QuillAudits

Audit Report
December, 2022

For

Voltage

Table of Content

Executive Summary

Project Name	Voltage Finance
Overview	Voltage Finance,Fuse Liquid Staking allows for anyone to stake Fuse tokens via a Consensus contract to a validator securing the Fuse Network as delegated tokens. In return users get Liquid Staked Fuse Tokens (sFuse) which earn rewards which are updated on deposit and withdrawal activities by users. sFuse are minted and burned as required for deposits and withdrawals and amounts calculated using a price ratio that starts as 1e18 but updates based on ratio delegated Fuse and Staked Fuse supply. Upgradeable using OpenZeppelin Proxy Liquid Staking Pool contract interacts with the Consensus Contracts for deposits, pool staking to validators, withdrawals, receiving block rewards etc
Timeline	24 November,2022 - 9th November,2022
Method	Manual Review, Functional Testing, Automated Testing etc.
Scope of Audit	The scope of this audit was to analyse Voltage Finance(Fuse Fi) Liquid Staking Smart Contracts codebase for quality, security, and correctness.This included testing of smart contracts to ensure proper logic was followed, manual analysis ,checking for bugs and vulnerabilities, checks for dead code, checks for code style, security and more. The audited contracts are as follows:
Git Repo link	*https://github.com/voltfinance/fuse-liquid-staking* **Commit Hash:** d39d6cf502e08f5e8186e6329d56868436f56278
Fixed in	*https://github.com/voltfinance/fuse-liquid-staking/blob/audit-resolve/contracts/LiquidStakingPool.sol* **Commit Hash:** 8c4d99ac22693f756760423da7342bc77d24e0e5
Mainnet Address:	*https://explorer.fuse.io/address/0x370C54A964BD03633741a3517dF01E6c531CabA8/contracts*

Executive Summary

High
Medium
Low
Informational

	High	Medium	Low	Informational
Open Issues	0	0	0	0
Acknowledged Issues	0	0	0	5
Partially Resolved Issues	0	0	0	0
Resolved Issues	0	0	2	0

Types of Severities

High

A high severity issue or vulnerability means that your smart contract can be exploited. Issues on this level are critical to the smart contract's performance or functionality, and we recommend these issues be fixed before moving to a live environment.

Medium

The issues marked as medium severity usually arise because of errors and deficiencies in the smart contract code. Issues on this level could potentially bring problems, and they should still be fixed.

Low

Low-level severity issues can cause minor impact and or are just warnings that can remain unfixed for now. It would be better to fix these issues at some point in the future.

Informational

These are severity issues that indicate an improvement request, a general question, a cosmetic or documentation error, or a request for information. There is low-to-no impact.

Types of Issues

Open

Security vulnerabilities identified that must be resolved and are currently unresolved.

Resolved

These are the issues identified in the initial audit and have been successfully fixed.

Acknowledged

Vulnerabilities which have been acknowledged but are yet to be resolved.

Partially Resolved

Considerable efforts have been invested to reduce the risk/impact of the security issue, but are not completely resolved.

Checked Vulnerabilities

- Re-entrancy
- Timestamp Dependence
- Gas Limit and Loops
- Exception Disorder
- Gasless Send
- Use of tx.origin
- Compiler version not fixed
- Address hardcoded
- Divide before multiply
- Integer overflow/underflow
- Dangerous strict equalities
- Tautology or contradiction
- Return values of low-level calls
- Missing Zero Address Validation
- Private modifier
- Revert/require functions
- Using block.timestamp
- Multiple Sends
- Using SHA3
- Using suicide
- Using throw
- Using inline assembly

Techniques and Methods

Throughout the audit of smart contract, care was taken to ensure:

- The overall quality of code.
- Use of best practices.
- Code documentation and comments match logic and expected behaviour.
- Token distribution and calculations are as per the intended behaviour mentioned in the whitepaper.
- Implementation of ERC-20 token standards.
- Efficient use of gas.
- Code is safe from re-entrancy and other vulnerabilities.

The following techniques, methods and tools were used to review all the smart contracts.

Structural Analysis

In this step, we have analysed the design patterns and structure of smart contracts. A thorough check was done to ensure the smart contract is structured in a way that will not result in future problems.

Static Analysis

Static analysis of smart contracts was done to identify contract vulnerabilities. In this step, a series of automated tools are used to test the security of smart contracts.

Code Review / Manual Analysis

Manual analysis or review of code was done to identify new vulnerabilities or verify the vulnerabilities found during the static analysis. Contracts were completely manually analysed, their logic was checked and compared with the one described in the whitepaper. Besides, the results of the automated analysis were manually verified.

Gas Consumption

In this step, we have checked the behaviour of smart contracts in production. Checks were done to know how much gas gets consumed and the possibilities of optimization of code to reduce gas consumption.

Tools and Platforms used for Audit

Remix IDE, Truffle, Truffle Team, Solhint, Mythril, Slither, Solidity statistic analysis.

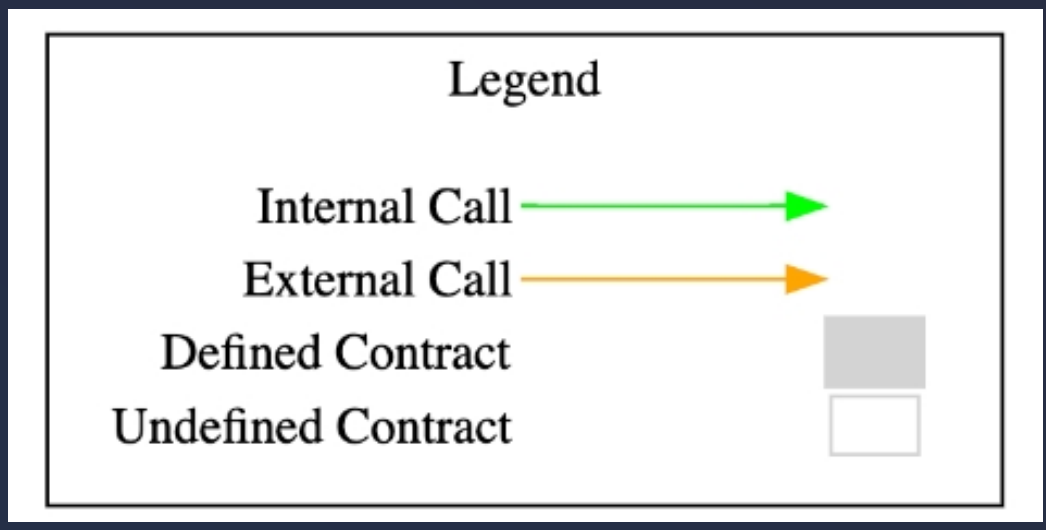
Contracts Information

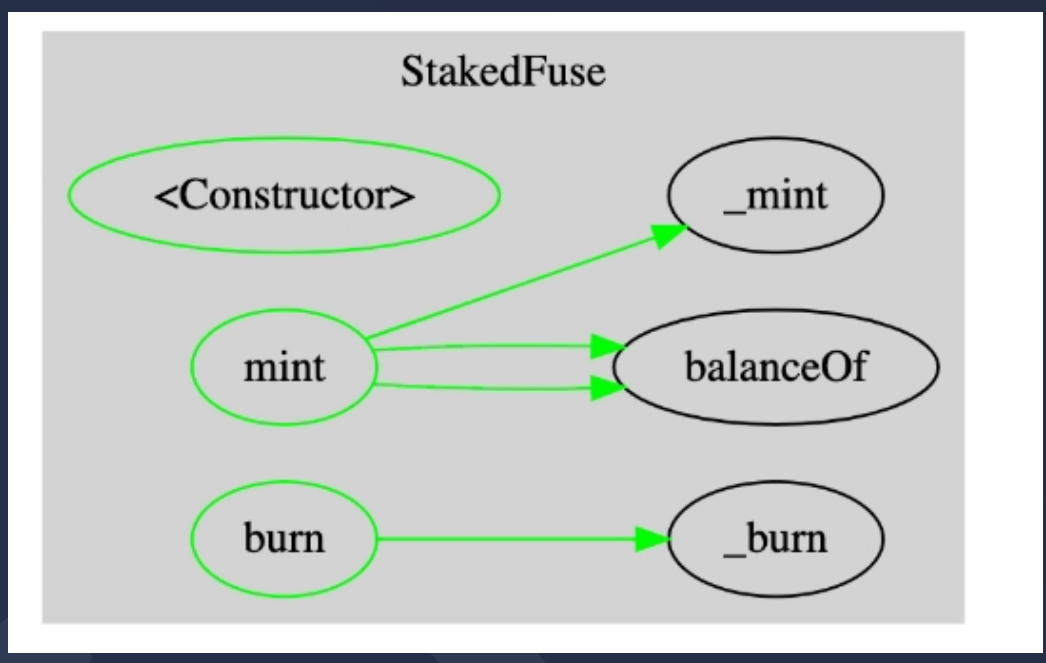
Contract	Lines	Complexity Score	Capabilities
contracts/LiquidStakingPool.sol	983	443	payable functions
contracts/interfaces/IToken.sol	8	5	
contracts/interfaces/IConsensus.sol	17	14	payable functions
contracts/token/StakedFuse.sol	36	16	
TOTALS	1044	478	payable functions

Dependency / Import Path	Count
@openzeppelin/contracts-upgradeable/access/OwnableUpgradeable.sol	1
@openzeppelin/contracts-upgradeable/proxy/utils/Initializable.sol	1
@openzeppelin/contracts-upgradeable/proxy/utils/UUPSUpgradeable.sol	1
@openzeppelin/contracts-upgradeable/security/ReentrancyGuardUpgradeable.sol	1
@openzeppelin/contracts/security/ReentrancyGuard.sol	1
@openzeppelin/contracts/access/Ownable.sol	1
@openzeppelin/contracts/token/ERC20/ERC20.sol	1
@openzeppelin/contracts/token/ERC20/IERC20.sol	1
@openzeppelin/contracts/utils/math/SafeMath.sol	1

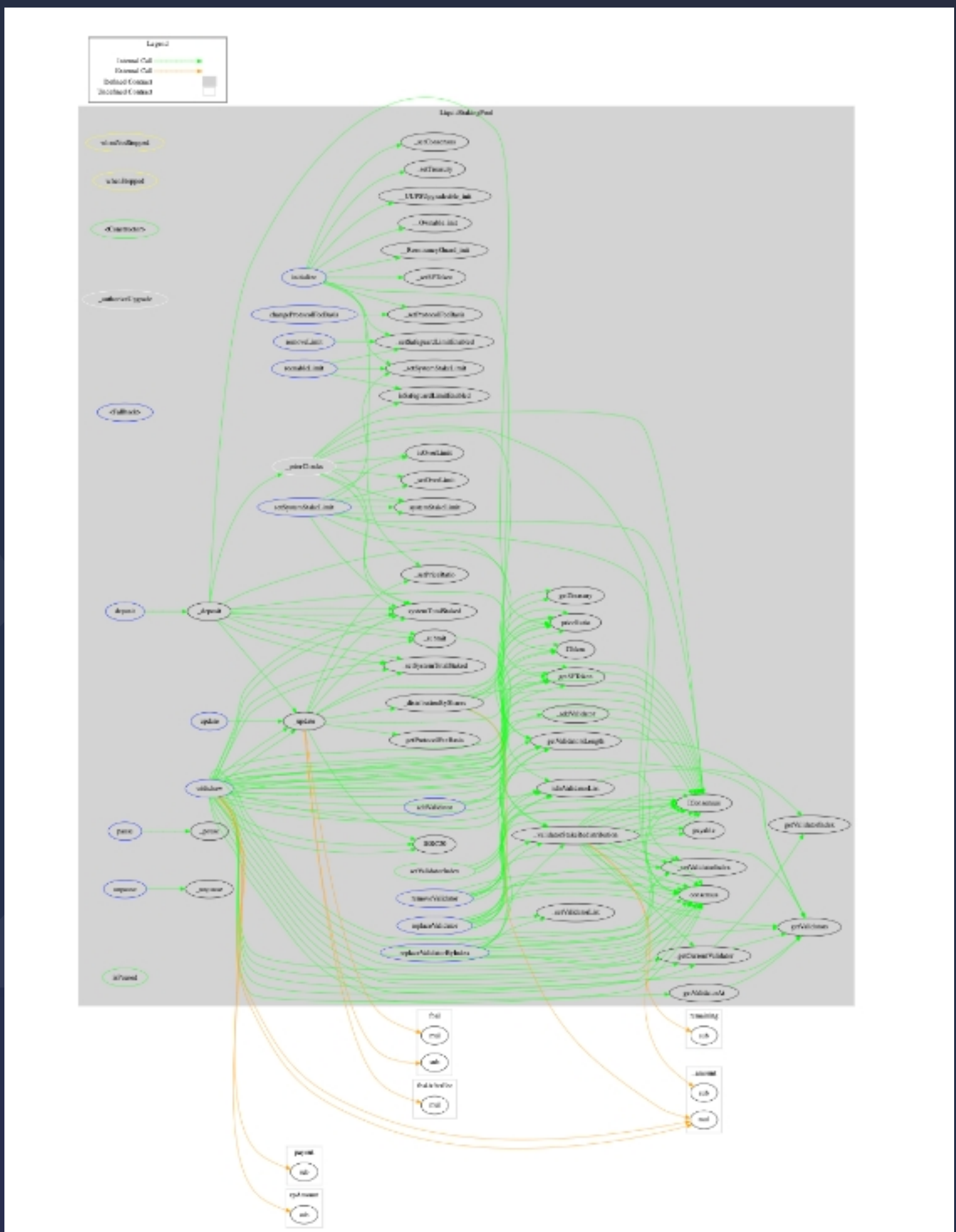
Call Graphs

- contracts/token/StakedFuse.sol





- contracts/token/StakedFuse.sol



Manual Testing

High Severity Issues

No issues found

Medium Severity Issues

No issues found

Low Severity Issues

A1. Centralization risks and overpowered ownership

Description

LiquidStakingPool.sol is an Ownable and Upgradeable contract that allows the owner to control critical functions and operations of the contracts. Critical functions and actions such as pausing, upgrading contracts, setting stake limits, removing or adding validators, setting fees etc. Currently this account appears to be a single address and not multisig. There is no indication of how security of keys will be managed to avoid wrongful access of the owner account. If single addresses this can bring about centralization risks.

Remediation

It may be prudent to have a multisig account or DAO Governance control owner role.

Auditor's Response: Transfers ownership of Liquid Staking Proxy to Protocol Multisig

Status

Resolved

A2. Unchecked return values

Description

Certain operations in functions are not checking the return values. This results in a risk processing function with assumption intermediate steps was a success when it could have failed, especially when returning booleans. Consider the following lines of code:

LiquidStakingPool.sol line 360 IERC20(getSFToken()).transferFrom(msg.sender, address(this), _amount)
LiquidStakingPool.sol line 292 IERC20(getSFToken()).transferFrom(msg.sender, address(this), _amount);
LiquidStakingPool.sol line 727 IToken(getSFToken()).mint(msg.sender, tokens);
LiquidStakingPool.sol line 777 IToken(getSFToken()).mint(getTreasury(), tokens);

Remediation

It is recommended to check all return values such as in above cases and any other cases in code that may not be specified above
E.g require(IToken(getSFToken()).mint(msg.sender, tokens), “Mint failed”);

Auditor's Response: Return value check applied using require() on mint transactions.

Status

Resolved

Informational Issues

A3. Floating pragma

Description

Contracts make use of pragma ^0.8.0 which allows for solidity compiler versions from 0.8.0 to the version just before 0.9.0. This can result in different versions being used for testing and production.

Remediation

It is recommended to deploy contracts using the same compiler version/flags with which they have been tested. The solidity version must be fixed by locking the pragma by avoiding using ^. Consider using fixed pragma like 0.8.6 or 0.8.11 etc

Status

Acknowledged

A4. Obsolete library

Description

Contracts make use of pragma ^0.8.0 and from this version breaking changes were made to allow checks for overflow and underflow by default within the compiler. Adding this library adds to the code unnecessarily.

Remediation

It is recommended to remove the SafeMath library and make use of normal arithmetic where needed.

Status

Acknowledged

A5. Public functions that can be made external

Description

Contract is using old solidity version 0.6.12, Using an old version prevents access to new Solidity security checks.

Remediation

Consider using the latest solidity version.

Status

Acknowledged

A6. require() missing error strings

Description

Some require() statements are missing error strings .
LiquidStakingPool.sol line 199 require(_initialValidator != address(0));
LiquidStakingPool.sol line 247 require(_amount < systemMaxStake);
LiquidStakingPool.sol line 271 require(!isSafeguardLimitEnabled())
LiquidStakingPool.sol line 291 require(_amount > 0);
LiquidStakingPool.sol line 354 require(_amount > 0);
LiquidStkingPool.sol line 371 require(payout <= withdrawableAmount);
LiquidStakingPool.sol line 413 require(_amount <= 2000);
LiquidStakingPool.sol line 481 require(_replacement != address(0));
LiquidStakingPool.sol line 529 require(_replacement != address(0));
LiquidStakingPool.sol line 567 require(getValidatorsLength() > 0);
LiquidStakingPool.sol line 568 require(_index <= getValidatorsLength().sub(1));
LiquidStakingPool.sol line 693
require(systemTotalStaked().add(msg.value) <= systemMaxStake);

Remediation

It is recommended that all require() statements have an error string to describe the failure. E.g require(_initialValidator!= address(0), “zero address not allowed validator”);

Status

Acknowledged

A7. Contract layout

Description

It is considered best practice to order the Solidity file from variables, events, modifiers, constructor, fallback, external functions, public functions, internal functions and private functions in that order for readability and gas savings.

LiquidStakingPool.sol line 988
function getValidatorAt(uint256 _position) public view returns (address) {.. function is in between internal functions.

Remediation

It is recommended to move this function to where all other public functions are written. However it is also best to keep in mind any readability, gas optimisation issues in case that is relevant to current ordering

Status

Acknowledged

Functional Testing

constructor		PASS
mint	function	PASS
burn	function	PASS
Burned	interface	PASS

constructor		PASS
Deposited	Event	PASS
Withdrawn	Event	PASS
Burned	Event	PASS
ChangedValidatorIndex	Event	PASS
UpdatedPriceRatio	Event	PASS
DistributedProtocolFee	Event	PASS
NewSystemStakeLimit	Event	PASS
DisabledSafeguard	Event	PASS
ReenabledSafeguard	Event	PASS
ChangedProtocolFee	Event	PASS
AddedValidator	Event	PASS
RemovedValidator	Event	PASS
ReplacedValidator	Event	PASS
Paused	Event	PASS
Unpaused	Event	PASS
whenNotStopped	Modifier	PASS

Functional Testing

whenStopped	Modifier	PASS
initialize	function	PASS
_authorizeUpgrade	function	PASS
fallback	function	PASS
deposit	function	PASS
setSystemStakeLimit	function	PASS
removeLimit	function	PASS
reenableLimit	function	PASS
update	function	PASS

withdraw(uint)	function	PASS
withdraw(uint,uint)	function	PASS
pause	function	PASS
unpause	function	PASS
changeProtocolFeeBasis	function	PASS
addValidator	function	PASS
removeValidator	function	PASS
replaceValidator	function	PASS
replaceValidatorByIndex	function	PASS
setValidatorIndex	function	PASS
priceRatio	function	PASS
systemStakeLimit	function	PASS
systemTotalStaked	function	PASS
isSafeguardLimitEnabled	function	PASS
isOverLimit	function	PASS
getTreasury	function	PASS
consensus	function	PASS
getSFToken	function	PASS
getValidators	function	PASS
getValidatorsLength	function	PASS
isInValidatorList	function	PASS
getValidatorIndex	function	PASS
getProtocolFeeBasis	function	PASS
isPaused	function	PASS
_priorChecks	function	PASS
_deposit	function	PASS
_submit	function	PASS
_update	function	PASS

_distributionByShares	function	PASS
_validatorStakeRe	function	PASS
_setPriceRatio	function	PASS
_setSystemStakeLimit	function	PASS
_setSystemTotalStaked	function	PASS
_setSafeguardLimitEnabled	function	PASS
_setOverLimit	function	PASS
_setTreasury	function	PASS
_setConsensus	function	PASS
_setSFToken	function	PASS
_addValidator	function	PASS
_setValidatorList	function	PASS
_setValidatorIndex	function	PASS
_setProtocolFeeBasis	function	PASS
_pause	function	PASS
_unpause	function	PASS
getValidatorAt	function	PASS
getValidatorAt	function	PASS
_getCurrentValidator	function	PASS

```
LiquidStaking - Admin
  init
    ✔ should initialize with correct parameters (498ms)
    ✔ should fail if already initialized (435ms)
    - should fail with invalid initial validator param
    - should fail with invalid consensus param
    - should fail with invalid stakedFuse param
    - should fail with invalid treasury param
  initialized
    system stake limit
      ✔ owner should set system stake limit (246ms)
      ✔ should fail if not owner
      ✔ should fail if setting the same limit
      ✔ should fail if amount is more than max system stake
    pause
      ✔ owner can pause (118ms)
      ✔ should fail if not owner
    unpause
      ✔ owner can unpause (114ms)
      ✔ should fail if not owner
    protocol fee basis
      ✔ owner can change fee (116ms)
      ✔ should fail if not owner
      ✔ amount should be less than 20%
    validator
      add validator
        ✔ owner can add validator (152ms)
        ✔ should fail if not owner
        ✔ should fail if already in list
      remove validator
        ✔ owner can remove validator (268ms)
        ✔ should redistribute delegated amount to remaining validators (1279ms)
        ✔ should fail if not owner
        ✔ should fail if only validator (285ms)
      replace validator
        ✔ owner can replace validator (352ms)
        ✔ should redistribute delegated amount to remaining validators (1750ms)
        ✔ should fail if not owner (45ms)
        ✔ should fail if new validator not in list
        ✔ should fail if old validator not in list (166ms)
```

```
  ✔ should fail if only validator (285ms)
replace validator
  ✔ owner can replace validator (352ms)
  ✔ should redistribute delegated amount to remaining validators (1750ms)
  ✔ should fail if not owner (45ms)
  ✔ should fail if new validator not in list
  ✔ should fail if old validator not in list (166ms)
replace validator by index
  ✔ owner can add validator (251ms)
  ✔ should redistribute delegated amount to remaining validators (1139ms)
  ✔ should fail if not owner
  ✔ should fail if index is invalid
  ✔ should fail if validator in list (142ms)
```

```
      set validator index
        ✔ owner can add validator (257ms)
        ✔ should fail if not owner
    limit
      disable limit
        ✔ owner can disable limit (95ms)
        ✔ should fail if not owner
      enable limit
        ✔ owner can enable limit (216ms)
        ✔ should fail if not owner (105ms)

LiquidStaking - Deposit
  deposit
    ✔ user can deposit when sf supply is zero (585ms)
    ✔ user can deposit when sf supply is greater than zero (929ms)
    ✔ user can deposit amount when it exceeds current validator (1260ms)
    ✔ user can deposit after system update (1883ms)
    - should send fees to treasury on update
    ✔ should fail if amount is zero
    ✔ should fail if amount exceeds capacity (45ms)

LiquidStaking - Withdraw
  withdraw
    ✔ user can withdraw funds (489ms)
    ✔ user can withdraw after manual system update (1157ms)
    ✔ user can withdraw after auto system update (1135ms)
    ✔ user can withdraw when amount exceeds current validator amount (4958ms)
    ✔ should fail if amount 0
```

Automated Tests

No major issues were found. Some false positive errors were reported by the tools. All the other issues have been categorized above according to their level of severity.

Closing Summary

Some issues of Low Severity and Informational nature were found in this audit. Some suggestions and best practices are also provided in order to improve the code quality and security posture.

Disclaimer

QuillAudits smart contract audit is not a security warranty, investment advice, or an endorsement of the Voltage Fi Platform. This audit does not provide a security or correctness guarantee of the audited smart contracts.

The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them. Securing smart contracts is a multistep process. One audit cannot be considered enough. We recommend that the Voltage Fi Team put in place a bug bounty program to encourage further analysis of the smart contract by other third parties.

About QuillAudits

QuillAudits is a secure smart contracts audit platform designed by QuillHash Technologies. We are a team of dedicated blockchain security experts and smart contract auditors determined to ensure that Smart Contract-based Web3 projects can avail the latest and best security solutions to operate in a trustworthy and risk-free ecosystem.

700+
Audits Completed

$15B
Secured

700K
Lines of Code Audited

Follow Our Journey

Audit Report
December, 2022

For

Voltage

QuillAudits

Canada, India, Singapore, United Kingdom

audits.quillhash.com

audits@quillhash.com