



# **SFT Protocol**

audit report

## **Table of contents**

<b>Background</b>	<b>3</b>
<b>Update after Phase 1</b>	<b>3</b>
<b>Phase Two</b>	<b>4</b>
Document structure	4
Focus areas	4
Scope	5
Contracts in scope	5
<b>Issues found</b>	<b>6</b>
Severity description	6
Minor	6
Moderate	11
Major	11
Critical	11
<b>Test Suite Analysis</b>	<b>12</b>
Overview	12
Unexpected input testing	12
Suggestions for improvement	12
<b>Observations</b>	<b>13</b>
Block Timestamp (now) reliability	13
Uint underflow	13
<b>Conclusion</b>	<b>14</b>
<b>Disclaimer</b>	<b>16</b>

# Background

This audit report was undertaken by **BlockchainLabs.nz** for the purpose of providing feedback to the **HyperLink** team. It has subsequently been shared publicly without any express or implied warranty.

Solidity contracts were provided by the **HyperLink** development team at this commit[[SFT-Protocol @ 2d7046b621d348b79b7db1661d1138b10dd19638](#)] - we would encourage all community members and token holders to make their own assessment of the contracts.

The audit was performed in **two phases**:

1. System analysis
2. Security audit - *this document*

## Update after Phase 1

### [Changes diff](#)

The contracts have been updated between the time we made our phase 1 report, and when we created this phase 2 report. The updates to the contracts mean that the IssuingEntity contract now use on-chain governance. The actual implementation is not in scope for the audit, but the connecting section in the IssuingEntity is now in place. The updates also made changes to the test suite, adding some new tests and some changes to old tests. We cover the test suite in detail further into this report.

Overall the changes are fairly minor and do not make big enough changes to the core functionality that we would consider doing another phase 1 report or changing our scope. This observation about the update is purely for record keeping as we did not have to change the trajectory of our audit because of the code changes.

# Phase Two

## Document structure

The report will include the following sections:

- Security audit
- Gas optimisations
- Observations
- Conclusion

## Focus areas

<b>Correctness</b>	No correctness defects uncovered during static analysis? No implemented contract violations uncovered during execution? No other generic incorrect behaviour detected during execution? Adherence to adopted standards such as ERC20?
<b>Testability</b>	Test coverage across all functions and events? Test cases for both expected behaviour and failure modes? Settings for easy testing of a range of parameters? No reliance on nested callback functions or console logs? Avoidance of test scenarios calling other test scenarios?
<b>Security</b>	No presence of known security weaknesses? No funds at risk of malicious attempts to withdraw/transfer? No funds at risk of control fraud? Prevention of Integer Overflow or Underflow?
<b>Best Practice</b>	Explicit labeling for the visibility of functions and state variables? Proper management of gas limits and nested execution? The latest version of the Solidity compiler?

# Scope

All smart contracts, test files, migration and deployment scripts from this Github repo:  
<https://github.com/HyperLink-Technology/SFT-Protocol/tree/audit-blz>

Configurational files, documents and other assets **were out** of the scope of this audit.

## Contracts in scope

- Primary smart contracts:
  - bases/KYC.sol
  - bases/Modular.sol
  - bases/MultiSig.sol
  - bases/Token.sol
  - custodians/OwnedCustodian.sol
  - IssuingEntity.sol
  - KYCIssuer.sol
  - KYCRegistrar.sol
  - NFTToken.sol
  - SecurityToken.sol
- Interface contracts
  - interfaces/IBaseCustodian.sol
  - interfaces/IModules.sol
- Inherited smart contracts contained within the repository:
  - open-zeppelin/SafeMath.sol

# Issues found

## Severity description

Minor	A defect that does not have a material impact on the contract execution and is likely to be subjective.
Moderate	A defect that could impact the desired outcome of the contract execution in a specific scenario.
Major	A defect that impacts the desired outcome of the contract execution or introduces a weakness that may be exploited.
Critical	A defect that presents a significant security vulnerability or failure of the contract across a range of scenarios.

## Minor

### **underflow could be more descriptive** *Gas optimization*

In line 70 in the Checkpoint module, the expression 'uint256(-1)' is used to generate a 'uint256' with every bit set to 1.

We toyed with different approaches to achieve this (including 0xff) and compared them in terms of readability, gas usage, and prone to mistakes.

Finally, we recommend the following improvements:

- Usage of a constant so it will have a descriptive name (the use of a constant won't impact the gas usage).
- Using a bitwise negation '~uint256(0)' instead of underflowing will create the same result using 8,110 gas less on deployment and will cost 3 gas more when called.

```
uint256 constant ALL_BITS_TRUE = ~uint256(0);
```

```
function getPermissions()  
    external  
    pure  
    returns  
(  
        bytes4[] memory permissions,  
        bytes4[] memory hooks,  
        uint256 hookBools
```

```

)
{
    permissions = new bytes4[](1);
    permissions[0] = 0xbb2a8522;

    hooks = new bytes4[](3);
    hooks[0] = 0x35a341da;
    hooks[1] = 0x8b5f1240;
    hooks[2] = 0x741b5078;

    return (hooks, permissions, ALL_BITS_TRUE);
}uint256 constant ALL_BITS_TRUE = ~uint256(0);

function getPermissions()
    external
    pure
    returns
(
    bytes4[] memory permissions,
    bytes4[] memory hooks,
    uint256 hookBools
)
{
    permissions = new bytes4[](1);
    permissions[0] = 0xbb2a8522;

    hooks = new bytes4[](3);
    hooks[0] = 0x35a341da;
    hooks[1] = 0x8b5f1240;
    hooks[2] = 0x741b5078;

    return (hooks, permissions, ALL_BITS_TRUE);
}

```

- ✓ Fixed [88e33a8fadfa64](#)

## Favour the use of modifiers in recurrent checks *Best practice*

In the contract bases/Modular.sol, in lines 159, 185, and 214 the same require statement is used.

To improve readability and reduce complexity we recommend the use of a modifier as shown in the following example.

```

function setHook(
    bytes4 _sig,
    bool _active,

```

```


        bool _always
    )
    external
    onlyActiveModule
    returns (bool)
{
    ...
}

modifier onlyActiveModule() {
    require(isActiveModule(msg.sender));
    _;
}

function isActiveModule(address _module) public view returns (bool) {
    return moduleData[_module].active;
}

```

This won't affect the gas used to call the functions.

-  **No fix needed.** Using modifiers increases the gas cost on the deployment of a contract, readability can be sacrificed for the sake of saving gas.

## Avoid magic numbers (NFToken) *Best practice*

There are some magic numbers used in a few functions which could be replaced with named constants for improved readability.

For example

NFToken.sol at line 236

```

236     if (_id[0] == ownerID) {
237         _addr[0] = address(issuer);
238     }
239     if (_id[1] == ownerID) {
240         _addr[1] = address(issuer);
241     }

```

and with named constants:

```

236     if (_id[TRANSFER_FROM] == ownerID) {
237         _addr[TRANSFER_FROM] = address(issuer);
238     }
239     if (_id[TRANSFER_TO] == ownerID) {
240         _addr[TRANSFER_TO] = address(issuer);
241     }

```

-  Fixed [c2bb92d20d](#)



## Better param naming needed 🏷️ *Best practice*

KYCRegistrar.sol line 262

```
252     function setAuthorityRestriction(  
253         bytes32 _id,  
254         bool _permitted  
255     )  
256     external  
257     returns (bool)  
258     {  
259         if (!_checkMultiSig(true)) return false;  
260         require(_id != ownerID, "dev: owner");  
261         require(authorityData[_id].addressCount > 0, "dev: not  
authority");  
262         authorityData[_id].restricted = !_permitted;  
263         emit AuthorityRestriction(_id, _permitted);  
264         return true;  
265     }
```

The variable being changed is 'restricted' but the param's name is 'permitted'. When assigning the value, you use 'NOT' to flip the boolean.  
Suggest keeping naming consistency of these variables for readability and maintenance.

- ✓ Fixed [c2bb92d20d](#)

## Variables naming could be more specific 🏷️ *Best practice*

Be more specific in variables naming: `\_authorityId` rather than `\_id`; that would make the code clearer.

- ✓ Fixed [c2bb92d20d](#)

## Comments could be more informative 🏷️ *Enhancement*

```
167     /**  
168         @notice Private multisig functionality  
169         @dev common logic for _checkMultiSig() and  
checkMultiSigExternal()  
170         @param _id calling authority ID  
171         @param _sig original msg.sig  
172         @param _callHash keccak256 of msg.callhash  
173         @param _sender caller address  
174         @return bool - has call met multisig threshold?  
175     */
```

Some comments do not describe anything besides obvious facts, like common logic for `\_checkMultiSig()` and `checkMultiSigExternal()` at line 169.


- ✓ Fixed [c2bb92d20d](#)

## `_addAddresses()` complexity 📌 *Best practice* 📌 *Enhancement*

High code complexity, e.g.:

```
101 function _addAddresses(  
102     bytes32 _id,  
103     address[] _addr  
104 )  
105     internal  
106     returns (uint32 _count)  
107 {  
108     for (uint256 i; i < _addr.length; i++) {  
109         if (idMap[_addr[i]].id == _id && idMap[_addr[i]].restricted) {  
110             idMap[_addr[i]].restricted = false;  
111         } else if (idMap[_addr[i]].id == 0) {  
112             idMap[_addr[i]].id = _id;  
113         } else {  
114             revert("dev: known address");  
115         }  
116     }  
117     _count = uint32(_addr.length);  
118     emit NewAuthorityAddresses(  
119         _id,  
120         _addr,  
121         authorityData[_id].addressCount.add(_count)  
122     );  
123     return uint32(_count);  
124 }
```


The code complexity/readability could be improved by:

- Separating responsibilities (by splitting into different functions):
- Modifying an addresses restricted status
- Checking if an address already exists
- Adding a new address
- Counting the number of addresses
-  **No fix needed.** Readability can be sacrificed in favour of reducing gas costs.

## Modifiers vs `if()` 📌 *Best practice* 📌 *Enhancement*

```
88 function _onlySelfAuthority(bytes32 _id) internal view {  
89     require (_id != 0);  
90     if (idMap[msg.sender].id != ownerID) {  
91         require(idMap[msg.sender].id == _id, "dev: wrong  
authority");  
92     }  
93 }
```

The best practice is to implement it as a modifier rather than a function.

-  **No fix needed.** Using modifiers increases the gas cost on the deployment of a contract, readability can be sacrificed for the sake of saving gas.

## Moderate

- None found

## Major

- None found

## Critical

- None found

# Test Suite Analysis

*Coverage Summary - Top level contracts*

**IssuingEntity** - 85.4%

**KYCIssuer** - 73.9%

**KYCRegistrar** - 91.2%

**NFTToken** - 88.5%

**OwnedCustodian** - 28.0%

**SecurityToken** - 74.4%

## Overview

The test suite is well written and nicely organised. We were impressed with the amount of tests we saw and the coverage is generally high. In the summary above the **OwnedCustodian** contract coverage looks artificially low, the inherited multisig contract is not being tested for that contract so it drags down the average. The multisig contract has many tests for coverage in the **IssuingEntity** contract tests.

## Unexpected input testing

We noticed many tests for unexpected inputs which is a good sign, a search of the test suite returned 268 results for the `check.reverts` function which is a sign of a healthy test suite.

## Suggestions for improvement

Creating a perfect test suite can be a bit of an art sometimes, you need to find a balance between following every best practice, and the amount of time you can realistically spend on a test suite. For smart contracts the test suite should generally be above average compared to other projects.

**Automation** is an invaluable tool to be paired with a test suite, any changes that want to enter the code base should only be allowed after all tests have passed. Using a tool such as Travis CI which can integrate with Github can help here.

Tests should be **repeatable**, every time you run a test suite it should produce the same results. We had trouble running the tests on some of our machines, using something like docker could be a helpful tool in making sure tests run the same no matter what the environment is.

**Readability** is another important aspect of a test suite, the code should be just as professional and readable as the main codebase. In general these tests were well written but could be improved with comments about the reason for each test. Relating a test to a business assumption/requirement can make bugs more obvious than a test that simply tests that `X()` function reverts.

# Observations

*This section contains notes about issues that likely do not require any action to be addressed, but are not so trivial that we won't mention them.*

## Block Timestamp (now) reliability

Relying on the value of `block.timestamp` might not be safe as the time can be manipulated in the range of about 30 seconds. This doesn't look like it should be a problem for the use cases in this project but it is important to be aware that the time is not guaranteed to be exact and secure.

## Uint underflow

KYCRegistrar.sol at line 436

```
434     require(idMap[msg.sender].id == ownerID, "dev: not owner");
435     Authority storage a = authorityData[_id];
436     a.addressCount -= uint32(_addr.length);
437     require(a.addressCount >= a.multiSigThreshold, "dev: below
threshold");
```

NFTToken.sol at line 319

```
318     _range[_count] = _startRange[i];
319     if (r.stop - _range[_count] >= _value) {
320         return _range;
321     }
322     _value -= (r.stop - _range[_count]);
323     _count++;
```

The lines above can be underflowed, but the transaction is caught later in the function and will be reverted in these cases. We performed functional testing on both the functions above and confirmed that the transaction will always revert. In general it is better to explicitly disallow underflows or overflows using a library like `SafeMath`. This is safer in general since the function is guaranteed to revert in an underflow situation, also it is much easier for someone reading the code to feel secure in a function without having to spend time testing themselves. If these contracts were going to be used primarily by Hyperlink and there was continuity of the knowledge of how the overflows revert in other sections of the code then it wouldn't be a problem, however as SFT is presented as a framework, we assume that people far less knowledgeable than yourself will be deploying or integrating with these contracts and are more likely to inadvertently introduce an underflow condition that doesn't revert.

# Conclusion

The developers demonstrated a strong understanding of Solidity and smart contracts. They were very receptive to the feedback we provided, and implemented most of the changes we suggested. Overall we were very happy with the responsiveness of their development team and were impressed by their adherence to best practices for smart contract development.

We are satisfied that a user's Tokens aren't at risk of being hacked or stolen through interacting with these contracts provided that all user input is correct. We suggested some changes regarding best practices and but in general we noted that the potential attack surface was very low already and the developers clearly had this in mind.

Overall we consider the resulting contracts following the audit feedback period to be robust and have not identified any critical vulnerabilities. Solidity best practices have been followed and the developers have taken care to provide adequate documentation in the smart contracts.

# Disclaimer

Our team uses our current understanding of the best practises for Solidity and Smart Contracts. Development in Solidity and for Blockchain is an emerging area of software engineering which still has a lot of room to grow, hence our current understanding of best practise may not find all of the issues in this code and design.

We have not analysed any of the assembly code generated by the Solidity compiler. We have not verified the deployment process and configurations of the contracts. We have only analysed the code outlined in the scope. We have not verified any of the claims made by any of the organisations behind this code.

Security audits do not warrant bug-free code. We encourage all users interacting with smart contract code to continue to analyse and inform themselves of any risks before interacting with any smart contracts.