Verifying Credentials

Suggest Edits

As users collect more credentials, having to manually manage them (figuring out which are current, and which are appropriate for a given relying party) becomes infeasible. Unlike unstructured credentials (e.g., many .pdf files), which would require a user to manually select which data to share, verifiable credentials lend themselves to credential exchange protocols that are easier for end-users. These protocols are typically implemented for the user through the user's wallet (or other software agents) interacting with the requesting party.

Verite uses the concepts and data models from DIF's Presentation Exchange for this purpose. This document describes how a consumer of Verifiable Credentials, referred to as a verifier or relying party, informs users what types/formats of credentials they accept, and how the user's wallet/agent uses this information to select the appropriate credentials and respond. Note that what follows assumes a wallet that controls a fully-featured DID like did:ion rather than a crypto-wallet; the latter has a slightly different trust model and identity-binding model, which leads to different requirements for Verifiable Presentation (i.e. proving liveness and consent at time of credential exchange).

Presentation Requests and Definitions

A presentation definition is the way a relying party describes the inputs it requires, proof formats, etc. A presentation request is a generic term for a transport conveying this. It's meant to be flexibly embedded in a variety of transports, such as OIDC or WACI . Verite uses a JSON object that somewhat resembles the schema defined by WACI, but with additional fields including a challenge and reply URL.

Wallet Interactions

Assuming a mobile wallet stores the credentials, for the convenience of the user a verifier may initiate the process of sending the presentation request either by scanning a QR code (desktop) or a deep-link (mobile). Due to size limitations of a QR code, wallet and credential interactions often do not include the full presentation request in the QR code; instead the QR code encodes an endpoint with a unique URL. The wallet decodes the QR code, subsequently retrieving the presentation request from that endpoint.

See example Verite Presentation Request

Credential Submission

The wallet parses the Presentation Definition object to determine what types of inputs, proofs, and formats the verifier requires. The wallet displays a summary of the information requested to the wallet holder, asking for approval and/or asking the user to select the desired credential(s) from the set of matches. On confirmation, the wallet gathers the credentials and creates a verifiable presentation containing the credential and signs the presentation with the credential subject's private key. It embeds the VP in a presentation submission , and signs it along with the challenge to provide proof of identifier control .

Finally, the wallet sends the packaged credential to the reply_url contained in the presentation request.

See example Verite presentation submission .

Verification

The verifier receives the presentation submission, unwraps it, and maps the presentation to the original presentation request. Mapping the submission to the original request can be done in many ways. The Verite demos use a JWT in the reply_url to store the mapping. Next, the verifier verifies the submitted contents against the required inputs, ensures its signed by the subject's keys, and checking the credential's status to determine if it is revoked out not.

Verification cannot always occur immediately. In these cases, the presentation request has an optional status_url that can be used to check its status.

There is no required output or side-effect of verification. However, we have a pattern for integrating with Ethereum that models how an on-chain Verification Registry can capture verification events in a way that's easy to consume on-chain. A web app that is not constrained by on-chain data availability, however, might simply update its state and allow the user to continue some action.

Verification Flow

At a high level, the verification flow for an identity wallet looks like this:

However, when this maps to an on-chain use-case, where Dapps are requiring off-chain documentation to trust a crypto address, this flow can get a lot more complicated depending on the types of credentials and the identity-proofing they require.

In what follows, we will zoom in one two very different applications of this high-level flow that come from two different architectures: first, we will break down the flow for credentials issued to a user-controlled "identity wallet", then, we will break down the flow for address-specific credentials (signed over by only a crypto-currency wallet), showing the differences in both procedure and trust model.

Wallet-Bound Verification Flow

In this example, a user wants to verify credentials issued to, and stored in a mobile app that function as an "identity wallet," i.e., a wallet controlling a DID for identity applications rather than a "cryptocurrency wallet" controlling a keypair for onchain/payment purposes. (Note, some wallets combine both sets of capabilities, but for simplicity's sake, our sample implementation treats them as distinct and independent applications.)

Some things to note:

- The QR code is provided as a way of connecting a browser-based dApp interaction with a mobile-based identity wallet; different architectures (i.e., browser-based identity wallets without the cross-device requirement) can use different mechanisms to bootstrap the wallet-verifier relationship.
- Verifier prompts user for the Ethereum address the Verification Record will be bound to
- User provides their Ethereum address (e.g. copy pasting, or by connecting a blockchain wallet)
- Verifier generates a JWT that encodes the user's address, that will later be used to generate the URL the mobile identity wallet will submit to.
- Verifier shows QR Code
- User scans QR Code with their mobile identity wallet.
- Identity wallet parses the QR code, which encodes a JSON object with a
- challengeTokenUrl
- property.
- Wallet performs a GET request at that URL to return a Verification Offer, a wrapper around a Presentation Request
- , with three supplementary properties:* A unique identifier (such as a UUID) or other logging metadata.
    - The verifier's unique identifier, i.e. it's "DID" (including offchain DIDs like
    - did:pkh
    - ,
    - did:web
    - or
    - did:key
    -)
    - A URL for the wallet to submit the Presentation Submission
    - , using the unique JWT generated earlier.
- The identity wallet prompts the user to select credential(s) from the set of matches.
- Identity wallet prepares a Presentation Submission
- including:* The wallet's DID, control of which is proven by a signature on the submission object by that DID's private key. In the Verite examples, the holder's DID must match the
    - credentialSubject.id
    - s of the presented VCs, thus verifying both liveness and control of the identifier against which the credentials were issued.
    - Any Verifiable Credential(s) necessary to complete verification.
    - A fresh signature over the above (and the challenge/nonce provided by the verifier).
- Wallet submits the Presentation Submission to the URL found in the Verification Offer (
- reply_url
- property).
- The Verifier validates all the inputs

- Verifiers generates a Verification Record and adds it to the registry or sends it directly to a waiting relying party

Address-Bound Verification Flow

In this example, a user wants to verify address-bound credentials issued to a blockchain address. Since these credentials simplify the trust model, they only need a fresh, live signature from the [cryptocurrency] wallet's private key to authenticate the session in which those credentials are presented, rather than a signature over the credential themselves. The credentials do not necessarily need to be stored in an identity wallet or even in a crypto wallet, since the crypto wallet will need to be authenticated to present them and they are already tamper-proofed.

1. Crypto Wallet requests transaction or resource requiring verification.
2. Relying Party (e.g. Dapp) calls Verifier with wallet address to be verified.
3. Verifier returnsPresentation Request
4. (no additional properties required).
5. Verifier prompts user to connect the wallet controlling the blockcain address that the Verification Record will be bound to.
6. Wallet signs an offchain transaction to authenticate wallet and authorize a web3 session. For interoperability and auditing purposes, we recommend aEIP-4361
7. -conformant "Sign-In With Ethereum" message or aCAIP-122
8. -conformant equivalent.
9. Verifier sends itsPresentation Request
10. to wallet, or equivalent bespoke RPC calls describing the acceptable credentials.
11. The Wallet checks its storage for one or more matching credentials. If more than one is present, a user selection step should be triggered; if none, a redirect or informative message displayed. If exactly one, consent step may be optional, depending on use-case.
12. Wallet prepares aPresentation Submission
13. , or if needed, the Verifier can assemble it on behalf of the wallet; this submission object includes:
14. - Any Verifiable Credential(s) necessary to complete verification.
15. - Wallet key--or ephemeral session key
16. - --signs presentation submission object (with challenge) to create verifiable and replayable event for logging and audit purposes
17. - If session key rather than wallet key signed the submission, and/or if any additional (on-chain or off-chain) software was involved in the storing, fetching, and passing of the VC(s), then aCACAO
18. - receipt of sign-in message needs to be included for the submission to be functionally equivalent to a submission from a monadic DID wallet. Note: aCACAO
19. - receipt contains a replayable authorization of a session key and any other resource expressable as a URI.
20. Wallet or Verifier submits the Presentation Submission to the URL found in the Verification Offer (
21. reply_url
22. property).
23. The Verifier validates all the inputs
24. Verifiers generates a Verification Record and adds it to the registry or sends it directly to the relying party which is awaiting a record for that wallet address (see step 1). Transaction or resource request may be attached as well to process, per use case. Updated3 months ago
25. Table of Contents
26.
27.
28.
29.
30.
31.
32.