Abstract

"egyn" is a global type registry that lives on-chain. Its main goals is to curate and manage primitives in order to secure the software supply-chain. Each entry is a mapping between Content URI's that resolve to package metadata and allow users to audit sources and use cannonical implementations of code that are reused often in the ecosystem. To acheive this, a smart contract is used as a one-of-many source of truth, in addition to a publishing step that leverages semantic web constructions in order to increase the legibility of ABI interfaces. This new standard leans on pre-existing ERC-1820[1] to retrieve functionality from deployed contracts.

Introduction

As it stands, there are several problems that exist in the blockchain space as it pertains to reusing code:

- **Lack of deep introspection**. Currently there is a reliance on block explorers to "verify" contracts by checking bytecodes against sources provided by the author. While this provides a way to quickly check the set of interfaces a contract supports, it lacks the ability to track dependencies. Furthermore, methods can be obfuscated using similar function name signatures. There are also centralization concerns as the data is cumbersome to consume on-chain since it is behind an API provided by these services.
- **Discoverability of primitves, libraries, etc**. There are no places to find pre-deployed libraries of code on-chain causing many projects to redeploy the same implementations such as precompiles, ERC contracts, applications that have broad use in the ecosystem, etc.
- **Readability for auditors**. Often when conducting an audit, package repos are supplied in addition to the code itself. Hardening existing contracts with metadata that supplies prior audit reports will speed up the ability to develop more secure code.

These considerations are the motivation to create a standard that projects can coalesce around to create new protocols and modularize their platforms more efficiently reducing the amount of repeat contracts that get deployed on-chain.

Background

In software ecosystems outside of blockchains there has been a notion of package managers such as `npm` to help developers quickly iterate on original ideas. To enable code reuse, security is taken head on by leveraging public-key infrastructure to sign package manifests and then host them on a curated website. Here, one can see the widespread use of a particular primitive and make the decision to contribute or fork a new version with additional features or make different design decisions. Broadly speaking, the adoption of this technology has not made its way into the space for a variety of reasons, see EthPM[2]. While similar in spirit, Egyn attempts to bridge the gap by giving more tools for curation, in addition to providing a succiinct metadata standard that is composable. The scope of previous projects have concerned themselves solely with Solidity code, where as Egyn will attempt to encourage other ecosystems to get involved by allowing them to publish a set of keys that can be used to verify package signatures of all types. Primitives are then shared and discoverable given a set of semantics.

For example, the function `mapReduce` exists as a construction in many languages that can be curated on-chain. Developers and auditors can then check new implementations against previous ones that may have already gone under the same scrutiny. Furthermore, as projects begin to make their protocols more modular, this can be used to check that interoperability is preserved while contributors/maintainers make changes to the underlying protocol.

Solution

Egyn sets out to solve these problems by introducing several components including a smart contract registry, IPLD-based metadata system, and various curation mechanisms that allow both the project itself, as well as participants, to define their own rules for publishing updates and propagating changes with proxy contracts. In the following sections we will formally define the protocol, going in depth for each of the constructions. The next section will give a high level overview of these components. It will explicate the path for onboarding existing packages using the pre-established public-key infrastructure.

The following section 2.4 will outline the registry contract responsible for publishing. Then we will describe the metadata standard in Section 3 that will ultimately serve as the package manifest and how it will leverage DAG-JSON. Here, the mininal requirements are defined for composability and required fields.

Protocol Overview / Definition (Section 2)

There are several actors in the protocol, namely package owners, registry operators, and data relayers. First we will look at the responsibility package owners have including their roles in the context of the registry.

2.1 Package maintainers

The main role of package maintainers is to keep manifests up-to-date with the latest version of their code and provide signatures for others to verify the authenticity of manifests published to the registry. It is assumed that they have an Ethereum public/private keypair. Formally, every maintainer will have to submit the following:

```
{
    "ttl": 1698233087,
    "signature":
"iQEzBAABCgAdFiEEGInq7S250/ff+zymwaG5EGnEr1kFAmLWpkEACgkQwaG5EGnE...",
    "algo": "pgp",
    "publicKey":
"xsBNBFmUaEEBCACzXTDt6ZnyaVtueZASBzgnAmK13q9Urgch+sKYeIhdymjuMQta..."
}
```

- ttl: time to live for the signatures before requiring reapproval.
- signature: string of the signed CBOR message.
- algo: algorithm used to sign the message.
- publicKey: the public-key of the signer.

In the beginning, Archetype will operate a service that verifies signatures to surface a curation mechanism for maintainers. It will later receive commitment hashes of verified signatures from other operators that will be stored in the registry. Note: while this is a novel scheme for handling basic types of signatures, more advanced schemes will be supported as the scope of the service increases. This implies that this section is subject to change or have additional fields when using COSE envelopes [3] for example.

2.2 Registry Operators

While this protocol can provide a single instance of a registry that can be used canoncially across many different applications, the goal is to integrate the registry spec into an a new ERC standard that would allow other registries, which have the same interface, to share the same data repositories. Names of these registries can be resolved either through ENS, or contract addresses provided by a package signer. This is a desired consideration because it facilitates package maintainers to host their own sets of repositories that may only want to surface frozen versions of distributions. Each operator has the responsibility of sharing their signing keys and can choose to link their instance to other registries in the ecosystem. The motivation for this would be to have deeper links in the semantic graph that will be elaborated in the next section.

2.3 Data Relaying

The main gas costs will fall onto package maintainers to submit their signatures, as well as any additional updates to a content URI relevant to their releases. However, the goal will be to have a set of relayers available to handle transaction load.

2.4 Contract Registry

At a high-level, there are three main components to the registry: signature verification, content URI resolution, manage registry mappings for deployments. Below the interface will be outlined in Solidity code:

```
pragma solidity ^0.8.20;

interface ITypeRegistry {
    function submitSignature(bytes, uint256) external;
    function updateManifest(bytes, bytes, uint) external;
    function retrieveImplements(bytes) public view returns (memory address[]);
    function approveMaintainers(bytes, address) returns (bool);
    function commitments(bytes,bytes,bytes) external;
    function verifyDeployment(address) external verifierRole;
    function supportsStrict(address,bytes4[]) public view;
}

contract GlobalTypeRegistry is ITypeRegistry {
    // WIP
}
```

Metadata Specification (Section 3)

For each of the packages in the registry, there is an associated manifest URI that describes both the primitives, as well as the basic metadata surrounding publishing. Leveraging DAG-JSON, the manifest can be divided into subcomponents using IPLD to create a rich data structure.

3.1 Package Metadata

For each of the package, there must be at least a manifest that contains the code repository. From there, additional fields can be added that do not conform to the strict specification outlined below. For example, maintainers can attach an audit field to the document and link out to reports generated by audit teams.

```
{
    "name": "example-package",
    "version": "0.0.1",
    "manifestURI": "QmXg9Pp2ytZ14xgmQjYEiHjVjMFXzCVVEcRTWJBmLgR39U",
    "repository": "https://github.com/example-org/example-package.git"
}
```

See below for the basic outline of how the `manifestURI` resolves into a DAG structure that can contain many different links to other linked data formats.

```
{
    "Data":{"/":{"bytes":"c29tZSBkYXRh"}},
    "Links":[{"Hash":
{"/":"QmXg9Pp2ytZ14xgmQjYEiHjVjMFXzCVVEcRTWJBmLgR39U"},"Audits":"reports","Tsize":8},
{"Hash":{"/":"QmXg9Pp2ytZ14xgmQjYEiHjVjMFXzCVVEcRTWJBmLgR39V"},"Primitives":"some
other link","Tsize":8}]
}
```

One of the links in this DAG structure will contain the manifest outlined in the previous section with a COSE signature[4]. In the future, statements can be made using these envelopes and will be stored in the commitments mapping inside the registry contract. It will be the basis of the sparse merkle tree used to generate proofs about statements surrounding given packages.

```
  {
     "id": "did:example:123",
     "verificationMethod": [{
       "id": "#key-42",
       "type": "JsonWebkey",
       "controller": "did:example:123",
       "publicKeyJwk": {
         "kty": "EC",
         "crv": "P-384",
         "alg": "ES384",
         "x": "LCeAt2sW36j94wuFP0gN...Ler3cKFBCaAHY1svmbPV69bP3RH",
         "y": "zz2SkcOGYM6PbYlw19tc...rd8QWykAprstPdxx4U0uScvDcYd"
       }
     }]
   }
```

This will serve as the identity of a given signer for a code repository. When a statement is made, this will need to be validated in addition to the receipt of the message. Archetype is committed to leveraging open standards to achieve this functionality. For further reading about the topic please see the IETF standard for COSE message signing for supply chains[5].

In addition to commitments made about the codebase, developers can choose to implement their own linked data structure to encapsulate their own logic. An example of how documentation can be handled using the above specification would be to also share primitive metadata:

```
{
  "primitives": [
    {
      "id": "uint256",
      "description": "Unsigned 256-bit integer",
      "type": "integer",
      "constraints": "0 <= x < 2^256"
    },
    {
      "id": "address",
      "description": "Ethereum address",
      "type": "string",
      "format": "hex",
      "length": 42
    }
  ]
}
```

While there are no strict enforcements for particular structures outside of signature verification, we implore maintainers to use some of these templates to better document their codebase.

3.2 Method Byte Identifiers

A desired notion of deterministic builds is something we hope to make possible with this metadata registry. The way this would work in practice would be to publish the solidity versioning, as well as the bytecode / ABI in the metadata DAG inside the package manifest. Here, our curation layer would have the responsiblity of verifying builds and adding methods to the byte code mapping inside of the contract. These signatures will lean on the existing ERC-1820 implementation to perform a check if a contract conforms to an interface. However, in addition to this check it will register the contract to a mapping of contracts that all have the same byte code. This will then expose a method that has a more strict check that will ensure that a given contract both supports a given interface but also strictly conforms to a set of methods. In other words, when an address is linked to a code repository with a deterministic ABI, that contract may only have the methods specified to that implementation. This verified status will be displayed on the curation layer.

3.3 Identity System and Additional Commitments

Implicitly, the current public-key infrastructure is being leveraged in lieu of an identity system. This will allow the registry the capability of priviliging package maintainers with the ability to update their own manifest with the same keys they've used to publish a package on other services, i.e. git release, npm, etc. A future line of work would be to add an identity layer for arbitrary users to make commitments about published code. Curation could then be leveraged to surface relevant metadata surrounding contracts that a maintainer may not want to support in the immediate manifest but can be accommodated for at a later date.

Scope of Interest

Publisher trust, reputation, and curation in modular development paradigms

Prior to v4, users, LPs, and developers only had to trust Uniswap Labs and the vertically integrated v2 and v3 contracts. [6]

Given the frequency of hacks in DeFi, one cannot rely solely on the presence of an audit. A common consideration is the reputation of the entity that is publishing the code. Uniswap Labs, as a leader in the space spanning several years, has developed a strong reputation for shipping secure code. This strong reputation, in combination with industry leading development practices and extensive audits from highly reputable firms, has historically provided enough trust for users to confidently interact with the markets on Uniswap. They simply needed to trust one entity: Uniswap Labs. Now, with the introduction of hooks, users, LPs, and developers have to trust the Uniswap v4 singleton, the Uniswap v4 smart contracts, and the hooks that are used by the pools. This is a significant increase in the number of entities that users, LPs, and developers have to trust.

Front-end security

On the front-end, the simplicity of previous iterations of Uniswap allowed for a simple user experience. In a vertically integrated system, the surface area for third party integrations is small and simple. This led to simple integrations at the edge, simple SDKs, and simple open-source front-ends. These third parties were therefore inclined to use those existing SDKs, and many opted to forgo their own front-end entirely and instead use the open-source front-end. This led to a simple user experience, and a simple security model. With v4, the front-end has to be able to handle the complexity of the different hooks that are used by the pools. This introduces new security concerns.

Intents

In future work, a contract regime can be developed for the GTR system to provide stronger guarantees and a higher degree of trust and composability for the emerging Intents architecture used by protocols like Cowswap, UniswapX, Anoma, and others.

Relevant standards and documentation:

- https://eips.ethereum.org/EIPS/eip-7521
- https://ethresear.ch/t/a-decentralised-solver-architecture-for-executing-intents-on-evm-blockchain/16608
- https://medium.com/@okcontract/introducing-low-level-intents-a-summary-of-our-talk-at-ethcc-6-5f8246c80639
- https://blog.essential.builders/introducing-erc-7521-generalized-intents/

Conclusion

Overall, with these sets of metadata standards and contracts, Archetype sets out to harden the software supply chain. It achieves this by giving developers the tools to specify metadata about packages they release, as well as surface contracts that may exist on-chain that implement a specific set of interfaces. This can then be used as a preliminary check in contracts that wish to references addresses on-chain to ensure they have a correct implementation. As the registry matures, statements can be made about the veracity of a claim dealing with functionality. This could lead to sophisticated commitment schemes across projects that ensure that when new versions are released, shared test suites still pass. There is now a trail of artifacts, for auditors, that can quickly be surfaced for many parts of the dependency chain. The previous section discussing the scope of interest elaborates on real world use cases where the registry minimizes trust between actors, while still letting developers create novel protocols.

The ultimate goal is to forward the registry through the same scrutiny of the EIP process while other teams leverage the contracts to create a more secure ecosystem. Semantic schemas and commitment regimes are to come with additional examples. By allowing flexibility on this front we hope to see a wide ranging set of tagging and semantic constructions. This initial version has a particular focus on message signing to onboard existing public-keys.

References

[1] https://eips.ethereum.org/EIPS/eip-1820 [2] https://docs.ethpm.com [3] https://cose-wg.github.io/cose-spec/ [4] https://datatracker.ietf.org/group/scitt/about/ [5] https://datatracker.ietf.org/doc/draft-ietf-scitt-

architecture [6] https://github.com/Uniswap/v4-core/blob/main/whitepaper-v4-draft.pdf