Integrating Via WalletConnect

Suggest Edits

Introduction

One of the core difficulties in the web3 stack is that signing mechanisms and wallet-dapp interfaces are notoriously understandardized, and in many case numerous competing standards and interfaces persist for years, leading to a splintered user experience and a piecemeal security model. On top of adding friction and heightened risk of end-user error, requiring users to identify their wallets and click different buttons to go through different flows per-wallet is also of an anti-pattern in privacy terms, since it makes"fingerprinting" accounts substantially easier with just a few data points or analytics primitives.

WalletConnect, a relay network built on open standards incubated and specified at theChain Agnostic Standards Alliance , is often thought of as a convenience for developers of dapps who don't want to integrate wallet interfaces one-by-one, or a way for wallets to pick up "back-end" capabilities without having to stand up their own end-to-end web2 routing and security infrastructure. It establishes a websocket interface mutually between wallets and dapps, which both abstracts out some of the HTTPS complexity, but also creates herd privacy in making a large portion of today's web3 traffic over HTTPS wires uniform and more opaque. Better living through standardization!

One of the most vexing problems in the adoption of verifiable credentials in the web3 space has been the chicken-and-egg problem of VC exchange protocols-- few wallets have been willing to stick their neck to dogfood a new protocol with new attack vectors and liabilities. Add to this the lack of demand-- where are the applications demanding a specific form of verifiable credentials or a specific protocol for issuing and presenting them?

Verite is happy to sidestep both of these traditional "cold-start" problems by introducing a "minimum viable protocol" through WalletConnect, inheriting all of its privacy and security features so that all wallets and dapps supporting WalletConnect V2 can add on support for verifiable credentials with two tiny code-changes.

Feature Discovery

WalletConnect V2 introduces a lot of scaffolding for multi-chain and multi-Virtual-Machine features on both wallets and decentralized applications. This presents a similar cold-start problem-- how to let dapps and wallets communicate to each other their features and interfaces efficiently in-band? The solution lies in adding a rich and expressive "feature discovery" step to the wallet authorization flow, known to end-user as "the connect wallet button".

At a high level, a dapp finds out at time of connection which features (expressed as RPC methods and RPC notifications) the wallet supports and grants the dapp access to, by requesting authorization and receiving back granular permissions. This feature-discovery method is idempotent, and can be called again and again in the background of a session to enable progressive authorization flows, which are increasingly being promoted as best practice by browser security experts and protocol designers.

Simply put, a dapp asks a wallet: "Can we exchange VCs? Are you willing to store VCs I give you?" A wallet can answer, "No", "Yes and yes", or "Yes and maybe later". The dapps can always ask again later after establishing trust in other ways. Feature discovery basically occurs as a back-and-forth negotiation of scope objects expressed as JSON arrays, since the Ethereum ecosystem and most other major blockchains make extensive structural use of JSON-RPC for packets between dapps, wallets, and nodes. These scope objects bear some similarity to the JSON scope objects used by theRich Authorization Request (RAR) system in some contemporary Web2 protocols; RAR is also a building-block of theGNAP protocol. That said, they are also very much tailored to the cross-chain community research atCASA , which maintains a registry ofnamespaces to make cross-chain engineering adoptable and ergonomic.

The protocol and the syntax for this object-based negotiation are specified in one of the Chain-Agnostic Improvement Proposals,CAIP-25 ; optional extensions to the protocol are specified as independent CAIPs likeCAIP-171 andCAIP-211 . This lightweight JSON-RPC-based protocol outlines a flow where a dapp can prompt a wallet to factor in policy and/or user input in a granular authorization of wallet features and networks in the browser. This protocol is specified more abstractly than how it is implemented in the WalletConnect relay network, since parity is maintained with a relay-free implementation used by browser-embedded (i.e. "extension") wallets, including the MetaMask Snaps program.

Examples

Seehttps://chainagnostic.org/CAIPs/caip-25#request

TODO - Verite-specific verbose example with wallet scope object and current WalletConnect syntax

Issuance

The complex part of issuing a verifiable credential is negotiating between issuer and end-user (via their wallet and/or dapp

interfaces) what the contents and preconditions/inputs to a credential are. In the case of Verite, these are relatively simple, and the negotation can be as simple as a wallet downloading the issuer'sCredential Manifest and reducing it down to aCredential Application with any inputs provided. In a Verite use-case, this can be simplified even more-- a wallet can simply hard-code theCredential Application if no inputs are required, and dereference theCredential Manifest from a well-known endpoint to confirm it will still be enough to get the credential it is expecting in return.

The request for a credential, then, actually comes from the wallet to the dapp, accompanied by a Credential Application object and, optionally, an ordered array of signature formats the wallet supports from most to least preferred. This latter array is optional, and is largely supported for interoperability purposes in web2/web3 hybrid applications; most dapps and wallets will default to VC-JWTs for the foreseeable future, with the possible exception of Zero-Knowledge wallets and dapps which might prefer more complex signature formats where mutually supported.

Once the issuer has parsed the application and found it matches their expectations/needs, they simply return a VC as a JSON object.

Examples

Seehttps://chainagnostic.org/CAIPs/caip-169#issue

TODO - Verite-specific verbose example with expanded [Presentation Application][]

Presentation

As above, the complexity in presentations is abstracted by the JSON objects exchanged; in the Verite use-case, dapps can essentially hard-code the [Presentation Definition][] object that they use to request a verite credential, since the schemata are versioned and business logic is simple. The optional additional parameters for "holder-binding" do not apply to address-bound credentials, since Wallet Connect has already authenticated the user's wallet and the credentials are already bound to that wallet; in cases where the credential is issued to a DID or other identifiers, these options may be necessary to test liveness, prove continuity of wallet control, etc. These extensions are beyond the current scope of Verite but may be central to future use-cases.

Examples

Seehttps://chainagnostic.org/CAIPs/caip-169#present

TODO - Verite-specific verbose example with expanded [Presentation Submission][]

Extensibility: Storage

In the authorization example above, you may have noticed the slippage between"wallet has the capability to store a VC" and "wallet authorizes this dapp to issue it VCs it will store". Dapps are encouraged to request authorization progressively after explaining to the end-user the contents and utility of a given VC, after escalating from "connect wallet" to "create account", etc.

The actual storage of a VC is meant to be abstracted away from the dapp issuing or requesting the credential for both privacy and security reasons. The actual storage mechanism used can be encrypted or plaintext, local or remote, centralized or distributed, whole or sharded. Since wallets and architectures vary widely, this very broad and complex topic is entirely out of scope of Verite.

Extensibility: OIDC

Support for OIDC servers, particularly as OIDC evolves beyond its third-party cookies and postMessage dependences, is a complex topic well beyond the current scope of Verite. That said, Verite credentials might have a lot of utility beyond cryptocurrency, such as in Web2 applications, particularly for eCommerce and reputation systems looking for more progressive trust establishment and more secure pseudonymity. As OIDC systems incrementally support the presentation of verifiable credentials, thewallet_creds_metadata method may become a useful mechanism for legacy compatibility, presenting Verite tokens or other verifiable credentials in places where previously you could only share verified claims by "Logging in with Hooli." Updated5 months ago * Table of Contents * * Introduction * * Feature Discovery * * * Examples * * Issuance * * * Examples * * Presentation * * * Examples * * Extensibility: Storage * * Extensibility: OIDC