Tokenized Assets

Technical Implementation

v. 1.1

April, 2019

Abstract: In this short paper we introduce the tokenization structure, through which 'eTokens' are minted, upgraded and monitored. In this work, we focus on the technical aspects of the eToken platform implementation. We describe the design tradeoffs related to supporting features such as upgradability and permission management. At the same time, we prioritize contract security through a defensively implemented layered design which emphasizes a strong separation of concerns.

This work is the first in a series of papers addressing all matters related to the tokenization of assets.

General Disclaimer

This paper and any other documents published in connection with the same relate to the tokenization structure, through which ‘eTokens’ are minted, upgraded and monitored.

Risk Warning: the purchase of ‘eTokens’ carries with it significant financial risks. Prior to purchasing ‘eTokens’, you should carefully consider and understand the risks associated with these. For more detailed information on the associated risks please see the risk disclaimer that can be found on https://www.etoro-digital-assets.com

Advice: this paper does not constitute financial, legal, tax or any other type of advice in connection with the purchase of ‘eTokens’. You should not rely upon this document when making a decision relating to the purchase of ‘eTokens’.

Professional Advice: we always recommend that you consult an appropriate professional adviser (including but not limited to a lawyer, accountant or tax advisor) prior to purchasing any ‘eTokens’ in order that you are able to assess and determine whether this is the right product for you.

Regulation: this paper does not advise on the regulation of ‘eTokens’ in any jurisdiction. ‘eTokens’ do not have legal tender status, and rather represent a value connected to a selected fiat currency and/or commodity and/or any other financial instrument. There is no central bank that can take corrective measures to protect the value of ‘eTokens’. Moreover, there is always a risk that changes in the applicable legislative or regulatory regime may adversely affect the use, transfer, exchange, and value of these. ‘eTokens’ markets and exchanges are not currently regulated with the same control, and customers are not entitled to the same protections, available in relation to other financial instruments.

Changes/Updates: this paper currently sets outs our vision in relation to ‘eTokens’ as at the date on which it was published and with all the information that was made available at the time. ‘eTokens’ are continuously under development and, accordingly, may be updated and changed from the specifications below once fully completed. There are no representations or warranties given in relation to the completion or reasonableness of the vision contained in this paper and nothing should be relied upon as a commitment or representation.

Views/Opinions: all views and opinions expressed in this paper are those of eToro Digital Assets. Information contained in this paper is based on sources considered reliable by eToro Digital Assets and there is no assurance as to the accuracy or completeness of such views or opinions.

Language: this paper is produced and issued in English only. Any other translation of this paper is not certified by eToroX Digital Assets and therefore no assurance can be made as to its accuracy or completeness.

Third Parties: any references in this paper to third parties are for illustrative purposes only. Other than in connection with the companies in the eToro Group and our official advisers, the use of our name or ‘eTokens’ does not imply any affiliation with, or endorsement by, any of those parties.

1 INTRODUCTION

Tokenization is defined as the process of representing any financial asset class through an on-chain abstraction. Unit representation methodologies range over a broad array of practices with the shared aim of representing economic exposure to, or ownership and transfer rights of, a digital asset which can be sold or redeemed for the value of the underlying through trade on any publicly available exchange on which the asset is listed. For the purpose of clarity, we refer to tokenized assets as digital assets, while blockchain only assets which have no referable real-world asset are referred to as virtual assets. Tokenized assets are typically categorized by the mechanism by which they are created:

a) Algorithmic digital assets are stabilized through a sophisticated set of fiscal mechanisms through which investors are incentivized to seek arbitrage opportunities when the digital asset exhibits volatile price movements dissimilar from those of the pegged or underlying asset.

b) Asset-backed digital assets function as either: i) as an IOU through which the issuer or a guarantor is held liable for the value of the digital asset through legal agreements which entitle tokenholders to rights to or interests in the underlying real-world assets; or ii) in the case of the eTokens as an arrangement under which the value of the digital assets is stabilized through the buying and selling activities of a market making entity whose own exposures in stabilising the assets are backed by the market make, noting that tokenholders have no rights to the underlying and no guarantees in relation to the activities of the market maker.

While the majority of digital assets on the market today track the US dollar, nearly all issuers have announced plans for diversification into representations of a variety of asset-classes. While most remain within the domain of capital markets, a blue ocean mindset dominates the landscape, with some vendors approaching early stage efforts in tokenization of atypical, illiquid assets and novel instruments (Figure 1.1).

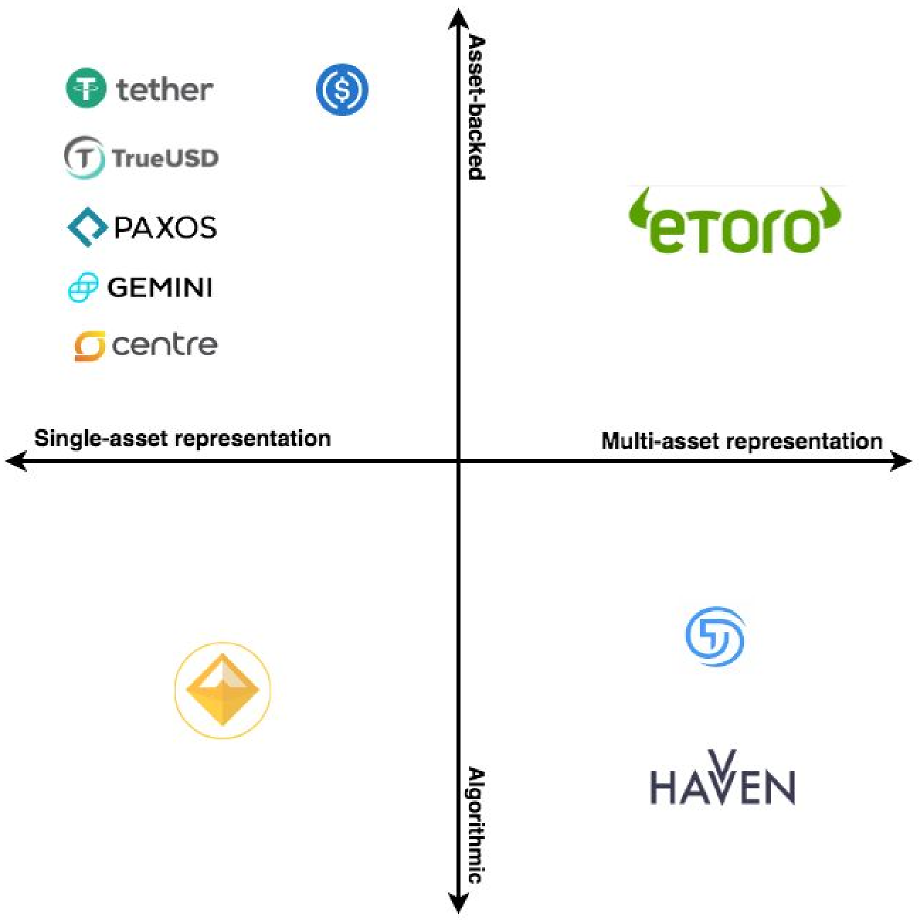

Figure 1.1 | Leading vendors currently offering tokenized asset-classes.

2 IMPLEMENTATION

Our solution is currently implemented using the Ethereum smart contract platform as a base. Ethereum provides a suitable foundation for supporting our needs due to its relative maturity as a smart contract platform and its broadly applicable and general feature set. Furthermore, the Ethereum Improvement Proposal, EIP, process facilitates the establishment of ecosystem-wide standards which enables seamless interoperability. An example of such a standard is ERC20, which defines a set of operations that must be supported by a conforming fungible token. At its core, eToken is an ERC20 token with additional supporting infrastructure relating to token management, permission management, and token upgradability. This section gives a high-level overview of the design and implementation of the token itself and its supporting infrastructure. As with other computer programs, using established libraries as a foundation for smart contracts makes the resulting code comparatively more robust with less required development effort. A notable standard library for supporting the development of Ethereum smart-contracts is developed by OpenZeppelin and provides a large body of reusable functionality. For our case, however, we found that the OpenZeppelin libraries were not sufficiently flexible to support our needs. Thus, our current implementation consists mostly of custom code and it only relies on standard library code to a limited extent.

2.1 Upgradability design tradeoffs

Immutability is an important functional characteristic of smart contracts as it provides a guarantee that a contract, once deployed, will behave exactly as prescribed by its code. However, the immutability also creates a number of challenges which revolves around how to update and evolve the functionality which is provided by the tokens. Various strategies exist for providing upgradability on the Ethereum platform, however, at their core, all of them implements post-deployment reconfigurable code paths which allows calls to be proxied to a specified location. A key aspect which must be considered in the design of an upgradable token is how to deal with data migration. In Ethereum smart contracts, data (e.g. account balances) are intimately tied with the functionality of the contract. Concomitant with the high cost of data store operations on Ethereum, migrating large amounts of data between contracts become infeasibly expensive. Consequently, a basic requirement of upgradable smart-contract systems is that their data storage must be separated from functionality. Implementations of upgradability embodying the considerations described antecedently exists in the implementations of all leading tokenized asset representations. For example TrueUSD and Tether USD.

- Token upgrades are transparent to the user and will not interrupt the operation of the previously deployed tokens.
- By separating token balances from token functionality, upgrades are cheap, since no data needs to be moved.

In the end, the issue relating to how and if upgradability should be handled relates to how centralized or decentralized a token should be. The current method for upgradability supported by the token does imply that the power to change functionality is entirely in the hands of the token owner

2.1.1 External ERC20 storage

As previously mentioned, a prerequisite for enabling upgradability in smart contracts is that functionality and storage must be placed in separate entities. In practice, this separation is implemented by creating a separate and extremely simple contract which purely supports data load and store operations. In order to prevent multiple tokens from, accidentally or intentionally, using the same storage, the storage contract will only accept requests from a single contract at a time. This contract is known as the implementor. In addition to load and store operations, the only additional operation provided supports changing the current implementor. When a token is upgraded, the implementor of the attached storage is transferred as part of the same transaction. This prevents the token from being left in an inconsistent state.

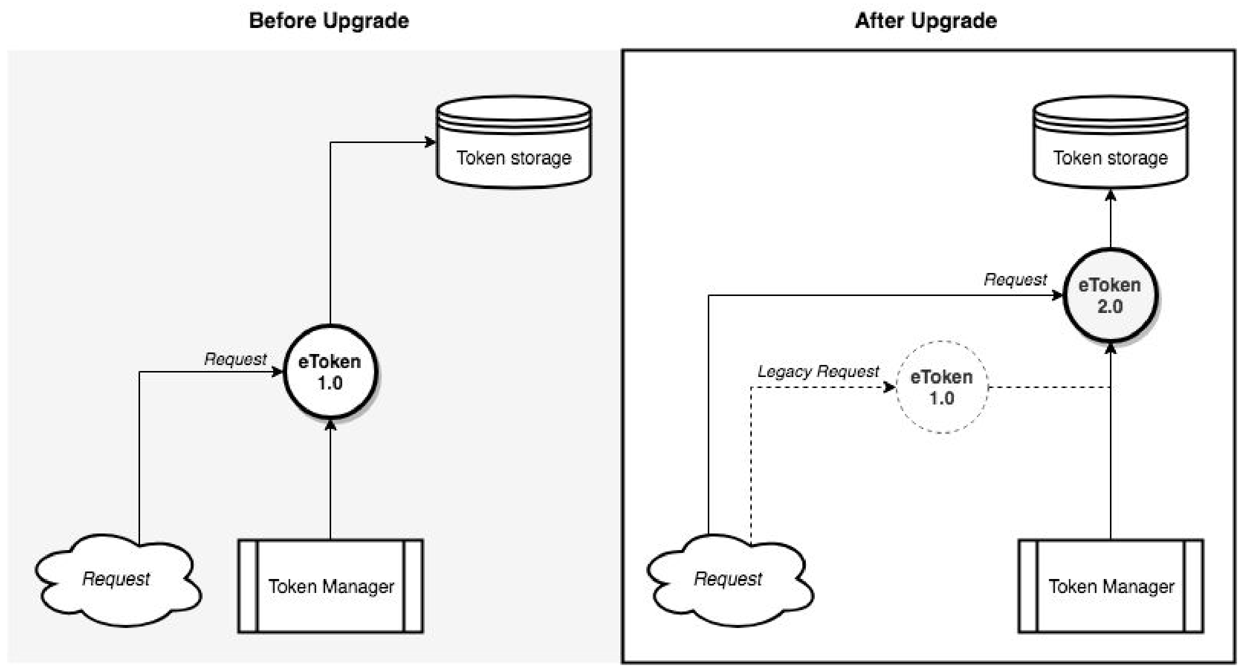

Figure 2.1 | Rendering of the token upgrade process. On the left side, we see the contract communications before the network upgrade. All clients and the token manager targets the token contract. The token contract uses a separate contract for storing balances. On the right side, the contract organization after an upgrade is shown. The token manager is updated to point to the new, upgraded, contract. The old contract is marked as deprecated which causes it to proxy all incoming calls to the new contract. The balance storage contract now provides storage for the new token contract such that balances are transferred seamlessly.

2.1.2 Enforcing permissions in upgradeable tokens

In Ethereum smart contracts permission enforcement based on the address of the account making the initial request. In the Solidity language, this address is readable through a special property called msg.sender. To make the token upgradable, all calls initially target a proxy contract which inherits from the contract implementing the token functionality. When a non-upgraded token is called directly, the execution flow continues internally along an inherited (subclass) function. When an upgraded contract is called, it makes an external call to the corresponding function in the new contract. The challenge now is, that when the new contract retrieves the call, msg.sender points to the upgraded contract rather than the original sender. In order to get around this, the new token is required to provide a set of functions, accepting an explicit sender parameter, which can be used as the entry point for the proxy calls. In order to prevent abuse of these functions, they can only be called by the upgraded (old) contract. The security of this system relies on the fundamental assumption that the old contract cannot be compromised in a way which allows the sender addresses it sends to the new contract to be compromised. The upgrade process is depicted in Figure 2.1.

2.2 Permission management

The owner of the contracts has complete control and can perform all operations. Being all-powerful, it is essential that the owner account gets as little exposure as possible. Therefore, we separate day-to-day management tasks into fine-grained roles which accounts can be assigned to as needed. Note that these permissions are assigned on a token-by-token basis. We define the following roles:

- MinterRole can mint new tokens
- BurnerRole can burn tokens from their own account
- PauserRole can temporarily pause a token, preventing new transactions
- WhitelistAdminRole can add and remove accounts to and from the whitelist contained in the AccessList.

- BlacklistAdminRole can add and remove accounts from the blacklist part of the AccessList

The implementations of these roles are mostly identical to the corresponding roles provided in the OpenZeppelin library. However, one important aspect of our implementation diverges from the OZ-provided implementations. In the OZ implementation of the role contracts permissions may only be granted to an account by another account holding the same permission (e.g., only a minter can add other minters), and accounts can only be removed by permission holders renouncing their own privilege. This was not acceptable for us, as we need a hierarchical permission structure with more centralized control. Therefore, our role implementations follow the following principles:

- All role contracts were made Ownable
- Functions enabling the owner to revoke permissions from an account were added
- Functions for granting permissions to an account were made owner-only

Furthermore, since the roles are tightly integrated into the OpenZeppelin system, modifying otherwise unrelated parts of OZ to use our roles, such as lifecycle/Pausable, was unfortunately unavoidable.

2.2.1 AccessList (KYC whitelist and blacklist)

Access to token transfer functions is guarded by the AccessList. Accounts can be given access (whitelisted) when they have passed a KYC check and abusive accounts can be blacklisted temporarily or permanently. The whitelist functionality may optionally be enabled on a per-token basis, while the blacklist is always enabled. Enabling or disabling the whitelist must be done at token deployment time and can therefore not be changed during the lifetime of a token contract. AccessList is implemented separately from the previously described roles for the following reasons:

1. It is always shared between all the tokens. That is, once a customer is KYC’d, we allow them to trade all of our tokens. The aforementioned roles, on the other hand, may be limited to a specific token. For example, different accounts may be responsible for minting USD and gold tokens.
2. The AccessList contains a large number of addresses and it is infeasible to migrate this data if we need to upgrade the token contracts.

2.2.2 Restricted Minting

In order to limit the damage which can be caused by a compromised minter account, we have devised and implemented the concept of a restricted minting. In restricted minting, a minter is only allowed to mint to a predetermined account which is specified by the owner. This prevents an attacker from minting new supply directly to their account since new supply can only be minted to the dedicated minting account and is subsequently transferred to its final destination as an independent action.

2.3 Practical implementation considerations

In this section, we elaborate on the internal design of the eToken implementation which was devised to underpin the security requirements, upgradability features, and permission enforcement. The eToken design features a well-defined separation of concerns and a streamlined control flow. This was achieved by dividing the design into four layers, each with a clearly defined responsibility. The layers are depicted in Figure 2.3.4 and described after this paragraph. An additional goal of this design was to avoid overriding several public functions with functions of the same name. With this, we aim to minimize the risk that an internal function is unintentionally exposed. A description of the layers follows:

Interface layer	The Interface layer implements the public interface of an EToken as defined by IEToken. If an upgraded token is called, the call is forwarded to the proxy interface of the token next in the chain of upgrades. If the token is not upgraded, the call is forwarded internally to the access-control wrappers in the ETokenGuarded contract. At this point, the initial msg.sender property is captured and passed as an explicit sender parameter to subsequent function calls.
Proxy layer	The Proxy layer implements the chaining of calls through multiple generations of upgraded tokens. All functions implemented here are defined by the IEtokenProxy interface. All upgrade targets must implement this interface. When the chain of calls eventually reaches the currently active token, the call is forwarded to the appropriate function in the ETokenGuarded contract. The functions exposed by this layer are public, but they are only callable by their immediate parent.
Access control	The Access control layer implements wrappers around internal functions which enforce access control using the access control and KYC (whitelist/blacklist) contracts. All access control checks which are performed on the request sender use the explicit sender parameter passed down from EToken. The wrappers are implemented in the ETokenGuarded contract.
ERC20 functionality layer	In the ERC20 functionality layer, the functionality comprising an ERC20 token is implemented. This functionality is now only exposed through internal functions which are wrapped by the outer layers.

As a rule of thumb, the public functions exposed from the token are defined in the IEToken interface and implemented by the EToken contract. However, exceptions were made for administrative functions which are not threaded through the upgrade machinery. Examples of such are functions for managing permissions and roles.

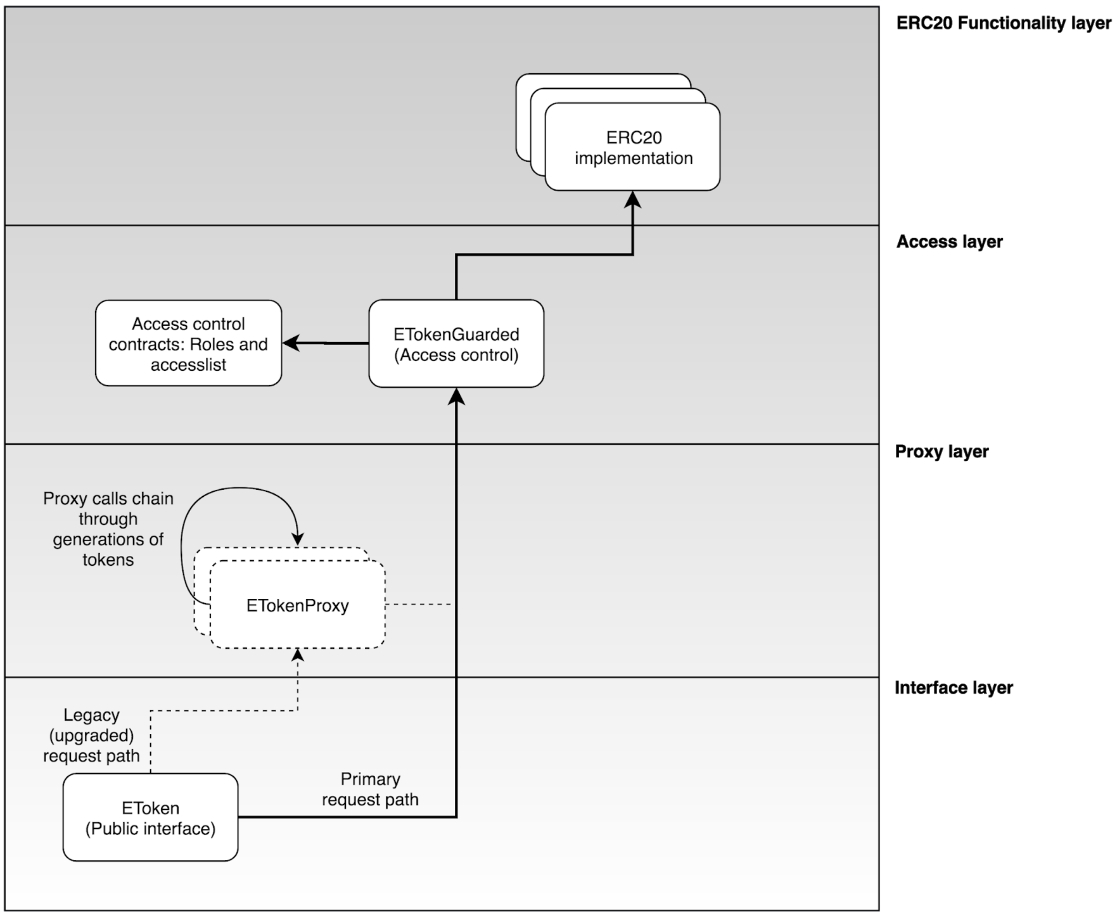
ERC20 Functionality layer
ERC20 implementation
Access layer
Access control contracts: Roles and accesslist
ETokenGuarded (Access control)
Proxy layer
Proxy calls chain through generations of tokens
ETokenProxy
Interface layer
Legacy (upgraded) request path
Primary request path
EToken (Public interface)

Figure 2.2 | Visualization of the layers which comprises the implementation.

Summary

In this paper, we have described the technical underpinnings of our on-chain abstract representation of tokenized assets. Starting from a high-level introduction to tokenized assets, we introduce a specific implementation of the supporting system based on Ethereum smart contracts. At its core, the described solution implements a unified approach to deploying multiple tokenized assets which conform to the established ERC20 standard. This standard describes operations which must be supported by non-fungible tokens in order to allow effortless interoperability with the Ethereum ecosystem. Beyond that, we detail specific challenges related to lifecycle management of such a system when deployed on a state-of-the-art blockchain smart-contract platform. Specifically, in this regard, we describe how controlled mutation of functionality can be achieved in a generally immutable setting. As this is highly security sensitive code, we outline our approach to guarding a subset of the provided functionality using fine-grained roles with restricted capabilities. Finally, continuing our focus on security aspects, we describe the layered design of the system which features a clear separation of concerns and excellent auditability.