Blaize.Security

August 22nd, 2023 / V. 1.0

01NODE

SMART CONTRACT AUDIT

TABLE OF CONTENTS

AUDIT RATING

SCORE **9.7** /10

The scope of the project includes 01node set of contracts:

Staking Pools
contracts/NodeStakingPoolV220.sol
contracts/NodeLiquidETH.sol
contracts/OracleManager.sol
contracts/RewardManager.sol

Repository:

https://github.com/01node/staking-pool-v2-contracts

Branch: main

Initial commit:

- 440e41b6e2fbe2352c7c76d802ae3ffd54d366a1

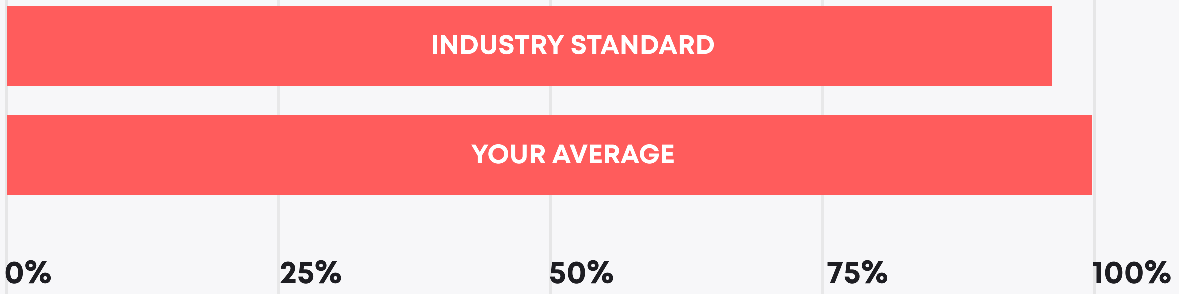
Final commit:

- d62c81e831b0e581e2449753c96e00553efad2e8

TECHNICAL SUMMARY

During the audit, we examined the security of smart contracts for the 01node protocol. Our task was to find and describe any security issues in the smart contracts of the platform. This report presents the findings of the security audit of the **01node** smart contracts conducted between **August 7th, 2023** and **August 22nd, 2023.**

Testable code



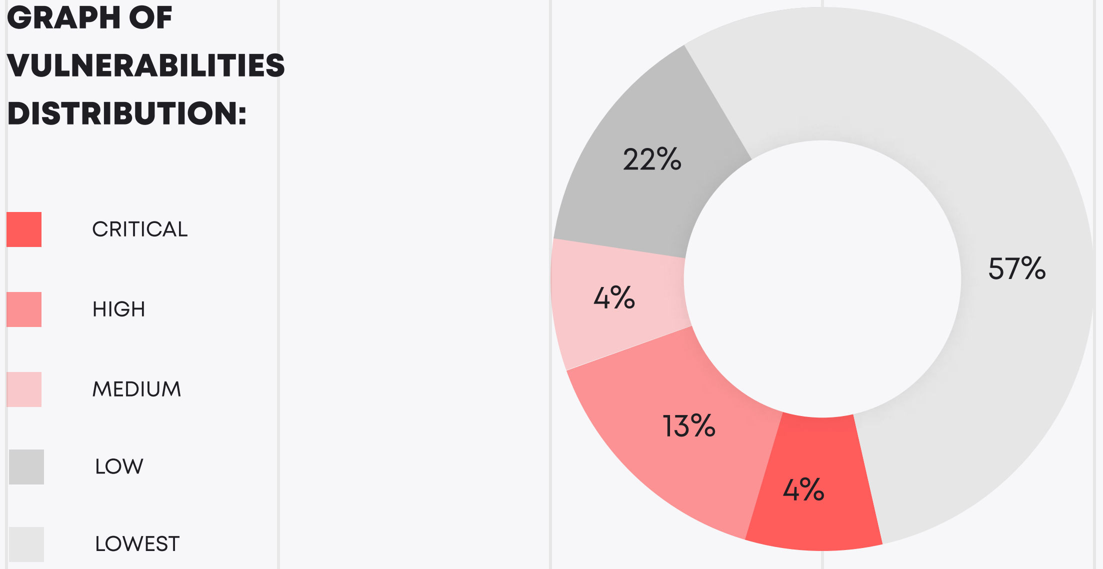
The code is 100% testable, which is above the industry standard of 95%.

The audit scope includes all tests and scripts, documentation, and requirements presented by the **01node** team. The coverage is calculated based on the set of Hardhat framework tests and scripts from additional testing strategies, and includes testable code from manual and exploratory rounds.

However, to ensure the security of the contract, the **Blaize.Security** team suggests that the **01node** team follow post-audit steps:

1. launch **active protection** over the deployed contracts to have a system of early detection and alerts for malicious activity. We recommend the AI-powered threat prevention platform **VigiLens**, by the **CyVers** team.
2. launch a **bug bounty program** to encourage further active analysis of the smart contracts.

GRAPH OF VULNERABILITIES DISTRIBUTION:



The table below shows the number of the detected issues and their severity. A total of 23 problems were found. 23 issues were fixed or verified by the 01node team.

	FOUND	FIXED/VERIFIED
Critical	1	1
High	3	3
Medium	1	1
Low	5	5
Lowest	13	13

SEVERITY DEFINITION

🟥 **Critical**

The system contains several issues ranked as very serious and dangerous for users and the secure work of the system. Requires immediate fixes and a further check.

🟥 **High**

The system contains a couple of serious issues, which lead to unreliable work of the system and migh cause a huge data or financial leak. Requires immediate fixes and a further check.

🟥 **Medium**

The system contains issues that may lead to medium financial loss or users' private information leak. Requires immediate fixes and a further check.

⬜ **Low**

The system contains several risks ranked as relatively small with the low impact on the users' information and financial security. Requires fixes.

⬜ **Lowest**

The system does not contain any issues critical to the secure work of the system, yet is relevant for best practices

AUDITING STRATEGY AND TECHNIQUES APPLIED/PROCEDURE

We have scanned this smart contract for commonly known and more specific vulnerabilities:

- Unsafe type inference;
- Timestamp Dependence;
- Reentrancy;
- Implicit visibility level;
- Gas Limit and Loops;
- Transaction-Ordering Dependence;
- Unchecked external call - Unchecked math;
- DoS with Block Gas Limit;
- DoS with (unexpected) Throw;
- Byte array vulnerabilities;
- Malicious libraries;
- Style guide violation;
- ERC20 API violation;
- Uninitialized state/storage/ local variables;
- Compile version not fixed.

Procedure

We checked the contract for the following parameters:

- Whether the contract is secure;
- Whether the contract corresponds to the documentation;
- Whether the contract meets the best practices in the efficient use of gas, code readability.

Automated analysis:

Scanning contracts by several publicly available automated analysis tools such as Mythril, Solhint, Slither, and Smartdec. Manual verification of all the issues found with tools.

Manual audit:

Manual analysis of smart contracts for security vulnerabilities. We checked smart contract logic and compared it with the one described in the documentation.

EXECUTIVE SUMMARY

The Blaize Security team conducted an audit of the 01node protocol, which allows users to stake their ETH and receive rewards. Users get LiquidETH tokens in exchange for verifying their staking on the protocol. By integrating with the SSV protocol, 01node can register a hardware infrastructure (operator) and an Ethereum validator by sending 32 ETH to the Beacon deposit smart contract.

The audit aimed to verify the security of staking, withdrawals, storage writing, and the rewards system. Additionally, we checked the smart contract against the list of common vulnerabilities, as well as our internal checklist and gas optimizations.

During the audit, a critical issue was identified regarding the reward balance for users. Consequently, when a new validator was added, the rewards were reset. Along with this critical issue, several high-risk problems were found that were related to the old transfer method for ETH, variable value writing, and storage updating. The 01node team successfully fixed all the issues that were found.

Other issues were associated with the lack of events, variable validation and usage, and several gas optimizations. All of them also were successfully fixed.

The overall security of the smart contract is high enough: it has passed all the security checks. However, the 01node team has not provided us with smart contract unit tests or technical documentation. Therefore, there is no integrity checks to perform during the further development, so, future features integration may affect the existing business logic. Also, the code quality has the potential to improve readability and optimization. All these facts are reflected in the final mark. Nevertheless, our team has thoroughly tested the entire set and made sure that the protocol passes necessary security checks.

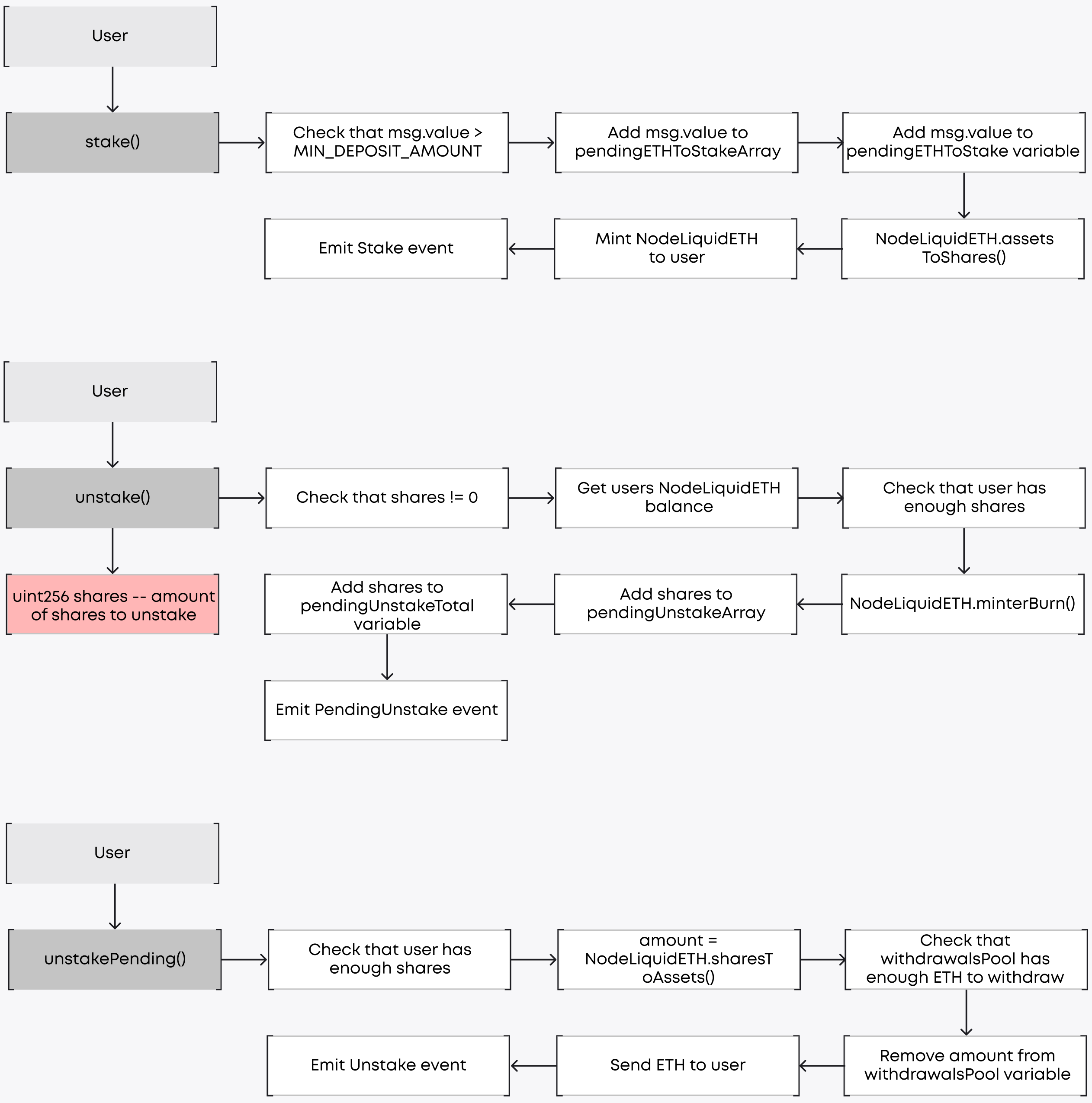
	RATING
Security	9.7
Gas usage and logic optimization	9.7
Code quality	9.6
Test coverage**	10
Total	9.7

** Contracts do not have native unit-test coverage - all tests are written by Blaize Security team in order to achieve sufficient coverage and check business-logic.

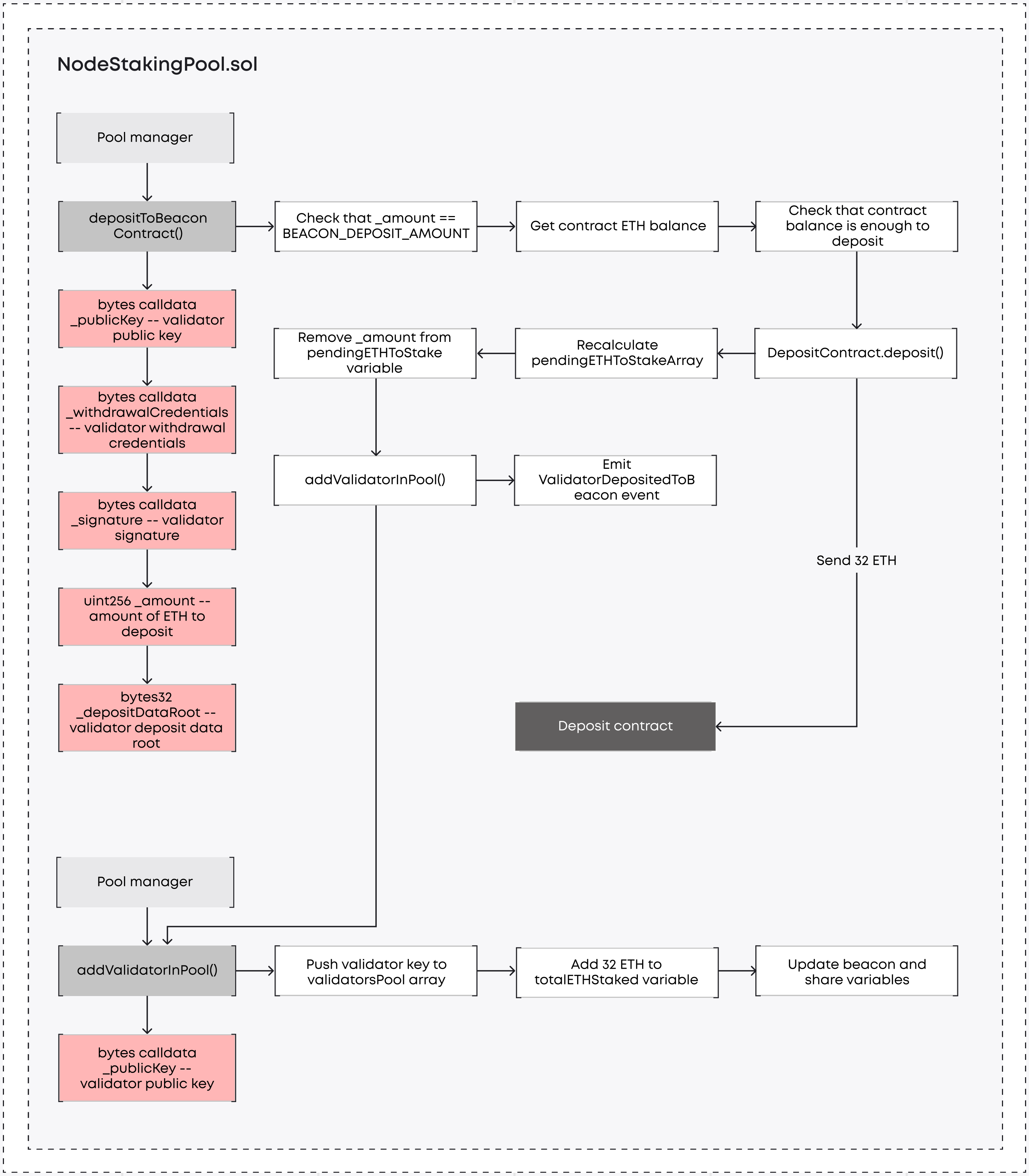
01NODE SCHEME

NodeStakingPool.sol

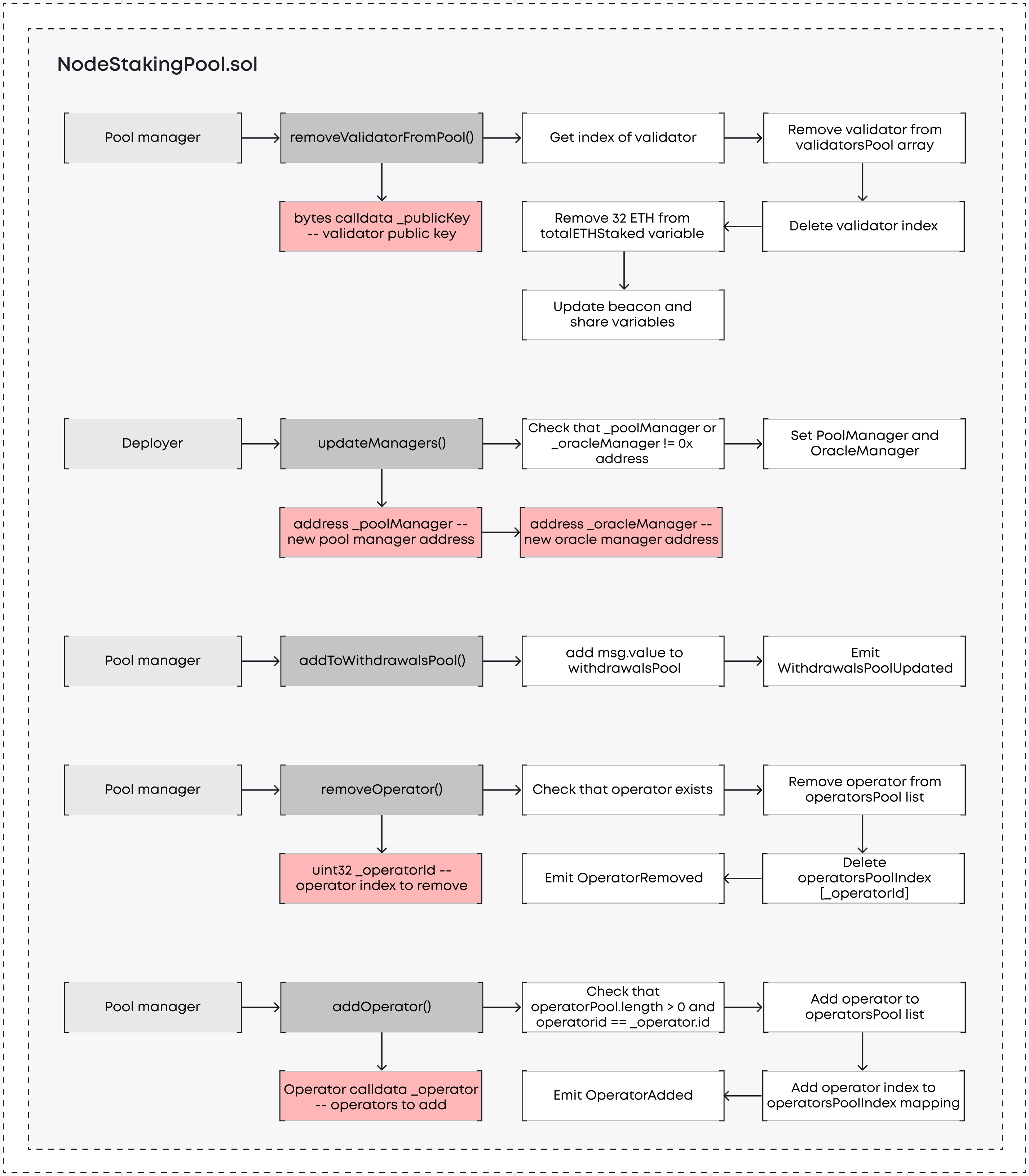
NodeStakingPool is a contract, that user could stake ETH to earn rewards.
Users get NodeLiquidETH in exchange to proof stake.
User flow: user stakes ETH with stake() -> gets NodeLiquidETH -> unstake() -> burn NodeLiquidETH -> unstakePending() -> gets ETH back.


User
stake()
Check that msg.value > MIN_DEPOSIT_AMOUNT
Add msg.value to pendingETHToStakeArray
Add msg.value to pendingETHToStake variable
NodeLiquidETH.assets ToShares()
Mint NodeLiquidETH to user
Emit Stake event
User
unstake()
uint256 shares -- amount of shares to unstake
Check that shares != 0
Get users NodeLiquidETH balance
Check that user has enough shares
NodeLiquidETH.minterBurn()
Add shares to pendingUnstakeArray
Add shares to pendingUnstakeTotal variable
Emit PendingUnstake event
User
unstakePending()
Check that user has enough shares
amount = NodeLiquidETH.sharesT oAssets()
Check that withdrawalsPool has enough ETH to withdraw
Remove amount from withdrawalsPool variable
Send ETH to user
Emit Unstake event

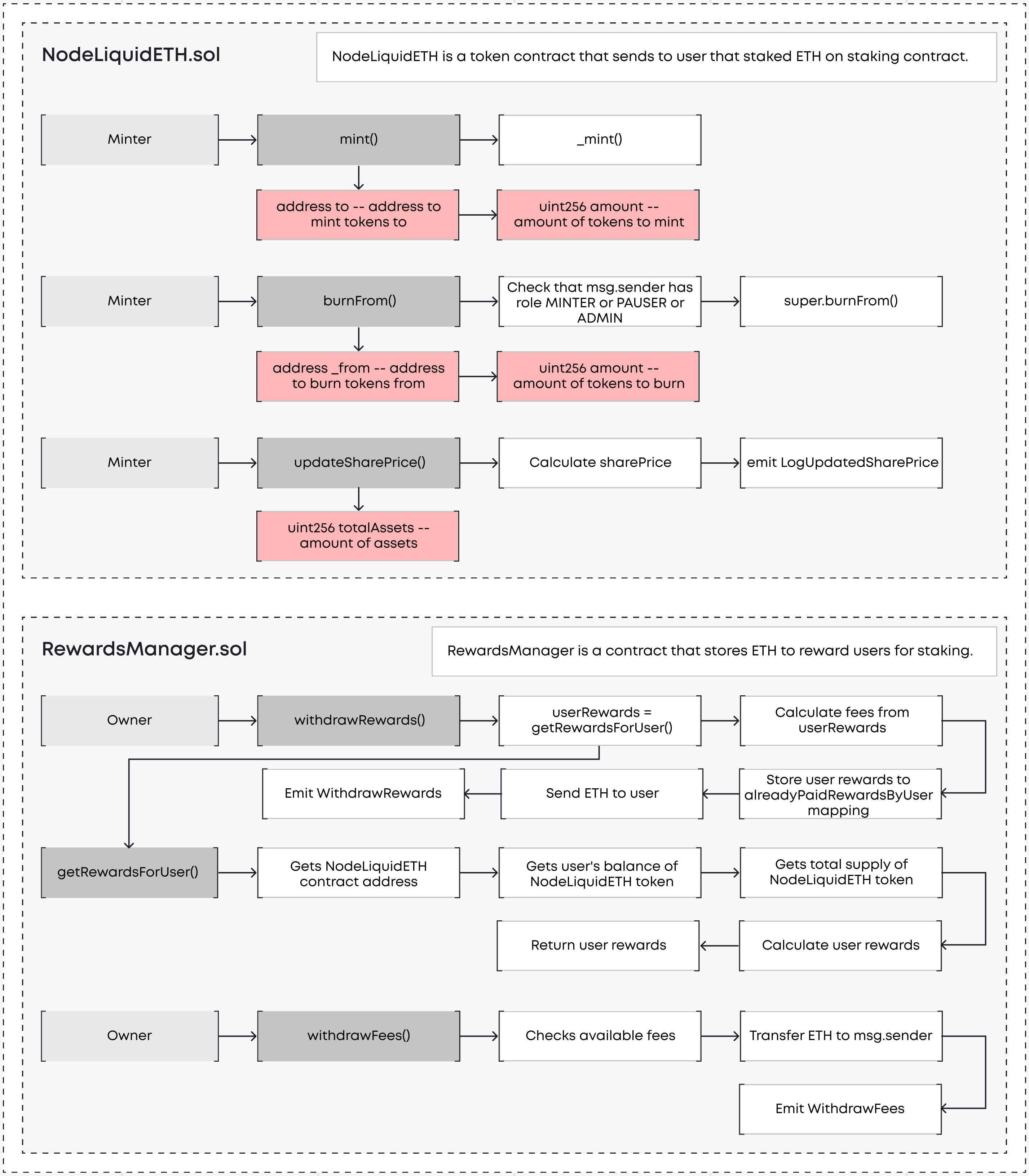
01NODE SCHEME


NodeStakingPool.sol
Pool manager
depositToBeacon Contract()
Check that _amount == BEACON_DEPOSIT_AMOUNT
Get contract ETH balance
Check that contract balance is enough to deposit
bytes calldata _publicKey -- validator public key
Remove _amount from pendingETHToStake variable
Recalculate pendingETHToStakeArray
DepositContract.deposit()
bytes calldata _withdrawalCredentials -- validator withdrawal credentials
addValidatorInPool()
Emit ValidatorDepositedToBeacon event
bytes calldata _signature -- validator signature
Send 32 ETH
uint256 _amount -- amount of ETH to deposit
bytes32 _depositDataRoot -- validator deposit data root
Deposit contract
Pool manager
addValidatorInPool()
Push validator key to validatorsPool array
Add 32 ETH to totalETHStaked variable
Update beacon and share variables
bytes calldata _publicKey -- validator public key

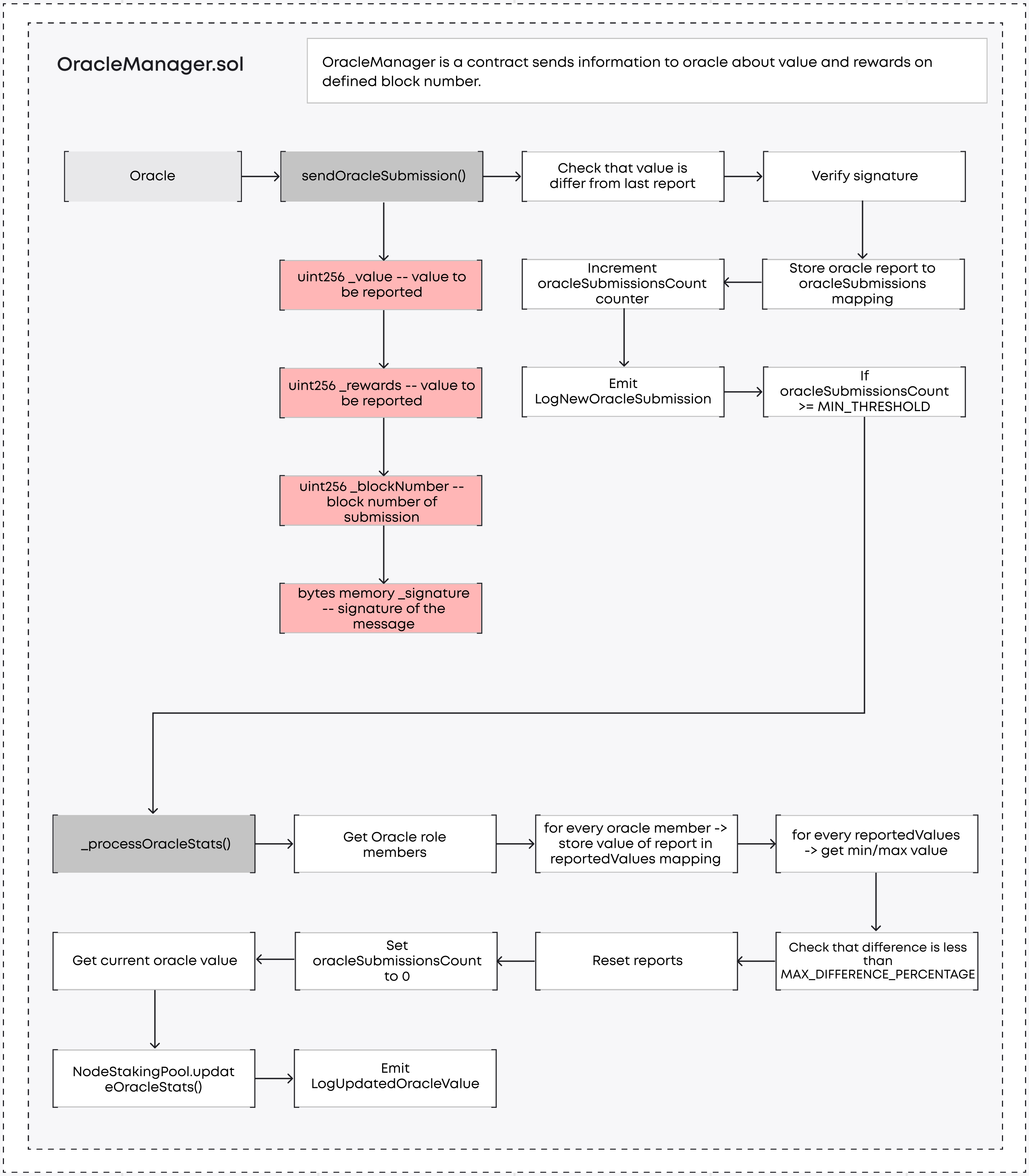
01NODE SCHEME


NodeStakingPool.sol
Pool manager
removeValidatorFromPool()
Get index of validator
Remove validator from validatorsPool array
bytes calldata _publicKey -- validator public key
Remove 32 ETH from totalETHStaked variable
Delete validator index
Update beacon and share variables
Deployer
updateManagers()
Check that _poolManager or _oracleManager != 0x address
Set PoolManager and OracleManager
address _poolManager -- new pool manager address
address _oracleManager -- new oracle manager address
Pool manager
addToWithdrawalsPool()
add msg.value to withdrawalsPool
Emit WithdrawalsPoolUpdated
Pool manager
removeOperator()
Check that operator exists
Remove operator from operatorsPool list
uint32 _operatorId -- operator index to remove
Emit OperatorRemoved
Delete operatorsPoolIndex [_operatorId]
Pool manager
addOperator()
Check that operatorPool.length > 0 and operatorid == _operator.id
Add operator to operatorsPool list
Operator calldata _operator -- operators to add
Emit OperatorAdded
Add operator index to operatorsPoolIndex mapping

01NODE SCHEME


NodeLiquidETH.sol
NodeLiquidETH is a token contract that sends to user that staked ETH on staking contract.
Minter
mint()
_mint()
address to -- address to mint tokens to
uint256 amount -- amount of tokens to mint
Minter
burnFrom()
Check that msg.sender has role MINTER or PAUSER or ADMIN
super.burnFrom()
address _from -- address to burn tokens from
uint256 amount -- amount of tokens to burn
Minter
updateSharePrice()
Calculate sharePrice
emit LogUpdatedSharePrice
uint256 totalAssets -- amount of assets
RewardsManager.sol
RewardsManager is a contract that stores ETH to reward users for staking.
Owner
withdrawRewards()
userRewards = getRewardsForUser()
Calculate fees from userRewards
Store user rewards to alreadyPaidRewardsByUser mapping
Send ETH to user
Emit WithdrawRewards
getRewardsForUser()
Gets NodeLiquidETH contract address
Gets user's balance of NodeLiquidETH token
Gets total supply of NodeLiquidETH token
Calculate user rewards
Return user rewards
Owner
withdrawFees()
Checks available fees
Transfer ETH to msg.sender
Emit WithdrawFees

01NODE SCHEME


OracleManager.sol
OracleManager is a contract sends information to oracle about value and rewards on defined block number.
Oracle
sendOracleSubmission()
Check that value is differ from last report
Verify signature
uint256 _value -- value to be reported
Increment oracleSubmissionsCount counter
Store oracle report to oracleSubmissions mapping
uint256 _rewards -- value to be reported
Emit LogNewOracleSubmission
If oracleSubmissionsCount >= MIN_THRESHOLD
uint256 _blockNumber -- block number of submission
bytes memory _signature -- signature of the message
_processOracleStats()
Get Oracle role members
for every oracle member -> store value of report in reportedValues mapping
for every reportedValues -> get min/max value
Get current oracle value
Set oracleSubmissionsCount to 0
Reset reports
Check that difference is less than MAX_DIFFERENCE_PERCENTAGE
NodeStakingPool.updateOracleStats()
Emit LogUpdatedOracleValue

PROTOCOL DESCRIPTION:

01node is a protocol that enables users to stake ETH to earn rewards/yield using the SSV network as a validator/operator provider. It consists of:

- NodeLiquidETH token contract that the user gets when staking ETH;
- the NodeStakingPool contract, which allows users to stake ETH, send ETH to Beacon contract to become a validator, provide operator management, and register SSV validator;
- the RewardManager contract that stores reward information for the user if the staked ETH on protocol;
- OracleManager contract that stores information about the protocol.

Roles and Responsibilities:

(NodeLiquidETH)

1. DEFAULT_ADMIN_ROLE
Responsibilities:
- Contract deployment
- Burn tokens

2. PAUSER_ROLE
Responsibilities:
- Pause/unpause contract
- Burn tokens

3. MINTER_ROLE
Responsibilities:
- Mint tokens
- Burn tokens
- Update Share price

Note: MINTER_ROLE and PAUSER_ROLE should be NodeStakingPool contract

(NodeStakingPool)

1. POOLDEPLOYER

Responsibilities:
- Contract deployment
- Update managers (PoolManager, OracleManager)

2. POOLMANAGER

Responsibilities:
- Pause/unpause contract
- Add ETH to withdrawal pool
- Reactivate cluster on SSVNetwork contract
- Add/remove Operator
- Deposits to Beacon contract
- Register validator on SSVNetwork contract
- Add deployed validator
- Add/remove validator in validator pool

3. ORACLEMANAGER

Responsibilities:
- Update Oracle stats

(RewardsManager)

1. OWNER

Responsibilities:
- Update owner
- Update Pool contract
- Pause/unpause contract
- Update reward fees
- Withdraw fees

2. POOLCONTRACT

Responsibilities:
- Update rewards

(OracleManager)

1. DEFAULT_ADMIN_ROLE
Responsibilities:
- Contract deployment

2. ORACLE_ROLE
Responsibilities:
- Send Oracle submission
- Process Oracle stats

3. MANAGER_ROLE (staking pool)
Responsibilities:
- Update minimum threshold
- Update maximum difference percentage
- Update staking pool
- Pause/unpause contract

Deployment:
(NodeLiquidETH)

The deployment script seems correct and follows the standard procedure for deploying smart contracts using the Hardhat deployment plugin. Here's the breakdown of the script:

- Gets deployer address and checks its address and balance
- Gets contract factory and deploys it using deployProxy function.
- Returns proxy address to console.

(NodeStakingPool)

The deployment script follows the standard procedure for deploying smart contracts using the Hardhat deployment plugin with deployment necessary contracts like RewardsManager and NodeLiquidETH contracts. Here's the breakdown of the script:

- Prepared necessary addresses to deploy staking pool (Manager address, SSVNetwork contract address, SSVToken contract address, ETHBeacon contract address, NodeLiquidETH contract although it deploys new contract in script)
- Gets deployer address and checks its address and balance
- Deploys RewardsManager using deployProxy function with deployer address as contract owner. Returns proxy address to console
- Deploys NodeLiquidETH using deployProxy function. Returns proxy address to console
- Deploys NodeStakingPool using deployProxy with deployed contracts. Deployer address sets as Pool and Oracle manager. Returns proxy address to console
- On NodeLiquidETH contract grants MINTER_ROLE to deployed NodeStakingPool contract.
- On RewardsManager updates Pool contract with deployed NodeStakingPool contract

Note: There is no deployment script for OracleManager contract. There are no separate deployment scripts for NodeStakingPool and RewardsManager contracts.

The upgrade script is only present for the NodeStakingPool contract, although implementation should be deployed before using the upgrade script.

COMPLETE ANALYSIS

CRITICAL-1 ✓ **Resolved**

Total rewards could be rewritten.

RewardsManager.sol: updateRewards().
Function updateRewards invokes in NodeStaking contract, when updating oracle stats. This will rewrite rewardsPool and totalRewards variables instead of adding value to it. As example, admin adds 2 ETH to RewardsManager -> rewardsPool and totalRewards are now 2 -> in NodeStaking contract invoked addValidatorInPool() -> _updateOracleStats with rewards as 0 -> rewardsPool and totalRewards are now 0 instead of 2, which means that ETH is stuck in contract unless change PoolContract to admin wallet and change rewards. Change logic of updating rewards to prevent such behavior.

Recommendation:

Change logic of updating rewardsPool and totalRewards variables.

Post audit.
Rewards now added instead of rewrite.

HIGH-1 ✓ **Resolved**

Obsolete eth transfer method.

NodeStakingPoolV230.sol: unstakePending(), line 592
RewardsManager.sol: withdrawRewards(), line 231
withdrawFees(), line 247
The function uses the .transfer() method for ETH transfer. The withdraw() function sends all ETH to the owner, which may be set to the multisig account. In this case, transfer() may revert since it does not forward enough gas. Therefore, the funds may be stuck on the contract. Since the .transfer() and .send() methods became obsolete after the Istanbul Ethereum update, it is recommended to use .call() for funds transfer with a mandatory check of the .call result.

Recommendation:

Use .call() for ETH transfer.

Post audit.

Changed transfer() to call().

HIGH-2 ✓ **Resolved**

beaconBalance is not updating its value.

NodeStakingPoolV230.sol: beaconBalance variable.
The beaconBalance variable is initialized as zero and remains zero because no value is added to it. The newBeaconBalance variable is a local variable, the result of which should be beaconBalance. Since beaconBalance is 0, there is a problem with removing a validator from the pool because it tries to subtract 32 from 0.

Recommendation:

Add value to beaconBalance variable when transferring ETH to beacon contract.

Post audit.

beaconBalance now saves its value.

HIGH-3 ✓ **Resolved**

An error in the algorithm for deleting operators

NodeStakingPoolV230.sol: `removeOperator()`
Since the deletion changes the location of the last current operator in the operatorsPool array of operators, but does not overwrite the location index of this operator (which must be updated in the operatorsPoolIndex map), errors occur when deleting this operator in the next transaction.

Recommendation:

Update the operator index.

```
function removeOperator(uint32 _operatorId) external
onlyManager {
  ...
  [index] = operatorsPool[operatorsPool.length - 1];
  [operatorsPool[operatorsPool.length - 1].id] = index; // <- new code
  operatorsPool.pop();
  ...
}
```

Post audit.

Added recommended code.

MEDIUM-1 ✓ **Resolved**

ETH could be stuck on contract.

RewardsManager.sol: variables totalFeesEarned, totalFeesWithdrawn
Variables totalFeesEarned and totalFeesWithdrawn not changing which makes it unable to complete withdrawFees() function.

Recommendation:

Store changes in variables.

Post audit.

Variables totalFeesEarned and totalFeesWithdrawn are now stores its value.

LOW-1	✓ Resolved

Variables set to default values in initializer.

NodeStakingPoolV230.sol: initialize(), lines 233, 241-252, 254, 257.
Global variables are initialized in the initializer with default values like 0 or the zero address. Removing the variables set from the initializer is recommended to reduce gas when deploying the contract.

Recommendation:

Remove default variable sets in the initializer.

Post audit:
Default values removed.

LOW-2	✓ Resolved

Parameters lack validation.

NodeStakingPoolV230.sol: initialize().RewardsManager.sol: initialize(), updateOwner(), updatePoolContract().
OracleManager.sol: initialize().
Parameters representing token addresses should be validated to ensure they are not zero addresses. The address parameter should be validated during deployment. In case of human error, the token could be presented as a zero address, resulting in errors before this problem is known and the contracts are upgraded.

Recommendation:

Validate functions parameters.

Post audit.
Added zero address check in given functions.

LOW-3	✓ Resolved

Contradiction in expressions.

RewardsManager.sol: withdrawRewards(), lines 226-228, withdrawFees(), lines 242-244.
In the given expressions, the variables are checked to see if they are less than zero. However, because the variables are of the uint type, the expressions will always be false. It is recommended to either remove these expressions or change the check conditions.

Recommendation:

Remove false expressions OR change variable check.

Post audit:
Expressions removed.

LOW-4	✓ Resolved

Lack of events.

NodeStakingPoolV230.sol: updateManagers(), addValidatorInPool(), removeValidatorFromPool()
RewardsManager.sol: updateOwner(), updatePoolContract(), updateRewardsFee(), updateRewards(),
Events from the functions above are not emitted. However, they can store information about important operations on the contract, so these operations can be tracked in the future.

Recommendation:

Emit events in the given functions.

Post audit.
Events are now emitted.

LOW-5	✓ Resolved

Unnecessary array length check.

NodeStakingPoolV230.sol: pendingUnstakeArray, lines 508, 520, 546, 584.
As array is initialized with value 0 in it, length cannot be 0. Because of it, there is no need to check if there are any values.

Recommendation:

Remove unnecessary checks on array length.

Post audit:
Checks removed.

LOWEST-1	✓ Resolved

Contradiction in the deployment of contracts.

NodeStakingPoolV230.sol`and OracleManager.sol`
To deploy the NodeStakingPoolV230`contract, you need to pass the address of the OracleManager`contract as an argument. However, to deploy the OracleManager`contract, you need to specify the address of the NodeStakingPoolV230`contract.

Recommendation:

Specify the address of the first contract after deploying the second contract, or vice versa.

Post audit.
Removed OracleManager address from initialization.

LOWEST-2	✓ Resolved

Complications in calculations.

NodeLiquidETH.sol: updateSharePrice(), lines 112-119
The `sharePrice`calculation uses a call to the `_decimalsOffset()` method, which returns a constant value. These are additional gas costs. Also in updateSharePrice() this function is used to represent the power of 10, which also will be always 1 because of 10 ** 0.

Recommendation:

Remove the unnecessary call to the `_decimalsOffset()`method and replace the result of this method call with a literal.

Post audit.
Removed _decimalsOffset(), added DECIMALS_OFFSET variable.

LOWEST-3	✓ Resolved

Unequal access for specified roles.

NodeLiquidETH.sol`
Only the MINTER_ROLE' role has the right to mint tokens, while the MINTER_ROLE', PAUSER_ROLE', and DEFAULT_ADMIN_ROLE' roles have the right to burn tokens.

Recommendation:

Remove unnecessary roles for burning tokens.

Post audit.
Removed roles, burn tokens now can do only MINTER_ROLE.

LOWEST-4	✓ Resolved

Burn tokens without allowance.

NodeLiquidETH.sol: burn()
The burn() function can burn tokens without the user's allowance. The token contract inherits from the ERC20BurnableUpgradeable contract, which has burn and burnFrom functions. Since the `burn()` function does not override the burn function from the inherited contract, users can burn tokens from their accounts. The same applies to the burnFrom() function, which can burn tokens if allowed. Consider overriding the functions from the inherited contract and using the burnFrom() function to burn tokens if the user has allowed it.

Recommendation:

Override burn functions and use burnFrom() instead of burn().

Post audit.
Changed burn function to burnFrom. Overridden burn and burnFrom functions.

LOWEST-5	✓ Verified

Ability to add the same operator with different id.

NodeStakingPoolV230.sol: addOperator()
Because the function only checks the id in the Operator calldata, it is possible to add the same operator with a different id.

Recommendation:

Check if validator id is already added.

Post audit.
01node has verified that they check operator entries before adding them.

LOWEST-6 ✓ **Resolved**

Functions should be marked as internal.

NodeStakingPoolV230.sol: addAlreadyDeployedValidator(), addValidatorInPool(), removeValidatorFromPool()
OracleManager.sol: processOracleStats()
The given functions are utilized in other functions. However, if they are not used in other contracts, it may be worth considering changing their visibility from public to internal and removing the unnecessary modifier for checking the role.

Recommendation:

Change functions to internal.

Post audit.
processOracleStats() function is marked as internal. Removed the addAlreadyDeployedValidator(). 01node team verified that addValidatorInPool() and removeValidatorFromPool() functions should be public to be able to add already deployed validators or that was deployed externally.

LOWEST-7 ✓ **Verified**

SSV contract addresses differ.

https://docs.ssv.network/developers/smart-contracts#goerli-testnet-v3. Verify using the latest version of SSV contracts when designing smart contracts.

Recommendation:

Verify that addresses from deploy script is latest.

Post audit.
01node team verified that contracts for SSV v4 is listed here: https://ssv-network.gitbook.io/docs-v4/

LOWEST-8	✓ Resolved

Users could invoke functions without getting any ETH.

RewardsManager.sol: withdrawRewards(), withdrawFees()
The function does not check if the value to transfer is 0. Therefore, if a user tries to invoke the function without rewards, the function will be successful but have no effect as no rewards were collected.
Add a check to ensure that the amount to transfer is not equal to zero.

Recommendation:

Add check value != 0.

Post audit.
Added check for user rewards and fees.

LOWEST-9	✓ Verified

Updating token price.

NodeLiquidETH.sol: updateSharePrice()
As the flow goes, the user stakes ETH on the NodeStakingPool contract, and liquidETH is minted to the user. However, the price is not changing. It is important to verify how the price will change, whether manually or automatically. Please provide steps on how this is done.

Recommendation:

Verify asset price change.

Post audit.
The 01node team verified that the price updates every time a new validator is added or manually by the OracleManager.

LOWEST-10	✓ Resolved

Custom errors should be used.

OracleManager.sol: lines 153-156, 163, 209-212, 307
Starting from version 0.8.4 of Solidity, it is advisable to use custom errors instead of storing error message strings in storage and using "require" statements. Custom errors are more efficient regarding gas spending and enhance code readability.

Recommendation:

Use custom errors.

Post audit.
Custom errors now used.

LOWEST-11	✓ Resolved

Double check in function.

OracleManager.sol: _processOracleStats(), lines 182-185
Since the _processOralceStats function can only be invoked when the oracleSubmissionsCount >= MIN_THRESHOLD expression is met in the sendOracleSubmission() function, it is redundant to check if oracleSubmissionsCount >= MIN_THRESHOLD again in the _processOralceStats function.

Recommendation:

Remove the unnecessary check.

Post audit.
Unnecessary check removed.

LOWEST-12 ✓ Verified

Unnecessary variable usage.

RewardsManager.sol: totalFeesEarned, totalFeesWithdrawn
The variables totalFeesEarned, and totalFeesWithdrawn are essentially the same when withdrawing fees, as they will have the same value. Consider leaving only one variable, and when invoking withdrawFees(), simply change the fee variable to 0.

Recommendation:

Eliminate one of the given variables and reset the variable to zero when withdrawing fees.

Post audit.
01node team verified that totalFeesWithdrawn is used to track how much fee the owner took out.

LOWEST-13 ✓ Resolved

Commented code.

NodeStakingPoolV230.sol: lines 329-338, 468-491, 499-514
The code is commented, which means that it is not being used. It is important to either uncomment or delete the commented code before moving to production.

Recommendation:

Uncomment OR remove commented code.

Post audit.
Commented code was removed. Additionally, SSVNework contracts integration was removed. The 01node team verified that the Pool Manager wallet will now interact with the SSV protocol.

contracts/NodeStakingPoolV220.sol
contracts/NodeLiquidETH.sol
contracts/OracleManager.sol
contracts/RewardManager.sol

✓	Re-entrancy	Pass
✓	Access Management Hierarchy	Pass
✓	Arithmetic Over/Under Flows	Pass
✓	Delegatecall Unexpected Ether	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

CODE COVERAGE AND TEST RESULTS FOR ALL FILES, PREPARED BY BLAIZE SECURITY TEAM

NodeLiquidETH

- ✓ Should be deployed (310ms)
- ✓ Contract is on paused (89ms)
- ✓ Contract is not on paused (88ms)
- ✓ Minting of tokens (94ms)
- ✓ Burning of tokens (96ms)
- ✓ Updating of share price (92ms)
- ✓ Conversion of assets to shares
- ✓ Conversion of shares to assets
- ✓ should successfully transfer funds between accounts (60ms)
- ✓ Should update decimals offset (65ms)

  # burn method
- ✓ Should burn tokens from the token owner (60ms)

  # burnFrom method
- ✓ Should burn tokens from another token owner with their approval (61ms)

  # minterBurn method
- ✓ Should burn tokens from another token owner without their approval (47ms)

NodeStakingPoolV230

- ✓ Should be deployed (622ms)
- ✓ Should receive ETH (56ms)
- ✓ Should be reactivate cluster (42ms)
- ✓ Removing of operator (58ms)
- ✓ Getting a pool of operators
- ✓ Should add of validator with correct publicKey (46ms)
- ✓ Should add of validator with incorrect publicKey (47ms)
- ✓ Getting a index of validator (66ms)
- ✓ Deposit ETH to Beacon contract (83ms)
- ✓ Staked ETH to the pool (115ms)
- ✓ Unstacked ETH from the pool (106ms)
- ✓ Unstacked pending ETH from the pool (121ms)
- ✓ Get pending stake ETH for user
- ✓ Get pending unstacked ETH for user (58ms)
- ✓ Should be updated oracle stats (64ms)

✓ Contract is on paused (55ms)
✓ Contract is not on paused (70ms)
✓ Add validator in pool by a non-manager

updatePoolManager method
✓ Should update the pool manager with the correct address
✓ Should not allow updates with zero address
✓ Should not allow updates without ònlyDeployer`access rights

updateOracleManager method
✓ Should update oracle manager with correct address
✓ Should not allow updates with zero address
✓ Should not allow updates without ònlyDeployer`access rights

addToWithdrawalsPool method
✓ Should update the withdrawals pool
✓ Should change ETH balances
✓ Should not allow updates without ònlyManager`access rights

addOperator method
✓ Should add an operator
✓ Should not add an already exists operator (40ms)
✓ Should not add without ònlyManager`access rights

removeValidatorFromPool method
✓ Should remove validator from pool (50ms)
✓ Remove validator from pool by a non-manager

inflation attack
✓ Should not update share price (51ms)

OracleManager

✓ Should be deployed (559ms)
✓ Should returns the hash of a message (39ms)
✓ Should returns the hash of a signed message
✓ Should be verified (53ms)
✓ Should be recover signer
✓ Should be split signature (45ms)
✓ Should be sending a new oracle submission (94ms)
✓ Calling of process oracle stats (159ms)

updateMinThreshold method
✓ Should update update min threshold with correct address
✓ Should not update with incorrect value
✓ Should not update without access rights (61ms)

```
      # updateMaxDifferencePercentage method
    ✓ Should update update max difference percentage
    ✓ Should not update without access rights (52ms)
      # updateStakingPool method
    ✓ Should update update staking pool with correct address
    ✓ Should not update with incorrect value (39ms)
    ✓ Should not update without access rights (47ms)
      # pause method
    ✓ Should be paused
    ✓ Should not be paused without access rights (57ms)
      # unpause method
    ✓ Should come off pause
    ✓ Should remain on pause without access rights (50ms)
  RewardsManager
    ✓ Should be deployed (545ms)
    ✓ Should receive ETH (105ms)
    ✓ Should be will update contract owner address (49ms)
    ✓ Contract is on paused
    ✓ Contract is not on paused
    ✓ Should be update of the rewards fee
    ✓ Should be withdraw rewards for user (162ms)
      # withdrawFees method
    ✓ Should withdraw fees for contract owner (138ms)
    ✓ Should throw an error NotEnoughFees`
      # updateRewards method
    ✓ Should update rewards (44ms)
    ✓ Should  not update rewards without access rights (38ms)
      # updatePoolContract method
    ✓ Should update pool contract
    ✓ Should not update pool contract with incorrect data (40ms)


      79 passing (6s)
```

TEST COVERAGE RESULTS

FILE	**% STMTS**	**% BRANCH**	**% FUNCS**
NodeStakingPoolV230.sol	100	80.95	100
NodeLiquidETH.sol	100	100	100
OracleManager.sol	100	82	100
RewardManager.sol	97.06	88.64	100
All files	**99.27**	**87.9**	**100**

DISCLAIMER

The information presented in this report is an intellectual property of the customer, including all the presented documentation, code databases, labels, titles, ways of usage, as well as the information about potential vulnerabilities and methods of their exploitation. This audit report does not give any warranties on the absolute security of the code. Blaize.Security is not responsible for how you use this product and does not constitute any investment advice.

Blaize.Security does not provide any warranty that the working product will be compatible with any software, system, protocol or service and operate without interruption. We do not claim the investigated product is able to meet your or anyone else's requirements and be fully secure, complete, accurate, and free of any errors and code inconsistency.

We are not responsible for all subsequent changes, deletions, and relocations of the code within the contracts that are the subjects of this report.

You should perceive Blaize.Security as a tool, which helps to investigate and detect the weaknesses and vulnerable parts that may accelerate the technology improvements and faster error elimination.