SFT Protocol Documentation

Benjamin Hauser, Alex Firmani

Nov 19, 2018

Contents

The SFT protocol is a set of compliance-oriented smart contracts built on the Ethereum blockchain that allow for the tokenization of debt and equity based securities. It provides a robust, flexible framework allowing issuers and investors to retain regulatory compliance throughout primary issuance and multi-jurisdictional secondary trading.

CHAPTER 1

How it works

SFT expands upon the ERC20 token standard. Tokens are transferred via the `transfer` and `transferFrom` functions, however the transfer will only succeed if approved by a series of permissioning modules. A call to `checkTransfer` returns true if the transfer is possible. The standard configuration includes checking a KYC/AML whitelist, tracking investor counts and limits, and restrictions on countries and accredited status. By implementing other modules a variety of additional functionality is possible so as to allow compliance to laws in the countries of the issuer and investors.

CHAPTER 2

Components

1. *Security Token*
    (a) ERC20 compliant tokens intended to represent a claim to ownership of securities
    (b) Modules may be applied to each security token to add additional permissioning or functionality
2. *Issuing Entity*
    (a) Central owner contract for tokens created by the same issuer
    (b) Modules may be applied at this level that introduce permissioning / functionality to all associated security token contracts
3. *KYC Registrar*
    (a) Whitelists that provide identity, region, and accreditation information of investors based on off-chain KYC/AML verification
4. *Custodian*
    (a) Contracts that represent an entity approved to hold tokens for multiple investors
    (b) Base interface that allows for wide customisation depending on the needs of the owner
5. ***Modules***
    (a) Wide range of functionality that modules can hook into allows for many different applications

CHAPTER 3

Testing and Deployment

Unit testing and deployment of this project is performed with brownie.

CHAPTER 4

Third-Party Integration

See *Third Party Integration* for in-depth details.

CHAPTER 5

License

This project is licensed under the Apache 2.0 license.

5.1 Contents

Keyword Index, Search Page

5.1.1 Security Token

SecurityToken represents a single, fungible class of securities from an issuer. It conforms to the ERC20 standard, with an additional `checkTransfer` function available to verify if a transfer will succeed. Before tokens can be transferred, all of the following checks must pass:

- Sender and receiver addresses must be validated by a KYC registrar
- Issuer imposed limits on investor counts: global, country specific, and accreditation rating specific
- Optional permissions added via modules applied at the SecurityToken and IssuingEntity level

Transfers that move tokens between different addressses owned by the same entity (as identified in the KYC registrar) are not as heavily restricted because there is no change of ownership. Any address belonging to a single entity can call `transferFrom` and move tokens from any of their wallets. The issuer can use the same function to move any tokens between any address.

5.1.2 Issuing Entity

Before an issuer can create a security token they must deploy an IssuingEntity contract. This contract has several key purposes:

- Holds a whitelist of associated KYC registries that investor data can be pulled from
- Tracks investor counts and total balances across all security tokens deployed by the issuer

- Enforces permissions relating to investor limits and authorised countries
- Holds a mapping of hashes for legal documents related to the issuer

5.1.3 KYC Registrar

KYCRegistrar contracts are registries that hold information on the identity, region, and rating of investors.

Registries may be maintained by a single entity, or a federation of entities where each are approved to provide identification services for their specific jurisdiction. The contract owner can authorize other entities to add investors within specified countries.

Contract authorities associate addresses to ID hashes that denotes the identity of the investor who owns the address. More than one address may be associated to the same hash. Anyone can call `getID` to see which hash is associated to an address, and then using this ID call functions to query information about the investor's region and accreditation rating.

Registry contracts implement a variation of the standard MultiSig functionality used in other contracts within the protocol. This document assumes familiarity with the standard multi-sig implementation, and will only highlight the differences.

It may be useful to also view the KYCRegistrar.sol source code while reading this document.

Components

Registrars are based on the following key components:

- **Investors** are natural persons or legal entities who have passed KYC/AML checks and are approved to send and receive security tokens. Each investor is assigned a unique ID and is associated with one or more addresses.
- **Authorities** are known, trusted entities that are permitted to add, modify, or restrict investors within the registrar. Authorities are also assigned a unique ID and associated with one or more addresses.
- The **owner** is the initial authority declared during the deployment the contract. Only the owner may add, modify, or restrict other authorities.
- **Issuers** are entities that have created security tokens, who rely on registrars for information about their token holders.

Authorities

The initial owner addresses and threshold are set during deployment. The owner ID is generated as a keccak of the contract address.

The owner may designate authorities using the `addAuthority` function. Authorities do not require explicit permission to call any contract functions. However, they may only add, modify or restrict investors in countries that they have been approved to operate in. This permission is initially declared when creating the authority and may be modified later with `setAuthorityCountries`.

Once an authority has been designated they may use `registerAddresses` or `restrictAddresses` to modify their associated addresses.

Investors

After verifying an investor's KYC/AML, an authority may call `addInvestor` to add the investor to the registrar.

Each investor is identified in the registrar via a unique ID hash. Their country, region, and investor rating are also recorded on-chain. See the Data Standards documentation for detailed information on how this information is generated and formatted.

Investors are also assigned an expiration time for their rating. This is useful in jurisdictions where accreditation status requires periodic reconfirmation. An authority may update the record for an existing investor by calling `updateInvestor`.

Similar to authorities, addresses associated with investors are assigned and restricted via calls to `registerAddresses` or `restrictAddresses`.

Issuer Integration

Issuers must associate their IssuingEntity contract with one or more registrars in order to alow investors to hold their tokens. This is accomplished by calling `IssuingEntity.setRegistrar`.

The investor ID associated with an address may be obtained by calling the `getID` view function. The ID may then be used to call a variety of view functions to obtain the investor's rating, region, country or KYC expiration date.

IssuingEntity contracts primarily rely on `getInvestor` and `getInvestors` to retrieve investor information in the most gas efficient manner possible.

See the Third Party Integration page for detailed information on how to integrate contracts within the protocol.

Security Considerations

Here we outline several unfavorable situations that may occur, and guidelines for how to handle them.

Investor Changes Country

An investor who changes their legal country of residence will necessarily alter their ID hash. In this case the investor should resubmit their KYC/AML to an authority within the new country, receive a new ID hash attached to a new address, and transfer their tokens from their old address to the new one. Their old ID may then be restricted.

Lost Invesor Private Key

An investor who has lost a private key should contact the registry authority and verify their identity off-chain. The authority can then restrict the address of the lost key and add one or more new addresses that the investor controls. The investor may retrieve tokens from the lost address either with assistance from the issuer or by using the `SecurityToken.transferFrom` function. See the SecurityToken documentation for more information on this process.

Compromised Authority

If an authority has been compromised or found to be acting in bad faith, the owner may apply a broad restriction upon them using `setAuthorityRestriction`. This will also restrict every investor that was approved by this authority.

A list of investors that were approved by the restricted authority can be obtained from `NewInvestor` and `UpdatedInvestor` events. Once the KYC/AML of these investors has been re-verified, the restriction upon them may be removed by calling either `updateInvestor` or `setInvestorAuthority`.

Compromised Owner

If the owner is compromised or found to be acting in bad faith, issuers can remove the registrar by calling `IssuingEntity.setRegistrar`. This will also restrict every investor that was approved by this registry. These investors will have to KYC via a different authority in order to be able to transfer their tokens.

5.1.4 Custodian

Custodian contracts allow approved entities to hold tokens on behalf of investors. Common examples of custodians include broker/dealers and secondary markets.

Custodians interact with an issuer's investor counts differently from regular investors. When an investor transfers a balance into the custodian it does not increase the overall investor count, instead the investor is now included in the list of beneficial owners represented by the custodian. Even if the investor now has a balance of 0, they will be still be included in the issuer's investor count.

Custodian contracts include a `transfer` function that optionally allows them to remove an investor from the beneficial owners when sending them tokens.

They may also call `addInvestors` or `removeInvestors` in cases where beneficial ownership has changed from an action happening off-chain. Each custodian must be individually approved by an issuer before they can receive tokens. Because custodians may bypass on-chain compliance checks, it is imperative this approval only be given to known, trusted entities.

5.1.5 MultiSig Implementation

This document outlines the multi-signature, multi-owner functionality used in IssuingEntity and Custodian contracts. Multisig functionality in KYCRegistrar contracts use a similar implementation, you can read about the differences in the registrar documentation.

It may be useful to also view the MultiSig.sol source code while reading this document.

Components

Multisig contracts are based around the following key components:

- **Authorities** are a collection of one or more addresses permitted to call specific admin-level functionality. Each authority is assigned a unique ID.
- The **owner** is the highest authority, capable of creating or restricted other authorities.
- Each authority has a unique **threshold** value, which is the number of required calls to a function before it executes. This value cannot be greater the number of addresses associated with the authority.

Initial Setup

Contracts that implement multisig require 2 arguments in the constructor:

- `address[] _owners`: One or more addresses to associate with the contract owner. The address deploying the contract is not implicitely included within the owner list.
- `uint32 _threshold`: The number of calls required for the owner to perform a multi-sig action.

The owner has the highest level of control over the contract. Associated addresses may always call any admin-level functionality.

Designating Authorities

After deployment the owner may designate authorities using the `AddAuthority` function, which takes the following arguments:

- `address[] _owners`: One or more addresses to associated with the authority.
- `bytes4[] _signatures`: Function signatures that this authority is permitted to call.
- `uint32 _approvedUntil`: The epoch time that this authority is permitted to make calls until. To approve an authority forever, set it to the highest possible uint32 value of 4294967296 (February, 2106).
- `uint32 _threshold`: The number of calls required by this authority to perform a multi-sig action.

Authorities differ from the owner in that they must be explicitely approved to call functions within the contract. These permissions may be modified by the owner via a call to `setAuthoritySignatures`. You can check if an authority is permitted to call a specific function with the view function `isApprovedAuthority`.

Authorities may also be given a time-based restriction, either at the time of creation or by calling `setAuthorityApprovedUntil`. The owner can also restrict an authority by calling this function and setting `_approvedUntil` to 0.

Authorities may add or remove associated addresses with `addAuthorityAddresses` or `removeAuthorityAddresses`. The owner may call this function to add or remove addresses for any authority.

It is important to note that **once an address has been associated to an authority, this association may never be fully removed**. Once an address is removed, that address is now forever unavailable within the protocol. This is necessary to prevent an address from later being associated with a different entity, which could allow for a variety of non-compliant actions. See the KYCRegistrar documentation for more information on this concept.

Calling MultiSig Functions

All multi-sig functions return a single boolean to indicate if the threshold was met and the call succeeded. Functions that implement multi-sig include the following line of code, either at the start or after the initial require statements:

```
if (!_checkMultiSig()) return false;
```

Calls that fail to meet the threshold will trigger an event `MultiSigCall` which includes the current call count and the threshold value. Once a caller meets the threshold the event `MultiSigCallApproved` will trigger, the call will execute, and the call count will be reset to zero.

The number of calls to a function is recorded using a keccak hash of the call data. As such, it is required that each callingn address format their call data in exactly the same way.

Repeating a multi-sig call from the same address before reaching the threshold will revert.

Implementing MultiSig in External Contracts

By calling `checkMultiSigExternal`, it is possible to implement multi-sig functionality in external contracts with the same set of authorities. The function arguments are:

- `bytes4 _sig`: The original function signature being called
- `bytes32 _callHash`: a keccak hash of the original calldata

To implement this in an external contract, you would use the following code:

```
bytes32 _callHash = keccak256(msg.data);
if (!MultiSigContract.checkMultiSigExternal(msg.sig, _callHash)) return false;
```

`checkMultiSigExternal` relies on tx.origin to verify that the original caller is an approved authority. Permissions are chekced against the signature value in the same way as with an internal call. The recorded keccak hash of the call is formed by joining the address of the calling contract, the signature, and the supplied call hash. As such it is impossible to exploit the external call to advance the count on internal multisig events.

An imporant security consideration: If an external contract includes a function with the same signature as a one inside the multi-sig contract, it will be impossible to set unique permissions for each function. Developers and auditors of external contracts should always keep this in mind.

5.1.6 Modules

Issuers may attach modules to IssuingEntity or SecurityToken. When a module is attached, a call to `getBindings` checks the hook points that the module should be called at. Depending on the functionality of the module it may attach at any of the following hook points:

- `checkTransfer`: called to verify permissions before a transfer is allowed
- `transferTokens`: called after a transfer has completed successfully
- `balanceChanged`: called after a balance has changed, such that there was not a corresponding change to another balance (e.g. token minting and burning)

Modules can also directly change the balance of any address. Modules that are active at the IssuingEntity level can call this function on any security token, modules at the SecurityToken level can only call it on the token they are attached to.

When a module is no longer required it can be detached. This should always be done in order to optimize gas costs.

The wide range of functionality that modules can hook into allows for many different applications. Some examples include: crowdsales, country/time based token locks, right of first refusal enforcement, voting rights, dividend payments, tender offers, and bond redemption.

5.1.7 Third Party Integration

KYC Registrar

To setup an investor registry, deploy KYCRegistrar.sol. Owner addresses will then be able to add investors using `addInvestor` or approve other whitelisting authorities with `addAuthority`. See the KYCRegistrar page for a detailed explanation of how to use this contract.

Issuing Tokens

Issuing tokens and being able to transfer them requires the following steps:

1. Deploy IssuingEntity.sol.
2. Call `IssuingEntity.setRegistrar` to add one or more investor registries. You may maintain your own registry and/or use those belonging to trusted third parties.
3. Deploy SecurityToken.sol. Enter the address of the issuer contract from step 1 in the constructor. The total supply of tokens will be initially creditted to the issuer.
4. Call `IssuingEntity.addToken` to attach the token to the issuer.

5. Call `IssuingEntity.setCountries` to approve investors from specific countries to hold the tokens.

At this point, the issuer will be able to transfer tokens to any address that has been whitelisted by one of the approved investor registries *if the investor meets the country and rating requirements*.

Note that the issuer's balance is assigned to the IssuingEntity contract. The issuer can transfer these tokens with a normal call to `SecurityToken.transfer` from any approved address. Sending tokens to any address associated with the issuer will increase the balance on the IssuingEntity contract.

You can also introduce further limitations on investor counts or attach optional modules to add more bespoke functionality. See the IssuingEntity and SecurityToken pages for detailed explanations of how to use these contracts.

Transferring Tokens

SecurityToken.sol is based on the ERC20 Token Standard. Token transfers may be performed in the same ways as any token using this standard. However, in order to send or receive tokens you must also:

- Be approved in one of the KYC registries associated to the token issuer
- Meet the approved country and rating requirements as set by the issuer
- Pass any additional checks set by the issuer

You can check if a transfer will succeed without performing a transaction by calling the `checkTransfer` function of the token contract.

Restrictions imposed on investor limits, approved countries and minimum ratings are only checked when receiving tokens. Unless an address has been explicitely blocked, it will always be able to send an existing balance. For example, an investor may purchase tokens that are only require being accreditted, and then later their accreditation status expires. The investor may still transfer the tokens they already have, but may not receive any more tokens.

Transferring a balance between two addresses associated with the same investor ID does not have the same restrictions imposed, as there is no change of ownership. An investor with multiple addresses may call `transferFrom` to move tokens from any of their addresses without first using the `approve` method. The issuer can also use `transferFrom` to move any investor's tokens, without prior approval.

See the SecurityToken page for a detailed explanation of how to use this contract.

Custodians

To set up a custodian contract to send and receive tokens, you must deploy it and then attach it to an IssuingEntity with `IssuingEntity.addCustodian`. At this point, investors may send tokens into the custodian contract just like they would any other address.

The `Custodian.transfer` function allows you to send tokens out of the contract. You may modify the list of benficial owners using `addInvestors` and `removeInvestors`.

See the Custodian page for a detailed explanation of how to use this contract.

5.1.8 Investor Data Standards

The following generation and format standards should be followed across the SFT protocol to ensure interoperability between network participants.

Investor IDs

Investor IDs are stored as a bytes32 keccak256 hash of the investor's personally identifiable information.

For legal entities, the hash is generated from their Global Legal Entity Identifier (LEI):

> The International Organization for Standardization (ISO) 17442 standard defines a set of attributes or legal entity reference data that are the most essential elements of identification. The Legal Entity Identifier (LEI) code itself is neutral, with no embedded intelligence or country codes that could create unnecessary complexity for users.

For natural persons, a hash is produced from a concatenation of the following:

- Full legal name in all capital letters without spaces
- Date of Birth as DDMMYYYY
- Unique tax ID from current jurisdiction of residence

If any of the malleable fields are changed (via a legal name change or a change of home jurisdictions), the investor will be required to pass KYC/AML again and a new investor ID will be generated. Once KYC is passed, the tokens held in previous addresses must be transferred to addresses associated to the new investor ID. It is impossible to remove or change the ID association of an address.

Country Codes

Based on the ISO-3166-1 numeric standard. Country codes are stored as a uint16 and follow the standard exactly.

A CSV of country and region codes is available 'here <country-and-region-codes.csv>'__.

Region Codes

Based on the ISO 3166-2 standard.

Region codes are stored as a bytes3 and are generated in the following way:

1. Convert each character of the ISO 3166-2 code to it's hexadecimal ASCII code point
2. Concatenate the hex values
3. Pad right where necessary

A quick example to generate region codes using python:

```
iso3166 = "US-AL"[3:]
iso3166 = [hex(ord(i)).replace('0x','') for i in iso3166]
print("0x"+"".join(iso3166)).ljust(6, '0'))
```

- Original code: US-AL
- Resulting bytes32: 0x414c00

A CSV of country and region codes is available 'here <country-and-region-codes.csv>'__.