XY0 2.0 Platform: The Sovereign Internet Platform based on the XYO Protocol

Arie Trouw [*], Joel Carter [†], Matt Jones [‡]

January 2024

Abstract

The XYO 2.0 Platform is a system implementation of the XYO Protocol as defined in the XYO Protocol Whitepaper published in January 2018. It focuses on providing a solution that achieves high performance without sacrificing the sovereignty, provenance, and permanence that is the goal setout by the whitepaper. This XYO 2.0 Platform also expands the usage of the core concepts defined in the White Paper to be useful in a much broader set of use-cases, specifically not limiting its use to location. The implementation set forth in this Yellow Paper adds additional protocol definitions to provide guidelines through which future components and alternative implementations can be created while maintaining the ability for them to work together to form a singular XYO Network.

[*]XYO Network, `arie.trouw@xyo.network`
[†]XYO Network, `joel.carter@xyo.network`
[‡]XYO Network, `matt.jones@xyo.network`

Contents

1 Introduction

During the last decade, Web 3 development has been primarily focused on expanding the use of shared ledgers to create decentralized systems. Even though there have been great strides on this front, the very core of this effort is flawed in two ways.

First, shared ledgers moves the control of a system from being fully centralized (effectively a kingdom model) towards a decentralized solution that is based on majority rule and finality (effectively a democracy or republic). Like with all governance systems, the natural evolution of these systems have pulled back from maximizing decentralization towards more centralized concepts for practical, regulatory, or other, potentially sinister, reasons. Even if this pull-back did not occur, the ceiling of shared ledger decentralization is that of majority rule and not true sovereignty.

Second, the performance of shared ledger technology has been and will always be substantially (orders of magnitude) slower and more costly than their centralized equivalents. By its very definition, a shared ledger must either has massive redundancy of data and validation, or lean on trusted systems to improve performance.

This implementation of the XYO 2.0 Platform combined with the core concepts of the XYO Protocol strives to provide full decentralization with nodes that are 100 percent sovereign while using cryptographic technologies and concepts to deliver a trustless network that has performance at scale comparable or better than the performance of an equivalent Web 2 system and orders of magnitude better than equivalent Web 3 systems. This combination not only delivers on the goals of Web 3 visionaries, but also delivers on the goals set forth by the original founders of the internet. The current Web 2 implementation of the internet is completely devoid of sovereignty, provenance, and permanence and we must reverse that trend by delivering a solution that is the foundation for the Sovereign Internet by combining the core tenants of Web 2 and Web 3 combined with the concepts of the XYO Protocol as set out in the original XYO White Paper.

2 Architecture

In producing the XYO 2.0 Platform, various practical decisions have been made for the architecture of the implementation to facilitate interoperability and reduce ambiguity in the protocol.

2.1 Programming Language

The initial version of the XYO 2.0 Platform has been developed using TypeScript. We chose this due to the expansive tools that are available for developing with TypeScript and the compatibility that Javascript (the output of compiling TypeScript) allows for. As a result, this implementation can be used on browsers and with the NodeJS runtime, both on desktop and on mobile devices. The sacrifice of this decision is that running the technology stack on IoT devices, especially battery powered devices will be negatively impacted. This can be addressed by creating limited native implementations of the Platform for those devices.

WebAssembly is used for various high performance cryptographic algorithms since WebAssembly can be seamlessly interacted with from Javascript. Over time, it is possible that more TypeScript based code could be replaced with WebAssembly, but that will be done with care since there are costs of doing this when it comes to understandability of code and debugging.

2.2 Codeless Development

A primary goal of the XYO 2.0 Platform is to allow users to fully customize nodes through Codeless Development using the module system with manifests, configurations, and parameters. The only reason to create custom modules using TypeScript directly should be for performance reasons, not because what is being created is not possible via the Codeless Development paradigm supported by the platform.

2.3 Security

Security concerns in the XYO Platform follows that of the XYO Protocol exclusively using cryptographic mechanisms to provide security. This includes hashes, addresses, and cryptographic signing of data. As for individual security on specific modules, it is left up to that module to mange access either through lists of allowed/disallowed addresses, crypto-economic incentives, or a combination of the two.

2.4 Privacy

Privacy itself is not addressed in the core XYO Protocol, however, the ability to optionally keep payloads private is accommodated by the protocol. There are two primary paradigms that we utilize in the XYO 2.0 Platform.

2.4.1 Just-In-Time

Just-In-time Privacy (JITP) for XYO is the paradigm where a node shares hashes of payloads without sharing the actual payloads. This allows sovereign Bound Witnesses to be created, establishing provenance and maintaining immutable permanence, while maintaining privacy of the originating payload. JITP can be used to create sovereign games, where multiple

parties can lock in moves without exposing what those moves are, and then only exposing those moves when declaring victory.

2.4.2 Subnet

Subnet Privacy is the ability to run a private XYO Subnet. This is accomplished by running one or more nodes that are networked together, allowing each other to have access that is not available to nodes outside of that network. This is similar to a traditional intranet. The security concerns are also very similar to intranets in that there will be nodes that have access to both the private XYO Subnet and to external networks such as other private subnets or the public XYO Network. Since those nodes are on multiple networks at the same time, it is possible for them to intentionally or accidentally leak private data from the private subnet to the external network, so great care must be taken when allowing nodes to connect to a private subnet.

3 Types

3.1 Hash

All hashes in the XYO Platform are SHA256 hashes of a binary buffer. See the Payload section about details on the rules for creating the source buffer for a payload from which the hash is calculated.

When representing hashes as a string, they are lowercase hex strings that ARE NOT prefixed with a 0x.

3.2 Address

All addresses in the XYO Platform are ECDSA/Secp256k1 addresses that are compatible with Ethereum. This means that the address is the last 20 bytes of the Keccak-256 hash of the public key of the account.

When representing hashes as a string, they are lowercase hex strings that ARE prefixed with a 0x.

3.3 Schema

A schema is a string that defines the mapping of a payload to a TypeScript type or JSON schema definition. Schemas may include any lowercase alpha/numeric character and the period (.) character. This format is similar the reverse domain pattern that is used by many package definitions to produce large to small conical names.

Even though schemas can appear to represent derivation of types and efforts are and should be made to have that be true, there is no way to programmatically enforce this and thus it remains just a convention.

3.4 Payload

3.4.1 Encoding

Payloads are encoded as JSON objects for easy use in TypeScript/Javascript and universal use in other languages. This decision does however required various considerations which are outlined in other sections below. Meta fields in the data object are fields that are not included in the hash. All meta fields have their names prefixed with an underscore character.

3.4.2 Hash

For consistent hashing a specific procedure must be followed when generating the hash.

1. Sort the object by field name. This is a recursive operation, where each sub object also has its fields sorted by name.
2. Remove meta fields by removing all fields that are prefixed with an underscore character.
3. Stringify the JSON object without whitespace using utf-8 encoding.
4. Generate a SHA256 hash on the resulting string.

3.4.3 Schema

The schema field in a payload is used to communicate the structure of that payload. The schema is used to map to Payload type definitions in TypeScript and JSON Schema definitions that can be used to validate the structure of the payloads.

3.4.4 Structure Validation

Validating the structure of a payload is done by validating the schema field for conformity to the Schema type. Once it is determined that the schema field is valid, an JSON Schema can be loaded, if one exists, to fully validate the payload using AJV or a similar library.

In some cases, the JSON Schema to corresponds to a payload schema field are defined in importable packages for ease of use.

The official definition however is retrievable from the xyo-config.json file on a web domain. The registered domain name in reverse is the root of the items that are allowed to be defined by it, specifically hashes for the JSON schemas for Payloads. For example, xyo.network is the authority for all schemas that start with network.xyo.

3.4.5 Hash Validation

Generally the hash of a payload is not passed along with the payload, so no validation is required. The hash can always be recalculated as needed. A specific case when validation is required is when a service returns a payload based on a hash, such as an Archivist Get Query. The caller should always generate the hash of the payload after receiving a payload from a service rather than relying on service to return the correct payload.

This is one of the fundamental aspects of XYO that makes data exchange sovereign and trustless.

3.4.6 Pointer

A Payload Pointer is a payload that statically defines how to find a dynamic payload. A simple example is a pointer that points to the most recent instance of a payload with a specific schema. Substantially more complicated filter criteria can be used to define Payload Pointers.

3.4.7 Coat Check — Rename to Payload Promise?

A Coat Check payload is a specialized Payload Pointer. It is used to facilitate a delayed response, similar to a promise in many languages. The pointer is generated with the knowledge that initially it will not point to a result payload but will point to a result payload once the delayed processing is complete.

3.4.8 Hashed Meta Fields

Any fields whose names are prefixed by a dollar sign ($) are considered meta fields, but are included in the overall hash of the payload. In the case that there are hashed meta fields, an additional field ($hash) is created for the hash of the payload excluding the meta fields. In the cases where there are no hashed meta fields, the ($hash) field is not added since it will be equal to the overall hash by definition.

The purpose of hashed meta fields are to have the ability to add context to a payload where the data of the payload may be different because of context. This allows two witness or diviner payloads to be generated at the same time, but in different contexts, specifically given different working sets, to both be valid despite having different values.

3.4.9 Unhashed Metafields

Any fields whose names are prefixed by an underscore (_) are considered meta fields and are excluded from the overall hash of the payload.

The purpose of unhashed meta fields is to allow for additional data to exist in a data store, usually an archivist. Usually this data is used for optimization purposes, but since there is no signing or hashing of this data, it is considered unreliable and should never be included in payloads when passing them to other modules. Furthermore, returning internal data like this usually is a security problem, so should be avoided.

3.4.10 Best Pracitices

Best practices are not currently enforced by the system, bu maybe enforced in the future. To assure the best ongoing compatibility with the XYO Platform, it is highly recommended that payloads conform to the best practices.

1. Field names should all be lowercase, optionally using underscore or hyphen to split words. Mixed and Camel case should be avoided along with any other symbols.

2. schemas should be chosen in a conical pattern to allow for easy communication of logical hierarchy.

3. Treat conical schema hierarchy as derivation chain, making sure that longer (extended) schemas implement the same fields as their parent (shorter) conical base counterparts.

3.5 Bound Witness

3.5.1 Encoding

A BoundWitness object is a specialized payload that conforms to all the rules of a payload when it comes to encoding and hashing along with these additional requirements.

3.5.2 Addresses

The list of addresses that are partaking in the binding are listed in an array under the field name 'addresses'. These conform to the address encoding rules and have a fixed order that is required for matching to the signatures provided by them.

3.5.3 Signatures

The list of signatures are listed under the field name of 'signatures' prefixed with an underscore character. This causes this field to be considered meta data and is excluded from the hash. The order of the signatures must match the order of the address that are listed in the 'addresses' field and there must be a signature for each address for the object to be considered valid.

3.5.4 Validation

Validating a Bound Witness starts with validating it as a Payload since it is just a specialized Payload. Once that is complete, the Bound Witness specific fields are validated as follows:

1. Length of addresses array and _signatures array must be equal.
2. Each entry in _signature must be a valid signature or the hash of the Bound Witness by the corresponding entry in addresses, using position as the method to match the signature to the address.
3. Validate payloads entries: Validate that each entry is a valid hash and has a corresponding valid entry in schemas.
4. Additional validation can be done if the payloads that are bound are available to the validator. In that case, full Payload validation can be performed on each entry along with validation of the hash and schema for each payload map to the actual payload.

3.5.5 Query Field — Rename to root?

Some Bound Witnesses contain a query field. If this field is present, that Bound Witness is considered to be a Query Bound Witness.

3.6 Account

Account is an object that is defined by a 32 bytes private key and has the ability to generate the public key, and address. Accounts also have the ability to sign data with the private key and validate a signature of data, returning the address of the signer.

3.7 Wallet

Wallet is defined by a mnemonic phrase. A root account for that phrase can be generated, which is the account that is used if a Wallet is passed to something that expects an Account. Additionally, a Wallet can derive additional wallets/accounts by providing a path for derivation.

4 Module System

The XYO Platform is implemented with a module system that defines the initialization, discovery, interaction, security, normalization and bundling of functionality. There are 6 types of modules supported in the XYO Platform, Nodes, Archivists, Diviners, Sentinels, Witnesses, and Bridges, with the option for additional types in the future. Every modules shares core functionality, allowing it to be a module, that allows discovery and communication between modules.

4.1 Manifests

Manifests are JSON configurations of a Node or a dApp. XYO Platform initializes by loading a manifest or set of manifests. If a manifest requires external module implementations, a Module Locator is also required.

4.2 Locators

Locators have the ability to 'locate' a module implementation that can satisfy the requirements defined in a manifest file. A Locator also functions as the point where private or platform specific configuration can be provided for a module implementation.

4.2.1 Labels

Labels are used to identify differently parameterized versions of the same implementation of a module. Usually labels clearly communicate the variance that the parameters provides to allow mapping in the manifest to the correctly parameterized module implementation.

4.3 Addressing

Every module has an address associated with it. The address is used to uniquely identify and communicate with the module. Each module must have the private key that generated its address so that it can also sign Bound Witnesses when needed.

4.4 Discovery

Modules are loaded as a tree with every module, except Node and Bridge Modules, restricted to being leaves. Only Node Modules may be the root of the tree.

4.5 Resolvers

Resolvers provide the mapping between module addresses/names and the actual instances of those modules. When resolving a module, a ModuleInstance is always returned, but it is possible that the object is a Proxy or a Wrapper that facilitates calling the module in a way that is similar to calling it directly.

4.6 Interfaces

Modules have interfaces that allow for direct calls to their functionality. These entry points are only available to be called from another module that is running in the same memory

space. This is only provided for performance purposes and allows the module to be used as a trusted source. Additionally, when directly calling a module, no Bound Witness is produced. Interfaces do not support payment rails.

4.7 Queries

Modules are loaded as a tree with every module, except Node and Bridge Modules, restricted to being leaves. Only Node Modules may be the root of the tree. When calling a module through the query entry point, a Bound Witness is produced and authentication may be performed if required. Payment rails are available only when calling a module via queries.

4.8 Eventing

Events are generated by modules when notification is required. Events are currently only available through the direct interfaces of modules, which means that the listener of the events must be in the same memory space as the module. This also means that the events are 'trusted' and thus should be validated and verified if full trust of the emitting module can not be established.

5 Module

XYO Platform provides an abstract implementation of Module that implements the shared module interfaces, queries and events.

5.1 Account

A unique account for the module which produces the address that is used to communicate with the module and also to validate signatures produced by the module.

5.2 Name

Optional alias for the address of a module. The XYO Platform enforces uniqueness of this name only is a single Node scope. In the case that there are multiple modules with the same name that are discoverable, the modules that is logically closest to the discovering module will be found.

5.3 Config

At the point of creation, every module supports taking a config payload. The contents of a config payload is usually specified in the manifest that is being loaded. Since the Config is a Payload, only JSON serializable fields are allowed.

5.3.1 Permissions

TODO

5.4 Params

Params are similar to Configs in that they are used at the point of creation of an instance of a module. Params are not Payloads and may convey objects that are not JSON serializable. Secure data also is conveyed in Params, such as API keys. Params are generally provided when a module is registered with a Locator resulting in every instance having the same Params and differing Config.

5.5 Lifecycle

Modules have three distinct states, creation, registration and attachment.

5.5.1 Creation

A created module exists in a vacuum and has limited use. Creating a module processes the Params and Config objects to define the module instance. Before a modules can be registered, it must be created.

5.5.2 Registration

Registration makes a module available to a containing Node Module. A module should generally just be registered with one Node. Registration can only be done from the same memory space as the Node module receiving the registration. The module being registered also must be in the same memory space. Registration can not be done remotely.

Before a module can be attached, it must be registered.

5.5.3 Attachment

Attachment makes a module addressable, allowing both the module to communicate with other modules and other modules to communicate with it.

5.6 Queries

5.6.1 Address

Retrieve information about the module's account including the current previous hash of the account.

5.6.2 Manifest

Generate a manifest that reflects the structure of the module. This may be different than the actual manifest that was used to create it, since it is generated and also will exclude private modules that may have been specified in the creating manifest.

5.6.3 Subscribe

Subscribe to an Event. Not yet fully functional.

5.7 Events

5.7.1 Busy

Emitted when the busy state of the module changes.

5.7.2 Error

Emitted when an error occurs.

5.7.3 Queried

Emitted when the module is invoked via the query interface, providing both the input and the output.

6 Node

Module that contains other modules and facilitates discovery, resolution and communication between them.

6.1 Interface

6.1.1 Register

Register a module

6.1.2 Unregister

Unregister a module

6.2 Queries

6.2.1 Registered

Retrieve list of all registered modules

6.2.2 Attach

Attach a registered module

6.2.3 Detach

Detach a registered module

6.3 Events

6.3.1 Registered

A module was registered

6.3.2 Unregistered

A module was unregistered

6.3.3 Attached

A module was attached

6.3.4 Detached

A module was detached

7 Archivist

Provides storage of payloads.

7.1 Queries

7.1.1 Get

Get payloads by hash

7.1.2 Insert

Insert payloads

7.1.3 Commit

Insert all the payloads that are in the archivist into the archivist's parent(s)

8 Bridge

A Bridge model connects to a remote module via a predefined transport such as HTTP or Bluetooth.

8.1 Proxy Module

Proxy Modules represent remote modules that are reachable via a Bridge. Similar to a wrapper, a Proxy Module presents the remote module as if it exists in the same memory space as the calling module.

9 Diviner

Transforms one or more input payloads into zero or more output payloads. Diviners should be pure, not calling other modules behind the scenes.

9.1 Queries

9.1.1 Divine

Perform the divination as prescribed by the module, taking n inputs and returning m outputs.

10 Sentinel

Orchestrate a series of modules, usually witnesses, diviners, and sentinels to perform a more complicated set of logic. Supports various triggering mechanisms, including direct invocation, interval and event.

10.1 Queries

10.1.1 Report

Initiate the prescribed series as defined in the sentinel's configuration, taking an input payload working set and producing an output payload working set.

11 Witness

Onboard data from an external source including but not limited to API calls, sensor readings and system reflection.

11.1 Queries

11.1.1 Observe

Initiate an observation using the implementation of the witness.

12 Resolver

Resolves an address or name of a module by locating it and returning a ModuleInstance that can be used to access the module.

12.1 Local

Resolves an address or name by finding and returning the actual instance of the module and returning a reference to it.

12.2 Remote

Uses a Bridge to discover remote modules and then provides a reference to a Proxy Module that exposes the requested module indirectly.

12.3 Composite

Resolver that in a local resolver but also can be configured to have child resolvers, local or remote, that also will be inspected to locate the requested module.

13 Wrappers

Object that wraps a module instance, removing direct access to the wrapped instance's interface and only allowing access to it through query calls. The wrapper provides a replacement interface that mimics the wrapped module instance interface, converting the calls into query calls.

14 Huri

A Huri is a form of URI that references a payload via its hash.

14.1 Protocol

14.2 Hash

14.3 Hints

15 Component System

We chose React as the framework to create user-interface components for the XYO 2.0 Platform. This does not preclude using other frameworks and in the cases where native user-interfaces are required, alternative frameworks will be required. In all these cases, the paradigm which we embrace in our React implementation should be followed.

15.1 Renderers

A Renderer is a component that takes a working set of payloads and renders them. In many cases, the renderer is so simple that it only has the ability to render a single payload of a specific type. The most basic of renderers simply renders a payload's raw data, which usually is a JSON object. In most cases, a dApp strives to have much friendlier renderers than the basic renderer. Currently, the only way to make custom renderers is to create a new React component usually based on an existing renderer such as the basic raw renderer. We have a future goal to produce a codeless system to producing renderers.

15.2 Hooks

Hooks are a specific system used in React, but as a paradigm, exists in most user-interface frameworks. The purpose of hooks to to separate the acquisition/manipulation of data from the rendering of that data. In most cases, the provided generic hooks that provide access to XYO modules are sufficient to gather the data in a user interface for a dApp.

16 Acknowledgements

This white paper is the product of an inspiring team effort that was made possible through the belief in our vision from the following individuals:

References

[1] Trouw, Arie; Levin, Markus; Sheper, Scott *XYO Protocol White Paper*. XYO Website. January 2018