for users to send a $Contention$ event that invalidates the events associated to a DID published in the previous batch. During the processing of batches, participants will wait the contention period (the N blocks) before applying events from a batch in order to not apply the updates to contended DIDs.

Notes:

- we could change the $CreateDID$ event to incorporate an optional batcher DID from start, the batcher DID would be part of the initial state, bounding the batcher to the DID suffix.
- we could support events to de-register from a batcher, and to switch batcher too.
- a user is free to batch his own DIDs' related events. This could be useful for a case like a car manufacturer that would like to batch all DIDs associated to their cars, allowing car owners to de-register from a batcher at will.
- a priori, we do not see a need to batch credential issuance/revocation events

At the time of this writing, we haven't implemented any of the above approaches. We had, however, implemented on-chain batching, which allows to submit multiple events in a single transaction. The events are processed in order per transaction. This has been helpful in use cases where a multiple events coming from different entities, are submitted to the same PRISM node to publish them.