Moonstake

Smart Contract Security Audit

Prepared by ShellBoxes

Nov 14th, 2023 - Nov 23rd, 2023

Shellboxes.com

contact@shellboxes.com

Document Properties

Client	MoonStake
Version	1.0
Classification	Public

Scope

Repository	Commit Hash
https://gitlab.moonstake.io/moonstake1/staking/ssv-smart-contract/-/tree/release/v1.0.0-audit	e46c4720802faa22a59758177151a734f9fb8f9b

Re-Audit

Repository	Commit Hash
https://gitlab.moonstake.io/moonstake1/staking/ssv-smart-contract/-/tree/release/v1.0.0-audit-updated	a61b7bd85994de013c0e17abd6d4c50c21740e9a

Contacts

COMPANY	EMAIL
ShellBoxes	contact@shellboxes.com

Contents

1 Introduction 5
1.1 About MoonStake 5
1.2 Approach & Methodology 5
1.2.1 Risk Methodology 6

2 Findings Overview 7
2.1 Disclaimer 7
2.2 Summary 7
2.3 Key Findings 7

3 Finding Details 9
SHB.1 Risk of Denial of Service (DoS) Due to Unhandled Validator Slashing in StakingDAO 9
SHB.2 Attacker Can Prevent Proposal Approval in StakingDAO Using Sandwich Attack 11
SHB.3 Denial of Service Risk in StakingDAO Through Sandwich Attack on depositValidator Function 13
SHB.4 Inconsistency in Reward Distribution Due to Modifiable Operator Fee Rate 16
SHB.5 Potential Bypass of Minimum Staking Limitation 18
SHB.6 Potential DOS in the createUnstakeProposal function 21
SHB.7 Potential Loss of Precision in _getUserReward Function 23
SHB.8 Floating Pragma 25
SHB.9 Owner Can Renounce Ownership 26
SHB.10 Missing Address Verification 27
SHB.11 Missing Value Verification 28
SHB.12 Unprotected Exposure of API Keys 30

4 Best Practices 31
BP.1 Modularization of Dao Logic Management 31
BP.2 Remove Unused Variable _proposalEpoch 32

5 Tests 33

6 Conclusion 36

7 Scope Files 37
7.1 Audit 37
7.2 Re-Audit 37

8 Disclaimer 39

1 Introduction

MoonStake engaged ShellBoxes to conduct a security assessment on the Moonstake beginning on Nov 14th, 2023 and ending Nov 23rd, 2023. In this report, we detail our methodical approach to evaluate potential security issues associated with the implementation of smart contracts, by exposing possible semantic discrepancies between the smart contract code and design document, and by recommending additional ideas to optimize the existing code. Our findings indicate that the current version of smart contracts can still be enhanced further due to the presence of many security and performance concerns.

This document summarizes the findings of our audit.

1.1 About MoonStake

Moonstake was established to develop a staking pool protocol to satisfy the increasing demands of investors and businesses in regional and global blockchain markets. They aim to be the largest staking pool network in Asia by providing an active environment for crypto asset holders. In May 2021, Moonstake further enhanced its corporate credibility by becoming a wholly owned subsidiary of OIO Holdings Limited, a company listed on the Singapore Stock Exchange. As of October 2022, Moonstake is a Verified Provider as part of the StakingRewards Verified Provider Program (VPP).

Issuer	MoonStake
Website	`https://moonstake.io`
Type	Solidity Smart Contracts
Documentation	Milestone 1 - SSV-StakingDAO-ProductDesign
Audit Method	Whitebox

1.2 Approach & Methodology

ShellBoxes used a combination of manual and automated security testing to achieve a balance between efficiency, timeliness, practicability, and correctness within the audit's scope. While manual testing is advised for identifying problems in logic, procedure, and implementation, automated testing techniques help to expand the coverage of smart contracts and can quickly detect code that does not comply with security best practices.

1.2.1 Risk Methodology

Vulnerabilities or bugs identified by ShellBoxes are ranked using a risk assessment technique that considers both the LIKELIHOOD and IMPACT of a security incident. This framework is effective at conveying the features and consequences of technological vulnerabilities.

Its quantitative paradigm enables repeatable and precise measurement, while also revealing the underlying susceptibility characteristics that were used to calculate the Risk scores. A risk level will be assigned to each vulnerability on a scale of 5 to 1, with 5 indicating the greatest possibility or impact.

- Likelihood quantifies the probability of a certain vulnerability being discovered and exploited in the untamed.
- Impact quantifies the technical and economic costs of a successful attack.
- Severity indicates the risk's overall criticality.

Probability and impact are classified into three categories: H, M, and L, which correspond to high, medium, and low, respectively. Severity is determined by probability and impact and is categorized into four levels, namely Critical, High, Medium, and Low.

Impact				
	High	Critical	High	Medium
	Medium	High	Medium	Low
	Low	Medium	Low	Low
		High	Medium	Low
			Likelihood	

2 Findings Overview

2.1 Disclaimer

During the audit, it was noted that specific functions in the contract interact with the Ethereum 2.0 deposit contract using the IDepositContract interface. While this interface is anticipated to comply with Ethereum 2.0 specifications, it's crucial to emphasize that the audit did not conduct a dedicated evaluation of the DepositContract's security and correctness, as it falls outside the scope of this assessment. The findings are contingent on the assumption that the contract is implemented correctly, drawing reference from the Beacon Deposit Contract.

2.2 Summary

The following is a synopsis of our conclusions from our analysis of the Moonstake implementation. During the first part of our audit, we examine the smart contract source code and run the codebase via a static code analyzer. The objective here is to find known coding problems statically and then manually check (reject or confirm) issues highlighted by the tool. Additionally, we check business logics, system processes, and DeFi-related components manually to identify potential hazards and/or defects.

2.3 Key Findings

In general, these smart contracts are well-designed and constructed, but their implementation might be improved by addressing the discovered flaws, which include , 2 high-severity, 4 medium-severity, 5 low-severity, 1 informational-severity vulnerabilities.

Vulnerabilities	Severity	Status
SHB.1. Risk of Denial of Service (DoS) Due to Unhandled Validator Slashing in StakingDAO	HIGH	Fixed
SHB.2. Attacker Can Prevent Proposal Approval in StakingDAO Using Sandwich Attack	HIGH	Fixed

SHB.3. Denial of Service Risk in StakingDAO Through Sandwich Attack on depositValidator Function	MEDIUM	Fixed
SHB.4. Inconsistency in Reward Distribution Due to Modifiable Operator Fee Rate	MEDIUM	Fixed
SHB.5. Potential Bypass of Minimum Staking Limitation	MEDIUM	Fixed
SHB.6. Potential DOS in the createUnstakeProposal function	MEDIUM	Fixed
SHB.7. Potential Loss of Precision in _getUserReward Function	LOW	Fixed
SHB.8. Floating Pragma	LOW	Fixed
SHB.9. Owner Can Renounce Ownership	LOW	Fixed
SHB.10. Missing Address Verification	LOW	Fixed
SHB.11. Missing Value Verification	LOW	Fixed
SHB.12. Unprotected Exposure of API Keys	INFORMATIONAL	Fixed

3 Finding Details

SHB.1 Risk of Denial of Service (DoS) Due to Unhandled Validator Slashing in StakingDAO

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

The StakingDAO smart contract fails to adequately handle scenarios where a validator is slashed due to protocol violations in the Ethereum 2.0 staking process. Slashing is a penalty imposed on validators who act maliciously or incompetently, resulting in a significant reduction of their staked ETH. This oversight in the contract's design leads to a critical flaw in the _getTotalReward function. The _getTotalReward function currently assumes that the total amount of ETH sent to the contract will always be equal to the initially staked 32 ETH. This assumption does not account for the possibility of slashing, where the validator's balance can be significantly reduced as a penalty. In the event of a validator being slashed, the returned balance to the StakingDAO upon withdrawal will be less than the 32 ETH initially staked by the users. The _getTotalReward function, however, continues to calculate rewards based on the assumption of a full 32 ETH return. This miscalculation can lead to an underflow in the contract's logic. For instance, if the rewards at exit are 5 ETH, and the validator was slashed previously, resulting in a remaining balance of 20 ETH, the calculation would erroneously attempt to subtract the full 32 ETH (_totalAmount) from a combined balance of 25 ETH (20 ETH from the withdrawal + 5 ETH in rewards). This underflow will effectively lock users' funds in the contract, preventing them from unstaking and claiming rewards, thereby causing a Denial of Service (DoS).

Files Affected:

SHB.1.1: StakingDAO.sol

```
function _getTotalReward() internal view returns (uint256) {
  uint256 balanceToReward = address(this).balance;
  if (_poolStatus == PoolStatus.Exited) {
    balanceToReward = balanceToReward - _totalAmount;
  }
  return balanceToReward + _claimedReward;
}
```

Recommendation:

To mitigate this risk, the StakingDAO contract should incorporate logic to handle the slashing scenario effectively. This can be done by dynamically adjusting the expected return amount based on the actual balance of the contract post-withdrawal. In addition to that, the unstake function should be adjusted to use _mapAmount to determine the percentage that will be send to the user instead of sending the amount directly.

Updates

The team resolved the issue by calculating the user staked amount based the returned balance from the beacon chain instead of assuming it to be 32ETH.

SHB.1.2: StakingDAO.sol

```
function _getUserStakedAmount(address user) internal view returns (
   ↪ uint256) {
  if (_mapAmount[user] == 0) return 0; // not deposited
  if (_poolStatus != PoolStatus.Exited) return _mapAmount[user];

  uint256 totalShares = VALIDATOR_AMOUNT;
  // _balanceAtExited - _balanceAtExiting < VALIDATOR_AMOUNT
  if (_balanceAtExited < _balanceAtExiting + VALIDATOR_AMOUNT) {
    // Case: Validator is slashed
    totalShares = _balanceAtExited - _balanceAtExiting;
```

```
    }

    return (totalShares * _getUserShares(user)) / MULTIPLIER;
```

SHB.2 Attacker Can Prevent Proposal Approval in StakingDAO Using Sandwich Attack

- Severity : HIGH
- Status : Fixed
- Likelihood : 2
- Impact : 3

Description:

The StakingDAO smart contract is susceptible to a sandwich attack, where an attacker can exploit the proposal approval process to prevent the approval of legitimate proposals. This issue is rooted in the design of the approveProposal function and the lack of mechanisms to prevent manipulative transaction ordering.

Exploit Scenario:

An attacker can initiate a sandwich attack by first creating a proposal. When a legitimate approveProposal transaction is broadcasted, the attacker can front-run this transaction with their transaction to cancel their initial proposal and immediately follow (back-run) it with another transaction to create a new proposal. This sequence disrupts the approval process, as the legitimate approveProposal transaction will not find the proposal in the expected voting state or will be directed at an outdated proposal. Consequently, this can lead to a denial of service, where genuine users are unable to get their proposals approved, therefore unable to unstake their funds.

Files Affected:

SHB.2.1: StakingDAO.sol

```
function approveProposal(address user) external virtual override onlyOwner {
  require(_poolStatus != PoolStatus.Exiting && _poolStatus != PoolStatus.Exited, "StakingDAO: already exited.");
  require(_isProposalVoting(), "StakingDAO: not voting.");

  Proposal storage proposal = _proposals[_proposals.length - 1];
  require(proposal.createdBy == user, "StakingDAO: user is not proposal owner.");

  proposal.status = ProposalStatus.Approved;

  emit ProposalApproved(user, uint256(proposal.proposalType));

  if (proposal.proposalType == ProposalType.ExitValidator) {
    _poolStatus = PoolStatus.Exiting;

    emit Exiting();
  }
}
```

Recommendation:

To mitigate this issue, the StakingDAO contract can implement one of the following recommendations:

- Implement a locking mechanism during the proposal approval process, preventing other proposals from being created or canceled in the meantime.
- Enforce a time-lock or minimum interval between the creation, cancellation, and approval of proposals to avoid quick, manipulative transactions.

- Restructure the feature design by using a combination of an array and a mapping where the owner can select which proposal to accept preventing any manipulation.

Updates

The team resolved the issue by allowing the creation of new proposals after _proposalTimeout and _minProposalCreationTime.

SHB.2.2: StakingDAO.sol

```
// Check timeout
if (block.timestamp > proposal.createdAt + _proposalTimeout) {
  return true;
}
```

SHB.2.3: StakingDAO.sol

```
if (_proposals.length > 0) {
  uint256 lastTime = _proposals[_proposals.length - 1].createdAt;
  require(block.timestamp > lastTime + _minProposalCreationTime, "
      ↪ StakingDAO: cannot create proposal this time.");
}
```

SHB.3 Denial of Service Risk in StakingDAO Through Sandwich Attack on depositValidator Function

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The StakingDAO contract's depositValidator function is vulnerable to a Denial of Service (DoS) attack via a sandwich attack technique.

This function, crucial for creating a new validator, is intended to be called when the contract's balance reaches 32 ETH. However, an exploitable design flaw allows an attacker to disrupt the validator creation process.

Exploit Scenario:

An attacker can execute a sandwich attack as follows:

1. The attacker stakes enough ETH to bring the contract's balance to the threshold of 32 ETH.
2. They monitor the mempool for the depositValidator transaction.
3. Upon detecting the transaction, the attacker front-runs it with a withdrawal transaction from their stake (unstakePending), intentionally dropping the contract's balance below 32 ETH. This causes the depositValidator transaction to fail due to the balance check (require(address(this).balance >= VALIDATOR_AMOUNT, "StakingDAO: 32 ETH required.")).
4. The attacker then back-runs the failed depositValidator transaction with another transaction to redeposit their withdrawn amount.

This sequence can be repeated to continually disrupt the validator creation process, leading to a Denial of Service where new validators cannot be formed.

Files Affected:

SHB.3.1: StakingDAO.sol

```
function depositValidator(bytes calldata signature, bytes32
    ↪ deposit_data_root) external payable virtual override
    ↪ nonReentrant onlyOwner {
  require(!_ethDeposited, "StakingDAO: Already deposited to Ethereum
      ↪ deposit contract.");
  require(_pubkey.length > 0, "StakingDAO: pubkey is not set.");
  require(address(this).balance >= VALIDATOR_AMOUNT, "StakingDAO: 32
      ↪ ETH required.");
```

SHB.3.2: StakingDAO.sol

```
function stake(uint64[] calldata operatorIds) external payable
    ↪ override nonReentrant {
  _stake(_msgSender(), payable(_msgSender()), operatorIds);
}
```

SHB.3.3: StakingDAO.sol

```
function unstakePending(uint256 amount) external override nonReentrant
    ↪  {
  _unstakePending(payable(_msgSender()), amount);
}
```

Recommendation:

To mitigate this vulnerability, consider introducing a locking mechanism that prevents withdrawals when the contract's balance reaches the 32 ETH threshold.

Updates

The team resolved the issue by enforcing a delay after reaching the 32TH threshold on the unstakePending.

SHB.3.4: StakingDAO.sol

```
    if (_totalAmount >= VALIDATOR_AMOUNT) {
      _lastMeetThresholdTime = block.timestamp;
```

SHB.3.5: StakingDAO.sol

```
function _unstakePending(address payable stakeholder, uint256 amount)
    ↪ internal virtual {
  require(_poolStatus == PoolStatus.Pending, "StakingDAO: unstakeable
      ↪ .");
  require(stakeholder != address(0), "StakingDAO: invalid stakeholder
      ↪ .");
  require(amount > 0, "StakingDAO: unstake amount under threshold.");
```

```
require(_mapAmount[stakeholder] >= amount, "StakingDAO: over balance.");
require(_lastMeetThresholdTime == 0  block.timestamp >= _lastMeetThresholdTime + _minUnstakeTime, "StakingDAO: unstakeable.");
```

SHB.4 Inconsistency in Reward Distribution Due to Modifiable Operator Fee Rate

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 1
- Impact : 3

Description:

The function _getUserReward in the StakingDAO's smart contract calculates the reward for a user based on the total accumulated reward and the user's share in the staking pool. However, the calculation is susceptible to inconsistencies due to the modifiable nature of the _operatorFeeRate. This variable determines the operation fee deducted from the total rewards before distributing them to the stakers.

Exploit Scenario:

Suppose the _operatorFeeRate is initially set to 10%. A user, Alice, stakes early and later claims her reward when the fee rate is still 10%. Her reward is calculated based on 90% of the total pool rewards (after deducting 10% fee). However, if the _operatorFeeRate is increased to 15% before another user, Bob, claims his reward, Bob's reward will be calculated based on 85% of the total pool rewards. This discrepancy leads to an uneven and potentially unfair distribution of rewards among stakers, depending on the timing of their claims.

Files Affected:

SHB.4.1: StakingDAO.sol

```
    // Calculate operation fee
    uint256 totalOperationFee;
    if (_operatorFeeRate > 0) {
      totalOperationFee = (totalReward * _operatorFeeRate) / A_HUNDRED_PERCENT;
    }

    uint256 totalNetReward = totalReward - totalOperationFee;
```

SHB.4.2: StakingDAO.sol

```
  function setOperatorFeeRate(uint256 rate) external override onlyOwner {
    _setOperatorFeeRate(rate);
  }
```

SHB.4.3: StakingDAO.sol

```
  function _setOperatorFeeRate(uint256 rate) internal virtual {
    require(_operatorFeeRate <= A_HUNDRED_PERCENT, "StakingDAO: over rate.");

    uint256 oldRate = _operatorFeeRate;
    _operatorFeeRate = rate;

    emit OperatorFeeRateChanged(oldRate, _operatorFeeRate);
  }
```

Recommendation:

Consider locking the _operatorFeeRate to one value once the pool status moves to Active.

Updates

The team resolved the issue by preventing _operatorFeeRate modifications after the pool status is Activated.

SHB.4.4: StakingDAO.sol

```
function _setOperatorFeeRate(uint256 rate) internal virtual {
  require(_poolStatus < PoolStatus.Actived, "StakingDAO: rate cannot
    ↪ changed once actived");
  require(_operatorFeeRate <= A_HUNDRED_PERCENT, "StakingDAO: over
    ↪ rate.");

  uint256 oldRate = _operatorFeeRate;
  _operatorFeeRate = rate;

  emit OperatorFeeRateChanged(oldRate, _operatorFeeRate);
}
```

SHB.5 Potential Bypass of Minimum Staking Limitation

- Severity: MEDIUM
- Status: Fixed
- Likelihood: 3
- Impact: 1

Description:

The StakingDAO smart contract enforces a minimum staking requirement _minStakeAmountRequired for participants. However, this constraint can be circumvented by staking an amount above the minimum threshold and subsequently partially unstaking, leaving a balance below the required minimum. This bypass undermines the integrity of the minimum staking rules set by the protocol.

Exploit Scenario:

A user, Charlie, initially stakes an amount higher than _minStakeAmountRequired, meeting the protocol's requirement. Afterward, Charlie exploits the loophole by calling the unstakePending function to withdraw a portion of his stake, leaving only a small, non-compliant amount (”dust”) in the staking contract. This action effectively bypasses the minimum staking requirement, allowing Charlie to maintain a stake in the DAO with an amount less than _minStakeAmountRequired.

Files Affected:

SHB.5.1: StakingDAO.sol

```
function _stake(address stakeholder, address payable recipient, uint64
    ↪ [] calldata operatorIds) internal virtual {
  uint256 depositedAmount = msg.value;

  require(_isStakeable(), "StakingDAO: not stakeable.");

  uint256 minThreshold = _minStakeAmountRequired;
  uint256 maxThreshold = _getStakeableAmount();

  if (minThreshold > maxThreshold) {
    minThreshold = maxThreshold;
  }

  require(depositedAmount >= minThreshold, "StakingDAO: stake amount
      ↪ under threshold.");
```

SHB.5.2: StakingDAO.sol

```
function unstakePending(uint256 amount) external override nonReentrant
    ↪ {
  _unstakePending(payable(_msgSender()), amount);
}
```

SHB.5.3: StakingDAO.sol

```
function _unstakePending(address payable stakeholder, uint256 amount) internal virtual {
  require(_poolStatus == PoolStatus.Pending, "StakingDAO: unstakeable.");
  require(amount > 0, "StakingDAO: unstake amount under threshold.");
  require(_mapAmount[stakeholder] >= amount, "StakingDAO: over balance.");

  uint256 finalAmount = amount;
  if (finalAmount > address(this).balance) {
    finalAmount = address(this).balance;
  }

  _totalAmount -= finalAmount;
  _mapAmount[stakeholder] -= finalAmount;

  AddressUpgradeable.sendValue(stakeholder, finalAmount);

  emit PendingUnstaked(stakeholder, finalAmount, _mapAmount[stakeholder]);
}
```

Recommendation:

Consider adding a restriction in the unstakePending function that requires the user's stake to be zero or an amount that's higher then _minStakeAmountRequired by the end of the function.

Updates

The team resolved the issue by requiring the remaining staked amount after unstaking to be zero or higher then the _minStakeAmountRequired.

SHB.5.4: StakingDAO.sol

```
uint256 remaining = _mapAmount[stakeholder] - finalAmount;
```

```solidity
    require(remaining == 0  remaining >= _minStakeAmountRequired, "StakingDAO: stake amount under threshold.");

    _totalAmount -= finalAmount;
    _mapAmount[stakeholder] = remaining;
```

SHB.6 Potential DOS in the createUnstakeProposal function

- Severity : MEDIUM
- Status : Fixed
- Likelihood : 2
- Impact : 2

Description:

The createUnstakeProposal function imposes a condition that either the _proposals array should be empty or the last proposal should not be in the Pending state to allow the creation of a new proposal. This condition introduces a potential Denial of Service (DoS) vulnerability, as it hinders the ability to create new proposals until the approval or rejection of the last proposal by the owner.

Files Affected:

SHB.6.1: StakingDAO.sol

```solidity
  function _isProposalVoting() internal view returns (bool) {
    if (_proposals.length == 0) return false;

    return _proposals[_proposals.length - 1].status == ProposalStatus.Pending;
  }

  function _createUnstakeProposal(address stakeholder) internal {
    require(_poolStatus != PoolStatus.Exiting && _poolStatus != PoolStatus.Exited, "StakingDAO: already exited.");
```

```
    require(!_isProposalVoting(), "StakingDAO: voting.");
    require(_mapAmount[stakeholder] > 0, "StakingDAO: no shares.");

    Proposal memory proposal = Proposal({
      proposalType: ProposalType.ExitValidator,
      amount: _mapAmount[stakeholder],
      blockNumber: block.number,
      status: ProposalStatus.Pending,
      createdAt: block.timestamp,
      createdBy: stakeholder
    });
    _proposals.push(proposal);
```

Recommendation:

Consider implementing a timeout mechanism for proposals in the createUnstakeProposal function. If a proposal remains unapproved or rejected for an extended period, introduce a timeout feature that allows the creation of a new proposal, ensuring the contract remains responsive and does not become susceptible to Denial of Service (DoS) scenarios due to delayed owner actions on the last proposal.

Updates

The team resolved the issue by allowing the creation of new proposals after _proposalTimeout and _minProposalCreationTime.

SHB.6.2: StakingDAO.sol

```
    // Check timeout
    if (block.timestamp > proposal.createdAt + _proposalTimeout) {
      return true;
    }
```

SHB.6.3: StakingDAO.sol

```
    if (_proposals.length > 0) {
      uint256 lastTime = _proposals[_proposals.length - 1].createdAt;
```

```
        require(block.timestamp > lastTime + _minProposalCreationTime, "
            ↪ StakingDAO: cannot create proposal this time.");
    }
```

SHB.7 Potential Loss of Precision in _getUserReward Function

- Severity: LOW
- Status: Fixed
- Likelihood: 1
- Impact: 1

Description:

The _getUserReward function in the StakingDAO contract introduces a potential loss of precision during the calculation of totalUserReward. Specifically, the _getUserShares function computes a value by multiplying_mapAmount[user] by MULTIPLIER and then dividing it by VALIDATOR_AMOUNT. This operation might result in a loss of the 5 least significant bits, equivalent to 32 wei which could have been preserved if the _mapAmount[user] * MULTIPLIER was multiplied by the totalNetReward first before dividing by the VALIDATOR_AMOUNT.

Files Affected:

SHB.7.1: StakingDAO.sol

```
  function _getUserShares(address user) internal view returns (uint256)
      ↪ {
    return (_mapAmount[user] * MULTIPLIER) / VALIDATOR_AMOUNT;
  }
```

SHB.7.2: StakingDAO.sol

```
  function _getUserReward(address user) internal view returns (uint256)
      ↪ {
```

```
    if (_poolStatus < PoolStatus.Actived) return 0;
    if (_mapAmount[user] == 0) return 0; // not deposited

    uint256 totalReward = _getTotalReward();

    // Calculate operation fee
    uint256 totalOperationFee;
    if (_operatorFeeRate > 0) {
      totalOperationFee = (totalReward * _operatorFeeRate) /
          ↪ A_HUNDRED_PERCENT;
    }

    uint256 totalNetReward = totalReward - totalOperationFee;
    uint256 totalUserReward = (totalNetReward * _getUserShares(user)) /
        ↪ MULTIPLIER;
    return totalUserReward - _mapUserClaimedReward[user];
  }
```

Recommendation:

Consider using a higher multiplier. This adjustment can help prevent precision loss and ensure accurate reward calculations.

Updates

The team resolved the issue by using a higher multiplier 1e36.

SHB.7.3: StakingDAO.sol

```
  uint256 public constant MULTIPLIER = 1e36;
```

SHB.8 Floating Pragma

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

All the contracts use a floating Solidity pragma of 0.8.9, indicating that they can be compiled with any compiler version from 0.8.9 (inclusive) up to, but not including, version 0.9.0.This flexibility could potentially introduce unexpected behavior if the contracts are compiled with a newer compiler version that includes breaking changes.

Files Affected:

SHB.8.1: StakingDAO.sol

```
pragma solidity ^0.8.9;
```

Recommendation:

It is generally recommended to lock the pragma statement to a specific Solidity compiler version to ensure consistent behavior across different compiler versions. To achieve this, consider removing the caret (^) from the pragma statement and specifying a fixed version, such as pragma solidity 0.8.9;.

Updates

The team resolved the issue by locking the solidity version to 0.8.18.

SHB.9 Owner Can Renounce Ownership

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

The StakingManager and StakingDAO contracts both inherit from the OwnableUpgradeable contract, allowing the owner to renounce ownership. Renouncing ownership results in the contract being left without an owner, effectively disabling any functionality exclusively available to the owner.

Files Affected:

SHB.9.1: StakingManager.sol

```
contract StakingManager is OwnableUpgradeable,
    ↪ ReentrancyGuardUpgradeable, IStakingManager {
```

SHB.9.2: StakingDAO.sol

```
contract StakingDAO is OwnableUpgradeable, ReentrancyGuardUpgradeable,
    ↪ IStakingDAO {
```

Recommendation:

It is recommended to prevent the owner from invoking the renounceOwnership function and to disable its functionality by overriding it.

Updates

The team resolved the issue by disabling the renounceOwnership functionality.

SHB.9.3: StakingDAO.sol

```
    function renounceOwnership() public view override onlyOwner {
```

```
    revert("StakingDAO: renounceOwnership is disabled");
  }
```

SHB.9.4: StakingManager.sol

```
  function renounceOwnership() public view override onlyOwner {
    revert("StakingDAO: renounceOwnership is disabled");
  }
```

SHB.10 Missing Address Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

In the _setDefaultOperatorFeeReceiver function of the StakingManager contract, the addr address is not properly verified to be different from address(0). This lack of verification could potentially lead to interactions with zero addresses. Similarly, the same issue is identified in the operatorFeeReceiver address within the __StakingDAO_init_unchained function of the StakingDAO contract.

Files Affected:

SHB.10.1: StakingDAO.sol

```
    _operatorFeeReceiver = operatorFeeReceiver;
```

SHB.10.2: StakingManager.sol

```
  function _setDefaultOperatorFeeReceiver(address addr) internal virtual
    ↪ {
    _defaultOperatorFeeReceiver = addr;
  }
```

Recommendation:

Consider implementing a verification check using the onlyValidAddress modifier to ensure that the provided addresses are different from address(0) before assigning them to the contract variables (_defaultOperatorFeeReceiver and _operatorFeeReceiver).

Updates

The team resolved the issue by implementing zero address checks on the _defaultOperatorFeeReceiver and _operatorFeeReceiver variables.

SHB.10.3: StakingDAO.sol

```
  require(operatorFeeReceiver != address(0), "StakingDAO: invalid
    ↪ operatorFeeReceiver.");
```

SHB.10.4: StakingManager.sol

```
function _setDefaultOperatorFeeReceiver(address addr) internal virtual
    ↪ {
  require(addr != address(0), "StakingManager: invalid address.");
  _defaultOperatorFeeReceiver = addr;
}
```

SHB.11 Missing Value Verification

- Severity : LOW
- Status : Fixed
- Likelihood : 1
- Impact : 1

Description:

Certain functions, particularly the __StakingDAO_init_unchained and __StakingManager_init_unchained functions in both the StakingDao and StakingManager contracts, lack value safety checks on their arguments.

Specifically, the contracts must ensure that minStakeAmountRequired is greater than 0, and operatorFeeRate and defaultOperatorFeeRate are less than or equal to A_HUNDRED_PERCENT.

Files Affected:

SHB.11.1: StakingDAO.sol

```
    _minStakeAmountRequired = minStakeAmountRequired;
    _operatorFeeRate = operatorFeeRate;
```

SHB.11.2: StakingManager.sol

```
    _defaultOperatorFeeRate = defaultOperatorFeeRate;
```

Recommendation:

We recommend that you verify the values provided in the arguments. The issue can be addressed by utilizing a require statement.

Updates

The team resolved the issue by implementing value checks on the arguments.

SHB.11.3: StakingManager.sol

```
  function _setDefaultOperatorFeeRate(uint256 rate) internal virtual {
    require(rate <= A_HUNDRED_PERCENT, "StakingManager: over rate");
    _defaultOperatorFeeRate = rate;
  }
```

SHB.11.4: StakingDAO.sol

```
    require(options[0] > 0 && options[2] > 0 && options[3] > 0 &&
        ↪ options[4] > 0, "StakingDAO: invalid options.");

    _poolStatus = PoolStatus.Pending;
    _stakingManager = stakingManagerAddr;
```

```

    _operatorFeeReceiver = operatorFeeReceiver;

    _minStakeAmountRequired = options[0];
    _operatorFeeRate = options[1];
```

SHB.12 Unprotected Exposure of API Keys

- Severity : INFORMATIONAL
- Status : Fixed
- Likelihood : 1
- Impact : 0

Description:

The .env.example file contains production API keys, including those for Infura and Etherscan, which are typically meant for development and testing purposes. This could lead to unintentional exposure of sensitive information.

Recommendation:

To address this concern, it is strongly advised to promptly remove the production API keys from the env.example file. This file should exclusively contain non-sensitive, placeholder values suitable for development and testing purposes.

Updates

The team resolved the issue by removing the API keys from the .env.example.

4 Best Practices

BP.1 Modularization of Dao Logic Management

Description:

The changeDaoLogicStatus function currently performs three distinct actions: deactivating Dao logics, activating other Dao logics, and setting a new active Dao logic. To enhance clarity, maintainability, and readability, consider splitting this function into three separate functions, each responsible for one specific feature. This modular approach not only improves code organization but also facilitates future updates and modifications.

Files Affected:

BP.1.1: StakingManager.sol

```
function changeDaoLogicStatus(address[] calldata disableList, address[] calldata enableList, address activeDaoLogic) external override onlyOwner {
  uint i;
  for (i = 0; i < disableList.length; i++) {
    _ensureDaoLogicExists(disableList[i]);
    _mapDaoLogic[disableList[i]] = DaoStatus.Inactive;
  }
  for (i = 0; i < enableList.length; i++) {
    _ensureDaoLogicExists(enableList[i]);
    _mapDaoLogic[enableList[i]] = DaoStatus.Active;
  }
  _ensureDaoLogicExists(activeDaoLogic);
  _mapDaoLogic[activeDaoLogic] = DaoStatus.Active;
  _activeDaoLogic = activeDaoLogic;
  emit DaoLogicStatusChanged(disableList, enableList, activeDaoLogic);
}
```

Status - Acknowledged

BP.2 Remove Unused Variable _proposalEpoch

Description:

The StakingDAO contract includes a private variable, _proposalEpoch, which is declared but remains unused throughout the contract. This unused variable introduces unnecessary storage consumption and does not contribute to the contract's intended functionality. To enhance code readability, reduce storage costs, and adhere to best practices, it is recommended to remove this variable from the contract.

Files Affected:

BP.2.1: StakingDAO.sol

```
uint256 private _proposalEpoch; // Unused, remove later
```

Status - Fixed

5 Tests

Results:

→ Attacker

✓ should prevent selfdestruct attack on StakingDAO contract (87ms)

→ Setup network

✓ should successfully setup network (450ms)

→ StakingDAO contract - Using EOA as Owner

✓ should successfully getInfo (88ms)

✓ should successfully getStakeholderInfo

✓ should successfully receive ETH directly (43ms)

✓ should successfully return rewards and shares rate (82ms)

✓ should successfully setMinStakeAmount (255ms)

✓ should successfully stake (307ms)

✓ should successfully unstakePending (1007ms)

✓ should successfully depositShares and depositValidator (1) (2804ms)

✓ should successfully generate oeprator keys (4818ms)

✓ should successfully update shares (35350ms)

→ StakingManager contract - Using EOA as Owner

✓ should successfully getInfo (43ms)

✓ should successfully setInitialAmount (107ms)

✓ should successfully addDaoLogic (259ms)

✓ should successfully changeDaoLogicStatus (2451ms)

✓ should successfully createStakingDAO (649ms)

→ StakingProxyAdmin contract - Using EOA as Owner

✓ should successfully upgrade to TimelockMock (155ms)

→ StakingProxyAdmin contract

✓ should successfully upgrade to StakingDAOMock (573ms)

✓ should NOT successfully upgrade to Invalid StakingDAOMock (434ms)

→ StakingProxyAdmin contract - Using Timelock as Owner

✓ should successfully upgrade to TimelockMock (201ms)

→ StakingSecurityScheme contract

✓ should successfully test all functions (516ms)

→ StakingTimelock and StakingSecurityScheme contracts

✓ should NOT successfully setInitialAmount by StakingTimelock (StakingSecurityScheme + StakingTimelock + StakingManager) (313ms)

✓ should successfully getInfo (57ms)

✓ should NOT successfully create schedule /scheduleBatch due to the SecurityScheme conditions do not match. (290ms)

✓ should NOT successfully grant/revoke role (186ms)

✓ should successfully setInitialAmount for StakingManager and setMinStakeAmount for StakingDAO (331ms)

✓ should successfully depositShares and depositValidator (1562ms)

28 passing (1m)

Coverage:

The code coverage results were obtained by running npx hardhat coverage in the moonstake project [commit : a5c3205a0ed2c64f1464fcb4051d711d978edbbf]. We found the following results :

File	% Stmts	% Branch	% Funcs	% Lines
contracts/IStakingDAO.sol	100	100	100	100
contracts/IStakingManager.sol	100	100	100	100
contracts/IStakingSecurityScheme.sol	100	100	100	100
contracts/libs/Libs.sol	83.33	75	100	83.33
contracts/proxy/StakingProxyAdmin.sol	75	100	80	77.78
contracts/proxy/StakingTransparentProxy.sol	0	100	0	0
contracts/StakingDAO.sol	45.18	31.33	50	50.44
contracts/StakingManager.sol	88.52	55	75	87.78
contracts/StakingSecurityScheme.sol	98.28	65	94.12	98.53
contracts/StakingSecuritySchemeConstant.sol	100	100	100	100
contracts/StakingTimelock.sol	100	50	100	100

6 Conclusion

In this audit, we examined the design and implementation of Moonstake contracts and discovered several issues of varying severity. MoonStake team addressed all the issues raised in the initial report and implemented the necessary fixes.

However Shellboxes' auditors advised MoonStake Team to maintain a high level of vigilance and participate in bounty programs in order to avoid any future complications.

7 Scope Files

7.1 Audit

Files	MD5 Hash
contracts/StakingDAO.sol	b58aa53b6c851cc8a817c2e9d1e684eb
contracts/StakingManager.sol	2e28fa23eed003d8967955173ee09e96
contracts/StakingSecurityScheme.sol	5b6b904b24758bcedbd79e8f0c2d9b41
contracts/StakingSecuritySchemeConstant.sol	1c4c23a096a1b34d34dff81c7faf9ee8
contracts/StakingTimelock.sol	6ddf241bfb73e1b9eef79a18abbe509d
contracts/utils/Libs.sol	3f3e0319e016987a7f30d26d488046e5
contracts/proxy/StakingImplChecker.sol	fcc3e330220dcc99189e157cc2d21487
contracts/proxy/StakingProxyAdmin.sol	9dd49d33735294671de95dc0d7b16e85
contracts/proxy/StakingTransparentProxy.sol	b3ea89da439c4fb016b97bf9ee5c3aa4
contracts/interfaces/IStakingDAO.sol	0e108a4353c5f924faa3ae43cb2a65c5
contracts/interfaces/IStakingManager.sol	8aaa3b9fac41368df870348697f68058
contracts/interfaces/IStakingSecurityScheme.sol	a40f2f5430103431e16fe6b330f42a6d

7.2 Re-Audit

Files	MD5 Hash
contracts/StakingDAO.sol	457535b5b235e96cf7ac1cfc750f4006

contracts/StakingManager.sol	552ca2dc491bf06dfca9072bf9d0e344
contracts/StakingSecurityScheme.sol	107e52953e797fec3511a2b08935ef5a
contracts/StakingSecuritySchemeConstant.sol	ed99dfc467333964874a5f88cfc1c7c1
contracts/StakingTimelock.sol	e8fe3a70209096a37fa345310c924285
contracts/utils/Libs.sol	fbfcb9c5ec94587def98c80c82cf5ec6
contracts/proxy/StakingImplChecker.sol	231fb4ae38e4c0d733b5e5d8c806b6a0
contracts/proxy/StakingProxyAdmin.sol	3aa35139cdb4941262a62e52d472e8b3
contracts/proxy/StakingTransparentProxy.sol	d970bd233eee7c493018d517b6bf25b3
contracts/interfaces/IStakingDAO.sol	764f1ed9ab77c48f67eb392bc2dc14c8
contracts/interfaces/IStakingManager.sol	2aeffe637e13f617f1e3e8d7ac47af08
contracts/interfaces/IStakingSecurityScheme.sol	e8f6acf1122aad27f91e7d7a650cf80a

8 Disclaimer

Shellboxes reports should not be construed as ”endorsements” or ”disapprovals” of particular teams or projects. These reports do not reflect the economics or value of any ”product” or ”asset” produced by any team or project that engages Shellboxes to do a security evaluation, nor should they be regarded as such. Shellboxes Reports do not provide any warranty or guarantee regarding the absolute bug-free nature of the examined technology, nor do they provide any indication of the technology's proprietors, business model, business or legal compliance. Shellboxes Reports should not be used in any way to decide whether to invest in or take part in a certain project. These reports don't offer any kind of investing advice and shouldn't be used that way. Shellboxes Reports are the result of a thorough auditing process designed to assist our clients in improving the quality of their code while lowering the significant risk posed by blockchain technology. According to Shellboxes, each business and person is in charge of their own due diligence and ongoing security. Shellboxes does not guarantee the security or functionality of the technology we agree to research; instead, our purpose is to assist in limiting the attack vectors and the high degree of variation associated with using new and evolving technologies.

SHELLBOXES

For a Contract Audit, contact us at contact@shellboxes.com