SMART CONTRACT AUDIT

zokyo

August 28th 2023 | v. 1.0

Security Audit Score

PASS

Zokyo Security has concluded that this smart contracts passes the security audit.

TECHNICAL SUMMARY

This document outlines the overall security of the STFIL smart contracts evaluated by the Zokyo Security team.

The scope of this audit was to analyze and document the STFIL smart contracts codebase for quality, security, and correctness.

Contract Status

There were 0 critical issues found during the audit. (See Complete Analysis)

It should be noted that this audit is not an endorsement of the reliability or effectiveness of the contracts but rather limited to an assessment of the logic and implementation. In order to ensure a secure contract that can withstand the Ethereum network's fast-paced and rapidly changing environment, we recommend that the STFIL team put in place a bug bounty program to encourage further active analysis of the smart contracts.

Table of Contents

AUDITING STRATEGY AND TECHNIQUES APPLIED

The source code of the smart contract was taken from the STFIL repository:
https://github.com/stfil-io/protocol

Last commit - 1ffacd9eba1212b90f5eb786f934ecbfb1a61a86

Within the scope of this audit, the team of auditors reviewed the following contract(s):

Contracts under the scope:
./stakingpool:
- StakingPool.sol base

./stakingpool/base:
- NodeOperation.sol
- StakingPoolStorage.sol

./tokenization:
- STFILToken.sol
- StableDebtToken.sol
- VariableDebtToken.sol base

./tokenization/base:
- DebtTokenBase.sol

./configuration:
- InterestRateStrategy.sol
- StakingPoolAddressesProvider.sol
- StakingPoolConfigurator.sol
- libraries/configuration/NodeConfiguration.sol

During the audit, Zokyo Security ensured that the contract:

- Implements and adheres to the existing standards appropriately and effectively;
- The documentation and code comments match the logic and behavior;
- Distributes tokens in a manner that matches calculations;
- Follows best practices, efficiently using resources without unnecessary waste;
- Uses methods safe from reentrance attacks;
- Is not affected by the most recent vulnerabilities;
- Meets best practices in code readability, etc.

Zokyo Security has followed best practices and industry-standard techniques to verify the implementation of STFIL smart contracts. To do so, the code was reviewed line by line by our smart contract developers, who documented even minor issues as they were discovered. In summary, our strategies consist largely of manual collaboration between multiple team members at each stage of the review:

01	Due diligence in assessing the overall code quality of the codebase.
02	Cross-comparison with other, similar smart contracts by industry leaders.
03	Thorough manual review of the codebase line by line.

Executive Summary

While no critical vulnerabilities were discovered in the audit, there were identified concerns of varying severity levels such as medium and low, accompanied by certain informational issues. These specifics are expounded upon in the section titled "Comprehensive Analysis."

STRUCTURE AND ORGANIZATION OF THE DOCUMENT

For the ease of navigation, the following sections are arranged from the most to the least critical ones. Issues are tagged as “Resolved” or “Unresolved” or “Acknowledged” depending on whether they have been fixed or addressed. Acknowledged means that the issue was sent to the STFIL team and the STFIL team is aware of it, but they have chosen to not solve it. The issues that are tagged as “Verified” contain unclear or suspicious functionality that either needs explanation from the Client or remains disregarded by the Client. Furthermore, the severity of each issue is written as assessed by the risk of exploitation or other unexpected or otherwise unsafe behavior:

Critical

The issue affects the contract in such a way that funds may be lost, allocated incorrectly, or otherwise result in a significant loss.

High

The issue affects the ability of the contract to compile or operate in a significant way.

Medium

The issue affects the ability of the contract to operate in a way that doesn’t significantly hinder its behavior.

Low

The issue has minimal impact on the contract’s ability to operate.

Informational

The issue has no impact on the contract’s ability to operate.

COMPLETE ANALYSIS

FINDINGS SUMMARY

#	Title	Risk	Status
1	Interest Value Can Be Incorrect If Unstaked Amount Is Sent Back To The Pool	Medium	Resolved
2	Liquidations Should Not Be Paused If Contract Is Paused	Medium	Resolved
3	Owner Not Assigned As Operator Inside delegateBeneficiary	Low	Invalid
4	No Incentive For Liquidator	Low	Acknowledge
5	Centralization Risks	Low	Acknowledge
6	Attacker Can Grief A User Who Is Unstaking	Low	Acknowledge
7	Misleading Event Emitted From a State Changing Action	Low	Resolved
8	Lack of Slippage Argument for Stake and Unstake	Low	Invalid
9	Out of scope Dependencies	Low	Acknowledge
10	Return Value Of Mint Function Not Clear	Informational	Resolved
11	Wrong Error Description	Informational	Resolved
12	Lack Of Documentation	Informational	Resolved
13	Node Owners are Allowed to Liquidate Themselves	Informational	Invalid
14	Missing Emit Events from Privileged Functions	Informational	Resolved
15	Floating Pragma	Informational	Resolved
16	Typos	Informational	Resolved

MEDIUM-1 | RESOLVED

Interest Value Can Be Incorrect If Unstaked Amount Is Sent Back To The Pool

Inside StakingPool.sol , while unstaking at L144 , if the user provides the pool address itself (StakingPool's address) as the *to* address , then essentially the pool address receives the liquidity back increasing liquidity , therefore the interest calculation at L159 would be incorrect since it's *reserve.updateInterestRates(address(this),0,amountToUnstake)* , meaning amountToStake is leaving the pool(liquidity removed) but it's not the case here since the liquidity is being added back.

Recommendation:
There can be 2 solution , the ideal solution would be to have a check *require(to != address(this))* or have an if condition to see if to address is the current address , if it is then Update interest as *reserve.updateInterestRates(address(this),amountToUnstake,0)*

MEDIUM-2 | RESOLVED

Liquidations Should Not Be Paused If Contract Is Paused

The contract StakingPool can be paused by the staking pool configurator in cases of emergency , once paused functions borrow , stake , unstake , withdraw , repay and liquidation can not be executed due to the *whenNotPaused* modifier.

If liquidations are paused , the protocol can incur huge losses due to unhealthy node debt positions . It is always advisable to resume liquidations in cases of emergency too.

When the StakingPool contract is paused , repayments and liquidations are paused . The debt of a node is calculated through its debt tokens (variable and stable) balance. These tokens are interest accruing tokens. It is possible that while the contract is paused , a user's position becomes subject to liquidation.
An attacker can set up a front-running bot , as soon as the contract is resumed the bot calls liquidate on the user's position without giving the user a chance to repay.

Recommendation:
Just resuming liquidations will not be the complete fix , resume repayments too while contract is paused , as if liquidations are unpaused and repayments are paused then positions can be liquidated without giving users the chance to repay/keep their positions healthy.

LOW-1 | INVALID

Owner Not Assigned As Operator Inside delegateBeneficiary

Inside NodeOperations.sol the function delegateBeneficiary should assign the owner as the operator according to the comment at L83 . But this is not done , we only assign the owner as the `0`-address but not the operator as the owner.

Recommendation:
Assign the owner as the operator, then assign the owner as `0` address.

Comment: This was found out to be intented/feature after discussion with the client.

LOW-2 | ACKNOWLEDGED

No Incentive For Liquidator

Liquidations are supposed to clear bad debt of an unhealthy position , and in the process incentivize the liquidator. Inside StakingPool is implemented the liquidation() function at L344, the flow of the liquidation process is as follows →

1. Calculate the debt of the node and then calculate the positionToLiquidate
2. Calculate paybackAfterLiquidation , which is the payback returned to the pool.
3. Burn the bad debt of the node.
4. Withdraw appropriate position from the node to the pool.

Throughout this process , the liquidator(msg.sender) was not rewarded , making this process Infeasible for the liquidator.

Recommendation:
Have a reward system for the liquidator.

Comment: In this protocol, liquidation is a low-frequency operation with low real-time requirements, and it is planned to provide incentives in the governance process.

LOW-3 | ACKNOWLEDGED

Centralization Risks

The CONTRACTS_ADMIN_ROLE holds a lot of power , inside StakingPoolAddressProvider.sol, it can upgrade the proxy to a new implementation address and set the proxy to a new address. These are all most critical functions.

Recommendation:
Ensure the CONTRACTS_ADMIN_ROLE is a multisig or has a timelock for system critical functions.

LOW-4 | ACKNOWLEDGED

Attacker Can Grief A User Who Is Unstaking

When a user unstakes , it is checked that the amount being unstaked is at most the FIL balance in the pool (L156).
An attacker can frontrun a user's unstake call , call borrow() and borrow all or almost all the FIL inside the pool , this would make the user's tx revert.
The attacker can then (after the victim's tx fail) instantly repay the borrow.
This way attacker can continue to grief unstake calls.

Recommendation:
If the pool does not have enough FIL mint the user a different token which can be swapped for FIL later when there is liquidity inside the pool.

Misleading Event Emitted From a State Changing Action

StakingPoolConfigurator.sol -

```
 function setLiquidationFactor(uint256 liquidationFactor) external
onlyPoolAdmin {
    DataTypes.PoolConfigurationMap memory currentConfig =
pool.getReserveData().configuration;
    currentConfig.setLiquidationFactor(liquidationFactor);
    pool.setPoolConfiguration(currentConfig.data);

    emit PoolFeeChanged(liquidationFactor);
  }
```

This mixes things up with

```
function setFee(uint256 fee) external
```

as they emit the same event.

Recommendation:
`setLiquidationFactor` requires a new event created for it to be emitted on calling the function.

Lack of Slippage Argument for Stake and Unstake

StakingPool.sol - In `stake()` a determined amount of native coin is being deposited in exchange of STFIL ERC20 to be minted for the user. The minted amount out relies on a mathematical formula that can be manipulated via a front running attack.

```
ISTFILToken(reserve.stFILAddress).mint(onBehalfOf, amount,
reserve.liquidityIndex);
```

Attackers might front run their transaction in order to affect the value of `reserve.liquidityIndex` from which it affects the minted amount for the user. This generally causes upset among users.

The same issue is also shown in the opposite direction: `unstake()`.

Recommendation:
Add an argument for `minimumAmountReceived` which the user can put so that the transaction should fail if the amount to be received does not fit the user's desire.

Comment: Considered "Invalid" because contract PoolReserveLogic which shows that dependency is out of scope of the Audit.

Out of scope Dependencies

In the contract, `StakingPool.sol` makes critical external calls to the unaudited `reserve` contract to update the state. As such, it is treated as a black box and is assumed to be correct. However, in the real world, contracts can be volatile and unaudited code cannot be guaranteed to work as intended.

In the contract, `STFILToken.sol`, `VariableDebtToken.sol`, and `StableDebtToken.sol` use external unaudited calls to `normalize()`.

Recommendation:
We recommend ensuring that the `PoolReserveLogic.sol` and `FilAddress.sol` contracts contain no errors, and it cannot interfere with future iterations of the protocol.

Comment: In Filecoin, contracts generally have multiple addresses. Two of these address types, f0 and f410f, can be converted to 0x-style (Ethereum) addresses which can be used in the CALL opcode. Refer to https://docs.filecoin.io/smart-contracts/filecoin-evm-runtime/differences-with-ethereum/#multiple-addresses.

INFORMATIONAL-1 | RESOLVED

Return Value Of Mint Function Not Clear

Inside VariableDebtToken's mint() function at L72 we return true if the previous balance was `0` , the significance of returning true for this case is unclear. If it is essential for the function to return true , then it's return value should be checked inside StakingPool contract at L205 Similarly , true is returned for StableDebtToken too.

Recommendation:
Add more clarity on the purpose of returning true if the previous balance is `0` , and if it is significant check the returned value.

INFORMATIONAL-2 | RESOLVED

Wrong Error Description

Inside StakingPool.sol at L355 , the error is DEBT_RATE_MORE_THAN_LIQUIDATION_THRESHOLD , while it should have been DEBT_RATE_LESS_THAN_LIQUIDATION_THRESHOLD

Recommendation:
Change to the above suggested name.

INFORMATIONAL-3 | RESOLVED

Lack Of Documentation

There are a lot of parts in the code which are not well documented/lack natspec. Inside StableDebtToken there are no comments about the state variables. Other areas could have more in-depth information. As liquid staking is a highly complex topic, there are key areas that need to be addressed for users to understand the protocol. With derivatives available to users, this adds more complexity to the issue. Users should understand exactly what their leverage gains them. For Example: Explaining how leveraging gains them a larger passive "Block Reward". Explaining the use case for liquid staking and what makes this important.

Recommendation:
Adding sufficient documentation for all users to understand what liquid staking is and how leverage affects this.

Recommendation:
Have property comments/natspec for state variables too.

INFORMATIONAL-4 | INVALID

Node Owners are Allowed to Liquidate Themselves

In StakingPool.sol - Function `liquidation(uint64 actorId)` enables the node owners to liquidate themselves. This is an uncommon pattern in DeFi projects as the owner repays debt while other wallets can liquidate the owner when the position approaches insolvency.

Fix: This is not having any negative effect on contract's security and being informed by client that this is in fact needed as a spec. Therefore the issue is considered invalid.

Recommendation:
Validate that caller is not the node owner.

INFORMATIONAL-5 | RESOLVED

Missing Emit Events from Privileged Functions

In StakingPool.sol

```
 function setPoolConfiguration(uint256 configuration) external override
onlyStakingPoolConfigurator {
    _reserve.configuration.data = configuration;
  }
```

In NodeOperation.sol

```
 function setNodeConfiguration(uint64 actorId, uint256 configuration)
external override onlyStakingPoolConfigurator {
    _nodes[actorId].configuration.data = configuration;
  }
```

Recommendation:
Emit events on important changes.

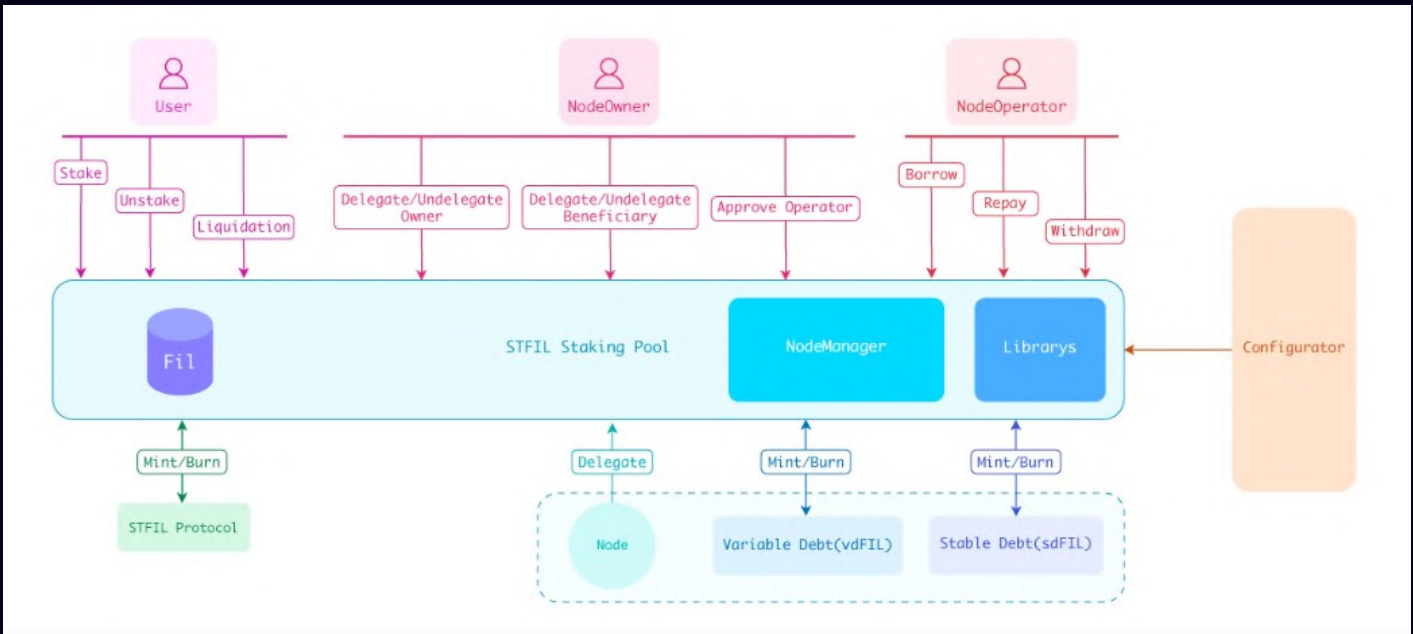
INFORMATIONAL-6 | RESOLVED

Floating Pragma

Throughout the codebase, the contracts that are unlocked at version ^0.8.0, and it should always be deployed with the same compiler version. By locking the pragma to a specific version, contracts are not accidentally getting deployed by using an outdated version that can introduce unintended consequences.

Recommendation:
Lock the compiler version to a specific. Known bugs are featured here.

Typos



In the following documentation diagram, Librarys is spelled incorrectly and should be changed to Libraries.

Recommendation:
It is recommended that the typos be changed.

	StakingPool.sol base NodeOperation.sol StakingPoolStorage.sol STFILToken.sol StableDebtToken.sol VariableDebtToken.sol base
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

	DebtTokenBase.sol InterestRateStrategy.sol StakingPoolAddressesProvider.sol StakingPoolConfigurator.sol libraries/configuration/ NodeConfiguration.sol
Re-entrancy	Pass
Access Management Hierarchy	Pass
Arithmetic Over/Under Flows	Pass
Unexpected Ether	Pass
Delegatecall	Pass
Default Public Visibility	Pass
Hidden Malicious Code	Pass
Entropy Illusion (Lack of Randomness)	Pass
External Contract Referencing	Pass
Short Address/ Parameter Attack	Pass
Unchecked CALL Return Values	Pass
Race Conditions / Front Running	Pass
General Denial Of Service (DOS)	Pass
Uninitialized Storage Pointers	Pass
Floating Points and Precision	Pass
Tx.Origin Authentication	Pass
Signatures Replay	Pass
Pool Asset Security (backdoors in the underlying ERC-20)	Pass

We are grateful for the opportunity to work with the STFIL team.

The statements made in this document should not be interpreted as an investment or legal advice, nor should its authors be held accountable for the decisions made based on them.

Zokyo Security recommends the STFIL team put in place a bug bounty program to encourage further analysis of the smart contract by third parties.

zokyo