Security Audit Report

Prepared for: *Stakewise*
Online report:
[stakewise-eth2-staking-implementation]
Date: *10/25/2021*

ETH Staking V Implementation Security Audit

We were tasked with performing an audit on the Stakewise V2 codebase, a liquid non-custodial ETH2 staking implementation which is deployed and actively managed under a proxy pattern. The particular areas of focus for the audit were the fixes applied to the recently discovered ETH2-specific vulnerability that affected the Rocket Pool, Lido and Stakewise's own code as well as the overall security of the liquid staking scheme.

To achieve a satisfactory level of coverage, we utilized a zero-assumption approach when auditing the codebase to assess whether appropriate access controls and input sanitizations were applied in the various workflows supported by the system. For documentation, we relied on the explicitness of the code and in-line documentation present at each contract's respective `interface` as the documentation of the Stakewise website references the V1 implementation.

Over the course of the audit, we were able to pinpoint certain misbehaviours as well as fringe cases that appeared unaccounted for and should be remediated to ensure the system achieves a higher level of security. In addition to the edge cases we included as findings, we assessed the logic paths of other commonly-vulnerable malicious inputs such as transfers-to-self which were permitted by the system but did not result in any misbehaviour as the system performs each account's action atomically.

In order to properly validate the fix applied by Stakewise for the ETH2-specific vulnerability, we resorted to the documentation of ETH2, potential solutions that were included in DAO discussions of other projects, and our own understanding of the issue as the implications of this vulnerability are significant. In simple terms, the vulnerability arises in the way ETH2 locks the withdrawal credentials to a particular set that is specified during the first minimum valid deposit equivalent to 1 ETH, thus allowing a race condition to arise whereby a user specifies a different withdrawal credential than the one that the Stakewise protocol is meant to specify and causing staked funds that would have been redirected to the protocol to instead be siphoned out by the attacker, inclusive of the stake necessary to become a node operator.

The fix Stakewise has decided to apply is to rely on the oracle members of the `Oracles` contract to manage the lifetime cycle of a node operator instead. This puts the onus of an honest operator's submission to the protocol oracles as they are responsible for validating both the `depositData` of an operator as well as the accuracy of the withdrawal credential they provide. To deal with potentially rogue operators, a slashing mechanism was implemented that donates the 1 ETH required for the initialization of an operator's status to the pool of the protocol itself.

A severe misbehaviour we identified in the contracts was with regards to the staleness of the reward per token within the `RewardEthToken` implementation that could permit a zero-balance user to claim a huge delta in the reward-per-token rate should they remain with a zero-balance over a long period. Vulnerabilities aside, the codebase overall is of a very high standard and all the best security principles have been followed when it comes to the upgrade-ability component of the system. Namely, of the contracts in scope the following three contracts are meant to be newly deployed (`Oracles`, `Roles`, `PoolValidators`) whereas the rest are meant to replace the existing deployed implementations in the Stakewise network.

We should note that several optimizations are capable of being applied to the system beyond the ones we have explicitly identified within the report. As an example, the codebase makes no use of the `immutable` keyword which is proxy-compliant as it affects the bytecode rather than storage of a contract and would significantly reduce read-access gas costs in a lot of areas, such as the canonical ETH2 deposit contract. Additionally, the codebase makes certain assumptions as to the reader's aptitude in understanding the codebase and contains little to no comments in areas where it should, such as the `(66.66~%, 100%]` consortium level applied for `Oracles` votes or the fractional maximum pending operator threshold which results in a `[0,1)` active to pending validator ratio for permitting instant token mints.

Files in Scope	Repository	Commit(s)
ERC20Upgradeable.sol (ERC)	contracts	2608b37dfd, 63d0cccb5e
ERC20PermitUpgradeable.sol (ERP)	contracts	2608b37dfd, 63d0cccb5e
MerkleDistributor.sol (MDR)	contracts	2608b37dfd, 63d0cccb5e
Oracles.sol (ORA)	contracts	2608b37dfd, 63d0cccb5e
OwnablePausable.sol (OPE)	contracts	2608b37dfd, 63d0cccb5e

Files in Scope	Repository	Commit(s)
OwnablePausableUpgradeable.sol (OPU)	contracts	2608b37dfd, 63d0cccb5e
Pool.sol (POO)	contracts	2608b37dfd, 63d0cccb5e
PoolValidators.sol (PVS)	contracts	2608b37dfd, 63d0cccb5e
Roles.sol (ROL)	contracts	2608b37dfd, 63d0cccb5e
RewardEthToken.sol (RET)	contracts	2608b37dfd, 63d0cccb5e
StakedEthToken.sol (SET)	contracts	2608b37dfd, 63d0cccb5e

During the audit, we filtered and validated a total of **9 findings utilizing static analysis** tools as well as identified a total of **26 findings during the manual review** of the codebase. We strongly recommend that any minor severity or higher findings are dealt with promptly prior to the project's launch as they introduce potential misbehaviours of the system as well as exploits.

The list below covers each segment of the audit in depth and links to the respective chapter of the report:

- 💻 **Compilation**
- 🔍 **Static Analysis**
- 👁 **Manual Review**
- 🖊 **Code Style**

Compilation

The project utilizes `hardhat` as its development pipeline tool, containing an array of tests and scripts coded in JavaScript.

To compile the project, the `compile` command needs to be issued via the `npx` CLI tool to `hardhat`:

```bash
npx hardhat compile
```

The `hardhat` tool automatically selects Solidity version `0.7.5` based on the version specified within the `hardhat.config.js` file.

The project contains discrepancies with regards to the Solidity version used, however, they are constrained in the external dependencies of the project and can be safely ignored.

The Stakewise team has locked the `pragma` statements to `0.7.5` which is also the version we utilized for our static analysis as well as optimizational review of the codebase.

During compilation with the `hardhat` pipeline, no errors were identified that relate to the syntax or bytecode size of the contracts.

Given that the compiler version utilized is one that has been seldomly used in production and the codebase makes extensive use of concepts that strain the capabilities of the compiler such as multiple dynamic array arguments, consecutive `keccak256` instructions and more during the audit we also assessed the codebase's susceptibility to those compiler vulnerabilities.

The list of known bugs applicable to the compiler version utilized by the project surface only when `abi.decode` is utilized (SOL-2021-2), when `immutable` variables are used (SOL-2021-3), or when `keccak256` operations are performed consecutively in an `assembly` block (SOL-2021-1). Neither of those traits was observable in the codebase and as such it does not suffer from any known vulnerabilities.

We should note that due to the said compiler version being seldomly used in production, we strongly advise the Stakewise team to closely monitor compiler vulnerability disclosures as they are released and to take the appropriate actions necessary to remediate them should they arise.

Static Analysis

The execution of our static analysis toolkit identified **226 potential issues** within the codebase of which **208 were ruled out to be false positives** or negligible findings.

The remaining **18 issues** were validated and grouped and formalized into the **9 exhibits** that follow:

ID	Severity	Addressed	Title
ERC-01S	Informational	Yes	Redundant Empty Code Block
ORA-01S	Informational	Yes	Variable Data Location Optimization
POO-01S	Minor	Yes	Potentially Misconfigured Upgrade
POO-02S	Informational	Yes	Variable Data Location Optimization
PVS-01S	Minor	Yes	Inexistent Zero-Based Input Validation
RET-01S	Minor	Yes	Inexistent Sanitization Against Claim to Zero
RET-02S	Minor	Yes	Inexistent Zero-Based Input Validation
RET-03S	Informational	No	Usage of Accuracy Numeric Literal
RET-04S	Informational	No	Usage of Percentage Numeric Literal

Manual Review

A **thorough line-by-line review** was conducted on the codebase to identify potential malfunctions and vulnerabilities in the ETH2 stakign implementation of Stakewise.

As the project at hand implements an ETH2 staking solution, intricate care was put into ensuring that the **flow of funds within the system conforms to the specifications and restrictions** laid forth both within the protocol's specification as well as the ETH2 protocol it is interfacing with.

We validated that **all state transitions of the system occur within sane criteria** and that all rudimentary formulas within the system execute as expected. We **pinpointed multiple misbehaviours** within the system including one which could have had **severe ramifications** to its overall operation, however, they were conveyed ahead of time to the Stakewise team to be **promptly remediated**.

Additionally, the system was investigated for any other commonly present attack vectors such as re-entrancy attacks, mathematical truncations, logical flaws and **ERC / EIP** standard inconsistencies. The documentation of the project was satisfactory to a certain extent, however, we strongly recommend the documentation of the project to be expanded at certain ambiguous points such as the numeric literals utilized across it.

A total of **26 findings** were identified over the course of the manual review of which **11 findings** concerned the behaviour and security of the system. The non-security related findings, such as optimizations, are included in the separate **Code Style** chapter.

The finding table below enumerates all these security / behavioural findings:

ID	Severity	Addressed	Title
ERP-01M	Minor	No	Non-Standard Upgradeable Initialization Pattern
MDR-01M	Minor	Yes	Improper Pause State Access Control
MDR-02M	Minor	Yes	Ineffectual Duration Argument
MDR-03M	Minor	Yes	Potentially Unclaimed Rewards
MDR-04M	Informational	Yes	Potential Claim Race Condition

ID	Severity	Addressed	Title
MDR-04M	Informational	Yes	Potential Claim Race Condition
ORA-01M	Medium	Yes	Inexistent Validation of Signature Payload Submitter
ORA-02M	Medium	Yes	Single Point of Failure
POO-01M	Minor	Yes	Inexistent Validation of Amount
PVS-01M	Minor	Yes	Inexistent Removal of Validator Status
RET-01M	Major	Yes	Circumvention of Checkpointing Mechanism
ROL-01M	Minor	No	Event-Based Role Management

Code Style

During the manual portion of the audit, we identified **15 optimizations** that can be applied to the codebase that will decrease the gas-cost associated with the execution of a particular function and generally ensure that the project complies with the latest best practices and standards in Solidity.

Additionally, this section of the audit contains any opinionated adjustments we believe the code should make to make it more legible as well as truer to its purpose.

These optimizations are enumerated below:

ID	Severity	Addressed	Title
ERP-01C	Informational	No	Sub-Optimal EIP-712 Implementation
ERP-02C	Informational	No	Unoptimized Variable Mutability
ERC-01C	Informational	No	Deprecated Maximum Representation Style
ORA-01C	Informational	Yes	Inefficient Block Number Comparison
ORA-02C	Informational	Yes	Multiple Top-Level Declarations
ORA-03C	Informational	No	Redundant Visibility Specifier
ORA-04C	Informational	Yes	Undocumented Consortium Level
OPE-01C	Informational	No	Redundant Visibility Specifier
OPU-01C	Informational	No	Redundant Visibility Specifier
POO-01C	Informational	Yes	Undocumented Value Literal
PVS-01C	Informational	No	Redundant Root Validations
RET-01C	Informational	Yes	Duplicate Event Emittance & Storage Write
ROL-01C	Informational	Yes	Unspecified Numerical Accuracy

ID	Severity	Addressed	Title
SET-01C	Informational	No	Incorrect Gas Optimization
SET-02C	Informational	No	Potential XOR Optimization

ERC20Upgradeable Static Analysis Findings

ON THIS PAGE

ERC-01S: Redundant Empty Code Block

Type	Severity	Location
Code Style	Informational ●	ERC20Upgradeable.sol:L202

Description:

The `_transfer` function contains an empty code block as it is meant to be overridden by contracts that inherit the contract in question.

Example:

contracts/tokens/ERC20Upgradeable.sol

SOL

Copy

```
function _transfer(address sender, address recipient, uint256 amount) internal vi
```

Recommendation:

In order to ensure that the function is always overridden, we advise the brackets to be omitted (`{}`) and the declaration to simply terminate with the `;` character to mandate derivative contracts to `override` this method or not compile.

Alleviation:

The function now properly terminates with the `;` character.

Oracles Static Analysis Findings

ON THIS PAGE

ORA-01S: Variable Data Location Optimization

Type	Severity	Location
Gas Optimization	Informational	Oracles.sol:L148, L194, L195, L237, L238, L278, L279, L280

Description:

The linked variables are `memory` arguments in `external` visibility functions.

Example:

contracts/Oracles.sol

SOL

```
235 function initializeValidator(
236     IPoolValidators.DepositData memory depositData,
237     bytes32[] memory merkleProof,
238     bytes[] memory signatures
239 )
240     external override whenNotPaused
241 {
```

Recommendation:

We advise them to be set to `calldata` optimizing the gas cost of the codebase.

Alleviation:

All data location optimization instances were properly adjusted according to our recommendation.

Pool Static Analysis Findings

ON THIS PAGE

POO-01S: Potentially Misconfigured Upgrade

Type	Severity	Location
Input Sanitization	Minor ●	Pool.sol:L58-L67

Description:

The `upgrade` function does not sanitize its input arguments, permitting the `_oracles` value to be the same as the current one thus permitting `validators` to change an arbitrary number of times.

Example:

contracts/pool/Pool.sol

SOL

Copy

```sol
/**
 * @dev See {IPool-upgrade}.
 */
function upgrade(address _poolValidators, address _oracles) external override onl
    require(address(oracles) == 0x2f1C5E86B13a74f5A6E7B4b35DD77fe29Aa47514, "Pool
    
    // set contract addresses
    validators = IPoolValidators(_poolValidators);
    oracles = _oracles;
}
```

Recommendation:

We advise input sanitization to be performed on the arguments, firstly to ensure they are non-zero and secondly to ensure that `_oracles` points to a different address than the current `oracles` implementation.

Alleviation:

The exhibit was partially alleviated by introducing the non-zero `require` check for the linked `_oracles` argument as well as the `_poolValidators` one. As such, we consider this exhibit dealt with.

POO-02S: Variable Data Location Optimization

Type	Severity	Location
Gas Optimization	Informational ●	Pool.sol:L228, L246

Description:

The linked variables are `memory` arguments in `external` visibility functions.

Example:

contracts/pool/Pool.sol

SOL

Copy

```
228 function initializeValidator(IPoolValidators.DepositData memory depositData) exte
```

Recommendation:

We advise them to be set to `calldata` optimizing the gas cost of the codebase.

Alleviation:

The data location specifiers for both instances were properly set to `calldata`.

PoolValidators Static Analysis Findings

ON THIS PAGE

PVS-01S: Inexistent Zero-Based Input Validation

Type	Severity	Location
Input Sanitization	Minor ●	PoolValidators.sol:L38-L45

Description:

The input arguments of the linked function are of the `address` type, are set once and are not validated to be different from the zero-address.

Example:

contracts/pool/PoolValidators.sol

SOL

Copy

```
38 /**
39  * @dev See {IPoolValidators-initialize}.
40  */
41 function initialize(address _admin, address _pool, address _oracles) external ove
42     __OwnablePausableUpgradeable_init(_admin);
43     pool = IPool(_pool);
44     oracles = _oracles;
45 }
```

Recommendation:

We advise such validations to be introduced to ensure no misconfiguration can occur.

Alleviation:

All arguments are now properly sanitized against the zero-address.

RewardEthToken Static Analysis Findings

ON THIS PAGE

RET-01S: Inexistent Sanitization Against Claim to Zero

Type	Severity	Location
Input Sanitization	Minor ●	RewardEthToken.sol:L262-L279

Description:

The `claim` function does not sanitize the `account` argument and it is not sanitized anywhere in the `MerkleDistributor` implementation, meaning that it can even represent the zero-address.

Example:

contracts/tokens/RewardEthToken.sol

SOL

Copy

```sol
262 /**
263  * @dev See {IRewardEthToken-claim}.
264  */
265 function claim(address account, uint256 amount) external override {
266     require(msg.sender == merkleDistributor, "RewardEthToken: access denied");
267 
268     // update checkpoints, transfer amount from distributor to account
269     uint128 _rewardPerToken = rewardPerToken;
270     checkpoints[address(0)] = Checkpoint({
271         reward: _balanceOf(address(0), _rewardPerToken).sub(amount).toUint128(),
272         rewardPerToken: _rewardPerToken
273     });
274     checkpoints[account] = Checkpoint({
275         reward: _balanceOf(account, _rewardPerToken).add(amount).toUint128(),
```

```
        rewardPerToken: _rewardPerToken
    });
    emit Transfer(address(0), account, amount);
}
```

Recommendation:

We advise a `require` to be introduced ensuring that the `account` cannot be zero. While it does not affect the correctness of the implementation, it will `emit` a non-standard `Transfer` event from the zero address to the zero address incorrectly.

Alleviation:

A `require` check was introduced properly validating that the `account` argument is different than the zero-address.

RET-02S: Inexistent Zero-Based Input Validation

Type	Severity	Location
Input Sanitization	Minor ●	RewardEthToken.sol:L54-L60

Description:

The input arguments of the linked function are of the `address` type, are set once and are not validated to be different from the zero-address.

Example:

contracts/tokens/RewardEthToken.sol

SOL

Copy

```sol
/**
 * @dev See {IRewardEthToken-upgrade}.
 */
function upgrade(address _oracles) external override onlyAdmin whenPaused {
    require(address(oracles) == 0x2f1C5E86B13a74f5A6E7B4b35DD77fe29Aa47514, "Rewa
    oracles = _oracles;
}
```

Recommendation:

We advise such validations to be introduced to ensure no misconfiguration can occur.

Alleviation:

The `upgrade` function now properly sanitizes its argument against the zero-address, however, it does not validate that it is different than the currently set oracle.

RET-03S: Usage of Accuracy Numeric Literal

Type	Severity	Location
Code Style	Informational ●	RewardEthToken.sol:L191, L222

Description:

The numeric literal `1e18` is meant to represent the accuracy of each `stakedEthToken` unit but is not documented as such.

Example:

contracts/tokens/RewardEthToken.sol

SOL

Copy

```sol
184 function _calculateNewReward(
185     uint256 currentReward,
186     uint256 stakedEthAmount,
187     uint256 periodRewardPerToken
188 )
189     internal pure returns (uint256)
190 {
191     return currentReward.add(stakedEthAmount.mul(periodRewardPerToken).div(1e18))
192 }
```

Recommendation:

We advise all instances of it to be replaced by a contract-level `constant` variable that properly depicts its intention. This does not alter the generated bytecode of the contract and increases the legibility and maintainability of the code.

Alleviation:

The Stakewise team stated that the constant is utilized in a single place only and as such should remain as is.

RET-04S: Usage of Percentage Numeric Literal

Type	Severity	Location
Code Style	Informational ●	RewardEthToken.sol:L92, L220

Description:

The numeric literal `1e4` is meant to represent the `protocolFee` accuracy but is not documented as such.

Example:

contracts/tokens/RewardEthToken.sol

SOL

Copy

```solidity
88 /**
89  * @dev See {IRewardEthToken-setProtocolFee}.
90  */
91 function setProtocolFee(uint256 _protocolFee) external override onlyAdmin {
92     require(_protocolFee < 1e4, "RewardEthToken: invalid protocol fee");
93     protocolFee = _protocolFee;
94     emit ProtocolFeeUpdated(_protocolFee);
95 }
```

Recommendation:

We advise all instances of it to be replaced by a contract-level `constant` variable that properly depicts its intention. This does not alter the generated bytecode of the contract and increases the legibility and maintainability of the code.

Alleviation:

The Stakewise team stated that the constant is utilized in a single place only and as such should remain as is.

ERC20PermitUpgradeable Manual Review Findings

ON THIS PAGE

ERP-01M: Non-Standard Upgradeable Initialization Pattern

ERP-01M: Non-Standard Upgradeable Initialization Pattern

Type	Severity	Location
Logical Fault	Minor ●	ERC20PermitUpgradeable.sol:L37-L40

Description:

The `__ERC20Permit_init` needs to invoke all `unchained` initializer instances of its inherited contracts, however, it does not do so for the `__ERC20_init_unchained` implementation.

Example:

contracts/tokens/ERC20PermitUpgradeable.sol

SOL

Copy

```solidity
23 abstract contract ERC20PermitUpgradeable is Initializable, ERC20Upgradeable, IERC
24     using CountersUpgradeable for CountersUpgradeable.Counter;
25 
26     mapping (address => CountersUpgradeable.Counter) private _nonces;
27 
28     // solhint-disable-next-line var-name-mixedcase
29     bytes32 private _PERMIT_TYPEHASH;
30 
31     /**
32      * @dev Initializes the {EIP712} domain separator using the `name` parameter,
33      *
34      * It's a good idea to use the same `name` that is defined as the ERC20 token
35      */
36     // solhint-disable-next-line func-name-mixedcase
37     function __ERC20Permit_init(string memory name) internal initializer {
38         __EIP712_init_unchained(name, "1");
```

```
        __ERC20Permit_init_unchained();
    }
```

Recommendation:

We advise it to properly do so to avoid improper usage of the `__ERC20Permit_init` function.

Alleviation:

The Stakewise team confirmed this exhibit, however, they will retain the current implementation in place to avoid replacing the `StakeWiseToken` contract.

MerkleDistributor Manual Review Findings

ON THIS PAGE

MDR-01M: Improper Pause State Access Control

Type	Severity	Location
Logical Fault	Minor ●	MerkleDistributor.sol:L69-L75

Description:

The `claim` function's invocation is prohibited when the contract is paused, however, the administrator can still mistakenly distribute tokens in such a state via the `distributePeriodically` and `distributeOneTime` functions.

Example:

contracts/merkles/MerkleDistributor.sol

SOL

Copy

```sol
63 /**
64  * @dev See {IMerkleDistributor-distributePeriodically}.
65  */
66 function distributePeriodically(
67     address from,
68     address token,
69     address beneficiary,
70     uint256 amount,
71     uint256 durationInBlocks
72 )
73     external override onlyAdmin
74 {
75     require(amount > 0, "MerkleDistributor: invalid amount");
```

```

    uint256 startBlock = block.number;
    uint256 endBlock = startBlock + durationInBlocks;
    require(endBlock > startBlock, "MerkleDistributor: invalid blocks duration");

    IERC20Upgradeable(token).safeTransferFrom(from, address(this), amount);
    emit PeriodicDistributionAdded(from, token, beneficiary, amount, startBlock,
}

/**
 * @dev See {IMerkleDistributor-distributeOneTime}.
 */
function distributeOneTime(
    address from,
    address origin,
    address token,
    uint256 amount,
    string memory rewardsLink
)
    external override onlyAdmin
{
    require(amount > 0, "MerkleDistributor: invalid amount");

    IERC20Upgradeable(token).safeTransferFrom(from, address(this), amount);
    emit OneTimeDistributionAdded(from, origin, token, amount, rewardsLink);
}
```

Recommendation:

We advise the `whenNotPaused` modifier to be added to both functions as well, preventing distributions during a pause state.

Alleviation:

The `whenNotPaused` modifier was properly introduced to both instances.

MDR-02M: Ineffectual Duration Argument

Type	Severity	Location
Logical Fault	Minor ●	MerkleDistributor.sol:L79-L81, L84

Description:

The `durationInBlocks` argument of the `distributePeriodically` function is meant to signify the block duration in which the rewards are meant to be distributed, however, no such limitation is placed anywhere in the codebase meaning that the start and end block thresholds are not enforced.

Example:

contracts/merkles/MerkleDistributor.sol

SOL

Copy

```solidity
63 /**
64  * @dev See {IMerkleDistributor-distributePeriodically}.
65  */
66 function distributePeriodically(
67     address from,
68     address token,
69     address beneficiary,
70     uint256 amount,
71     uint256 durationInBlocks
72 )
73     external override onlyAdmin
74 {
75     require(amount > 0, "MerkleDistributor: invalid amount");
76 
77     uint256 startBlock = block.number;
78     uint256 endBlock = startBlock + durationInBlocks;
79     require(endBlock > startBlock, "MerkleDistributor: invalid blocks duration");
80 
81     IERC20Upgradeable(token).safeTransferFrom(from, address(this), amount);
82     emit PeriodicDistributionAdded(from, token, beneficiary, amount, startBlock,
83 }
```

Recommendation:

We advise either the thresholds to be actively enforced in the `claim` function by repurposing the `lastUpdateBlockNumber` variable or the arguments to be omitted entirely as in the current implementation they waste gas while being ineffectual.

Alleviation:

The Stakewise team has responded specifying that this is a necessary argument given that it "...is used by the off-chain oracles to periodically calculate the earned rewards". As a result, we consider this exhibit dealt with.

MDR-03M: Potentially Unclaimed Rewards

Type	Severity	Location
Logical Fault	Minor ●	MerkleDistributor.sol:L53-L61

Description:

The `setMerkleRoot` does not perform any validation on the `_claimedBitMap`, meaning that the previously set merkle root may have had even zero claims made on it.

Example:

contracts/merkles/MerkleDistributor.sol

SOL

Copy

```solidity
53 /**
54  * @dev See {IMerkleDistributor-setMerkleRoot}.
55  */
56 function setMerkleRoot(bytes32 newMerkleRoot, string calldata newMerkleProofs) ex
57     require(msg.sender == address(oracles), "MerkleDistributor: access denied");
58     merkleRoot = newMerkleRoot;
59     lastUpdateBlockNumber = block.number;
60     emit MerkleRootUpdated(msg.sender, newMerkleRoot, newMerkleProofs);
61 }
```

Recommendation:

We advise some form of claim validation to occur, whereby the `_claimedBitMap` is equal to a particular bit sequence as claims can be made on the behalf of other users. Alternatively, if claims on behalf of other users are undesirable this exhibit can be considered null.

Alleviation:

The Stakewise team has stated that should the user not claim their rewards, they will be included in the next Merkle root update thus considering this exhibit null.

MDR-04M: Potential Claim Race Condition

Type	Severity	Location
Logical Fault	Informational	MerkleDistributor.sol:L128, L143, L144

Description:

The `claim` function relies entirely on its arguments to assess the validity of a claim. As a result, it is possible for an arbitrary user to inspect the mempool of the blockchain to replicate the exact same arguments as another pending transaction and make the claim on their behalf. While the funds will still be transferred to the intended recipient, this can cause a user experience misbehaviour as the user would see their transaction fail as already claimed whilst it may have been properly processed.

Example:

contracts/merkles/MerkleDistributor.sol

SOL

Copy

```
126 function claim(
127     uint256 index,
128     address account,
129     address[] calldata tokens,
130     uint256[] calldata amounts,
131     bytes32[] calldata merkleProof
132 )
133     external override whenNotPaused
134 {
135     address _rewardEthToken = rewardEthToken; // gas savings
136     require(
137         IRewardEthToken(_rewardEthToken).lastUpdateBlockNumber() < lastUpdateBloc
138         "MerkleDistributor: merkle root updating"
139     );
140
141     // verify the merkle proof
142     bytes32 _merkleRoot = merkleRoot; // gas savings
143     bytes32 node = keccak256(abi.encode(index, tokens, account, amounts));
144     require(MerkleProofUpgradeable.verify(merkleProof, _merkleRoot, node), "Merkl
145
146     // mark index claimed
```

```
147     _setClaimed(index, _merkleRoot);
```

Recommendation:

We advise the `account` member to be validated to be equal to the `msg.sender` to prevent this condition from arising. If claims on another user's behalf are desired this exhibit can be safely ignored.

Alleviation:

The Stakewise team confirmed that claims on another user's behalf are desired rendering this exhibit null.

Oracles Manual Review Findings

ON THIS PAGE

ORA-01M: Inexistent Validation of Signature Payload Submitter

Type	Severity	Location
Logical Fault	Medium ●	Oracles.sol:L148, L195, L238, L280

Description:

The various signature-based functions of the `Oracles` implementation do not validate the `msg.sender` and thus allow anyone to submit a set of valid `signatures` that will result in the corresponding action being executed. While this allows for versatility, it enables complex attacks to unfold by an attacker inspecting the mempool, identifying the action being performed and executing it themselves with transactions before and after it that would normally be impossible, such as flash loans. An example of this would be the significant dilution of the new reward-per-token increase by a user inspecting the mempool for a valid `submitRewards` invocation, making a flash loan deposit to a `Pool` and in turn to `StakedEthToken` thus diluting the rewards. While the impact is offset by the maximum pending validators threshold, it is still an example of what permissionless submission of vote executions can lead to.

Example:

contracts/Oracles.sol

SOL

Copy

```sol
142 /**
143  * @dev See {IOracles-submitRewards}.
144  */
145 function submitRewards(
```

```
    uint256 totalRewards,
    uint256 activatedValidators,
    bytes[] memory signatures
)
    external override whenNotPaused
{
    require(
        signatures.length.mul(3) > getRoleMemberCount(ORACLE_ROLE).mul(2),
        "Oracles: invalid number of signatures"
    );

    // calculate candidate ID hash
    uint256 nonce = rewardsNonce.current();
    bytes32 candidateId = ECDSAUpgradeable.toEthSignedMessageHash(
        keccak256(abi.encode(nonce, activatedValidators, totalRewards))
    );

    // check signatures and calculate number of submitted oracle votes
    address[] memory signedOracles = new address[](signatures.length);
    for (uint256 i = 0; i < signatures.length; i++) {
        bytes memory signature = signatures[i];
        address signer = ECDSAUpgradeable.recover(candidateId, signature);
        require(hasRole(ORACLE_ROLE, signer), "Oracles: invalid signer");

        for (uint256 j = 0; j < i; j++) {
            require(signedOracles[j] != signer, "Oracles: repeated signature");
        }
        signedOracles[i] = signer;
        emit RewardsVoteSubmitted(msg.sender, signer, nonce, totalRewards, activa
    }

    // increment nonce for future signatures
    rewardsNonce.increment();

    // update total rewards
    rewardEthToken.updateTotalRewards(totalRewards);

    // update activated validators
    if (activatedValidators != pool.activatedValidators()) {
        pool.setActivatedValidators(activatedValidators);
    }
}
```

Recommendation:

We advise the functions handling `signatures` payloads to either be invoked only by existing

`ORACLE_ROLE` members or by ensuring that the invocator is equal to the `tx.origin`. The latter would be a temporary solution as **EIP-3074** will deprecate this security feature, however, it will be valid for at least the foreseeable future (over 6 month lifetime) given that it would be a consortium upgrade. Should it be applied, we advise the Stakewise team to simply monitor upcoming Ethereum upgrades and adjust the code as necessary given that the upgrade-able nature of the contract permits them to.

Alleviation:

Caller validation was introduced to the sensitive subset of functions exposed by the contracts thus alleviating this exhibit.

ORA-02M: Single Point of Failure

Type	Severity	Location
Logical Fault	Medium	Oracles.sol:L118-L132

Description:

The contract suffers from a SPoF whereby an oracle's membership is completely dictated by either the role administrator or the administrator of the contract which is able to grant such a role. This can affect consortiums and to that extent all votes processed via the system.

Example:

contracts/Oracles.sol

SOL

Copy

```
118 /**
119  * @dev See {IOracles-addOracle}.
120  */
121 function addOracle(address account) external override {
122     grantRole(ORACLE_ROLE, account);
123     emit OracleAdded(account);
124 }
125
126 /**
127  * @dev See {IOracles-removeOracle}.
128  */
129 function removeOracle(address account) external override {
130     revokeRole(ORACLE_ROLE, account);
131     emit OracleRemoved(account);
132 }
```

Recommendation:

We advise this trait to be carefully examined and if deemed undesirable, we advise the inclusion and removal of new oracles to be performed via an on-chain vote instead.

Alleviation:

The Stakewiste team stated that the administrator of the system is the Stakewise DAO which can only perform actions after votes have been processed and properly timelocked. As a result, we consider this exhibit dealt with.

Pool Manual Review Findings

ON THIS PAGE

POO-01M: Inexistent Validation of Amount

Type	Severity	Location
Logical Fault	Minor ●	Pool.sol:L209-L219

Description:

The `activateMultiple` function does not validate the `amount` of a particular `validatorIndex`, allowing fake `Activated` events to be emitted along real ones which can cause off-chain processes to misbehave.

Example:

contracts/pool/Pool.sol

SOL

Copy

```sol
/**
 * @dev See {IPool-activate}.
 */
function activate(address account, uint256 validatorIndex) external override when
    require(
        validatorIndex.mul(1e4) <= activatedValidators.mul(pendingValidatorsLimit
        "Pool: validator is not active yet"
    );

    uint256 amount = activations[account][validatorIndex];
    require(amount > 0, "Pool: invalid validator index");

    delete activations[account][validatorIndex];
    stakedEthToken.mint(account, amount);
    emit Activated(account, validatorIndex, amount, msg.sender);
}

```

```
/**
 * @dev See {IPool-activateMultiple}.
 */
function activateMultiple(address account, uint256[] calldata validatorIndexes) e
    uint256 toMint;
    uint256 _activatedValidators = activatedValidators;
    for (uint256 i = 0; i < validatorIndexes.length; i++) {
        uint256 validatorIndex = validatorIndexes[i];
        require(
            validatorIndex.mul(1e4) <= _activatedValidators.mul(pendingValidators
            "Pool: validator is not active yet"
        );

        uint256 amount = activations[account][validatorIndex];
        toMint = toMint.add(amount);
        delete activations[account][validatorIndex];

        emit Activated(account, validatorIndex, amount, msg.sender);
    }
    require(toMint > 0, "Pool: invalid validator index");
    stakedEthToken.mint(account, toMint);
}
```

Recommendation:

We strongly recommend the code of `activate` to be refactored to invoke an internal `_activate` function that yields the `amount` of tokens that should be minted and the function to be utilized by both `activate` and `activateMultiple` as the `activate` implementation does impose the proper check on the `amount` being activated.

Alleviation:

The code was refactored according to our recommendation, utilizing an internal `_activateAmount` function that performs the equivalent statements of both implementations in a gas-efficient manner.

PoolValidators Manual Review Findings

ON THIS PAGE

PVS-01M: Inexistent Removal of Validator Status

Type	Severity	Location
Logical Fault	Minor ●	PoolValidators.sol:L136-L150

Description:

The `removeOperator` function does not completely omit the operator entry from the contract as the `validatorStatuses` entry remains unaffected.

Example:

contracts/pool/PoolValidators.sol

SOL

Copy

```sol
136 /**
137  * @dev See {IPoolValidators-removeOperator}.
138  */
139 function removeOperator(address _operator) external override whenNotPaused {
140     require(hasRole(DEFAULT_ADMIN_ROLE, msg.sender) || msg.sender == _operator, "
141 
142     Operator storage operator = operators[_operator];
143     require(operator.initializeMerkleRoot != "", "PoolValidators: invalid operato
144     require(!operator.locked, "PoolValidators: operator is locked");
145 
146     // clean up operator
147     delete operators[_operator];
148 
149     emit OperatorRemoved(msg.sender, _operator);
150 }
```

Recommendation:

We advise the status to also be properly updated as in the current implementation an operator can remove themself, withdraw their collateral and remain `Finalized` which may be an undesirable logic path.

Alleviation:

The Stakewise team responded that the code is performing according to the specification as the `validatorStatuses` is meant to represent the current registration status of the validator and shouldn't be cleaned up when a validator is removed. In light of this, we consider this exhibit null.

RewardEthToken Manual Review Findings

ON THIS PAGE

RET-01M: Circumvention of Checkpointing Mechanism

Type	Severity	Location
Logical Fault	Major ●	RewardEthToken.sol:L173

Description:

The `_updateRewardCheckpoint` function assigns the existing `rewardPerToken` to a user should they have a zero balance, meaning that the checkpointing system can be circumvented to take advantage of a huge delta between `newRewardPerToken` and `cp.rewardPerToken` on new accounts, acquiring a disproportionate reward.

Example:

contracts/tokens/RewardEthToken.sol

SOL

Copy

```sol
159 function _updateRewardCheckpoint(address account, uint128 newRewardPerToken) inte
160     Checkpoint memory cp = checkpoints[account];
161     if (newRewardPerToken == cp.rewardPerToken) return;
162 
163     uint256 stakedEthAmount;
164     if (account == address(0)) {
165         // fetch merkle distributor current principal
166         stakedEthAmount = stakedEthToken.distributorPrincipal();
167     } else {
168         stakedEthAmount = stakedEthToken.balanceOf(account);
169     }
170     if (stakedEthAmount == 0) {
171         checkpoints[account] = Checkpoint({
172             reward: cp.reward,
173             rewardPerToken: cp.rewardPerToken
174         });
```

```
        });
    } else {
        uint256 periodRewardPerToken = uint256(newRewardPerToken).sub(cp.rewardPe
        checkpoints[account] = Checkpoint({
            reward: _calculateNewReward(cp.reward, stakedEthAmount, periodRewardP
            rewardPerToken: newRewardPerToken
        });
    }
}
```

Recommendation:

We advise the `rewardPerToken` to be set to the latest one whenever the target `account` has no ETH staked to ensure such a circumvention is not possible.

Alleviation:

The code now properly introduces a checkpoint of the latest value when the `stakedEthAmount` is equal to `0`, alleviating this exhibit.

Roles Manual Review Findings

ON THIS PAGE

ROL-01M: Event-Based Role Management

Type	Severity	Location
Logical Fault	Minor ●	Roles.sol:L21-L69

Description:

The way roles are managed in the contract is purely ephemeral and does not rely on any contract-level storage.

Example:

contracts/Roles.sol

SOL

Copy

```solidity
21 /**
22  * @dev See {IRoles-setOperator}.
23  */
24 function setOperator(address account, uint256 revenueShare) external override onl
25     require(account != address(0), "Roles: account is the zero address");
26     require(revenueShare <= 1e4, "Roles: invalid revenue share");
27     emit OperatorUpdated(account, revenueShare);
28 }
29
30 /**
31  * @dev See {IRoles-removeOperator}.
32  */
33 function removeOperator(address account) external override onlyAdmin whenNotPause
34     require(account != address(0), "Roles: account is the zero address");
35     emit OperatorRemoved(account);
36 }
37
38 /**
```

```
 * @dev See {IRoles-setPartner}.
 */
function setPartner(address account, uint256 revenueShare) external override only
    require(account != address(0), "Roles: account is the zero address");
    require(revenueShare <= 1e4, "Roles: invalid revenue share");
    emit PartnerUpdated(account, revenueShare);
}

/**
 * @dev See {IRoles-removePartner}.
 */
function removePartner(address account) external override onlyAdmin whenNotPaused
    require(account != address(0), "Roles: account is the zero address");
    emit PartnerRemoved(account);
}

/**
 * @dev See {IRoles-addReferrer}.
 */
function addReferrer(address account) external override onlyAdmin whenNotPaused {
    require(account != address(0), "Roles: account is the zero address");
    emit ReferrerAdded(account);
}

/**
 * @dev See {IRoles-removeReferrer}.
 */
function removeReferrer(address account) external override onlyAdmin whenNotPause
    require(account != address(0), "Roles: account is the zero address");
    emit ReferrerRemoved(account);
}
```

Recommendation:

While gas efficient, this methodology is primarily prone to block re-organizations at the blockchain level which can cause the off-chain accounting system to break. Secondarily, the Ethereum community advises against using events as a permanent data source as it may change with future EIPs. This concern, however, is minimal given that on such a principle production applications have been built such as Optimism.

Alleviation:

The Stakewise team considered this exhibit but opted to retain the current implementation in place.

ERC20PermitUpgradeable Code Style Findings

ON THIS PAGE

ERP-01C: Sub-Optimal EIP-712 Implementation

Type	Severity	Location
Gas Optimization	Informational	ERC20PermitUpgradeable.sol:L9

Description:

The EIP-712 implementation present in `@openzeppelin/contracts-upgradeable` at version `3.4.1` is sub-optimal as it does not cache the domain separator of the actively deployed blockchain, meaning that there is room for significant gas improvement on the `permit` function.

Example:

contracts/tokens/ERC20PermitUpgradeable.sol

SOL

Copy

```sol
import "@openzeppelin/contracts-upgradeable/drafts/EIP712Upgradeable.sol";
```

Recommendation:

We advise the Stakewise team to consider forking the EIP-712 draft present at the latest **OpenZeppelin iteration** that does apply such a caching optimization to ensure optimal gas usage in their system.

Alleviation:

The Stakewise team confirmed this exhibit, however, they will retain the current implementation in place to avoid upgrading the token contract.

ERP-02C: Unoptimized Variable Mutability

Type	Severity	Location
Gas Optimization	Informational ●	ERC20PermitUpgradeable.sol:L29, L44, L56

Description:

The `_PERMIT_TYPEHASH` value is set only once during the `__ERC20Permit_init_unchained` hook and is being set to a static pre-calculated value.

Example:

contracts/tokens/ERC20PermitUpgradeable.sol

SOL

```
28 // solhint-disable-next-line var-name-mixedcase
29 bytes32 private _PERMIT_TYPEHASH;
30
31 /**
32  * @dev Initializes the {EIP712} domain separator using the `name` parameter, and
33  *
34  * It's a good idea to use the same `name` that is defined as the ERC20 token nam
35  */
36 // solhint-disable-next-line func-name-mixedcase
37 function __ERC20Permit_init(string memory name) internal initializer {
38     __EIP712_init_unchained(name, "1");
39     __ERC20Permit_init_unchained();
40 }
41
42 // solhint-disable-next-line func-name-mixedcase
43 function __ERC20Permit_init_unchained() internal initializer {
44     _PERMIT_TYPEHASH = keccak256("Permit(address owner,address spender,uint256 va
45 }
```

Recommendation:

We advise the variable to be set as `constant`, its assignment to be relocated to its declaration and the `unchained` initializer of the `ERC20Permit` to be omitted as it would be redundant once this optimization is applied.

Alleviation:

The Stakewise team confirmed this exhibit, however, they will retain the current implementation in place to avoid upgrading the token contract.

ERC20Upgradeable Code Style Findings

ON THIS PAGE

ERC-01C: Deprecated Maximum Representation Style

Type	Severity	Location
Code Style	Informational ●	ERC20Upgradeable.sol:L146

Description:

The `uint256(-1)` representation style has been deprecated in favor of the new `type` operator and in particular the `type(uint256).max` statement.

Example:

contracts/tokens/ERC20Upgradeable.sol

SOL

Copy

```sol
146 if (sender != msg.sender && currentAllowance != uint256(-1)) {
```

Recommendation:

We advise the `uint256(-1)` instance to be replaced by its more standardized format.

Alleviation:

The Stakewise team confirmed this exhibit, however, they will retain the current implementation in place to avoid upgrading the token contract.

Oracles Code Style Findings

ON THIS PAGE

ORA-01C: Inefficient Block Number Comparison

Type	Severity	Location
Gas Optimization	Informational	Oracles.sol:L139

Description:

The latter conditional of the `isMerkleRootVoting` function is inefficient as the case whereby `lastRewardBlockNumber` is greater than `block.number` is impossible due to `lastRewardBlockNumber` being set to the current `block.number` by the oracle itself.

Example:

contracts/Oracles.sol

SOL

Copy

```sol
uint256 lastRewardBlockNumber = rewardEthToken.lastUpdateBlockNumber();
return merkleDistributor.lastUpdateBlockNumber() < lastRewardBlockNumber && lastR
```

Recommendation:

We advise the comparison to be changed to an inequality one instead, better illustrating its purpose which is guarding against a reward and merkle root vote to be processed in a single block.

Alleviation:

The comparison was adjusted to an inequality one according to our recommendation.

ORA-02C: Multiple Top-Level Declarations

Type	Severity	Location
Code Style	Informational ●	Oracles.sol:L17, L29, L42

Description:

The `Oracles` contract contains two extra top-level `interface` declarations.

Example:

contracts/Oracles.sol

SOL

Copy

```
17 interface IAccessControlUpgradeable {
18     /**
19      * @dev See {AccessControlUpgradeable-getRoleMemberCount}.
20      */
21     function getRoleMemberCount(bytes32 role) external view returns (uint256);
22
23     /**
24      * @dev See {AccessControlUpgradeable-getRoleMember}.
25      */
26     function getRoleMember(bytes32 role, uint256 index) external view returns (ad
27 }
28
29 interface IPrevOracles {
30     /**
31     * @dev Function for retrieving current rewards nonce.
32     */
33     function currentNonce() external view returns (uint256);
34 }
```

Recommendation:

We advise them to be declared in their dedicated contracts to ensure standard-compliant code structure.

Alleviation:

The top level declarations have been omitted from the codebase and a new `IOraclesV1` file was created and is now imported in their place.

ORA-03C: Redundant Visibility Specifier

Type	Severity	Location
Gas Optimization	Informational	Oracles.sol:L46

Description:

The linked variable is meant to be used as an internally accessible `constant` and has no use outside of the contract as it represents a static value.

Example:

contracts/Oracles.sol

SOL

Copy

```
bytes32 public constant ORACLE_ROLE = keccak256("ORACLE_ROLE");
```

Recommendation:

We advise it to be set to either `internal` or `private` to reduce the bytecode size of the contract.

Alleviation:

The Stakewise team stated that they prefer to retain the current visibility in place to ensure non-technically attuned persons can still read the status of users in the system when using basic tools such as Etherscan.

ORA-04C: Undocumented Consortium Level

Type	Severity	Location
Code Style	Informational ●	Oracles.sol:L153, L201, L243, L285

Description:

The consortium level needed to be achieved for a particular vote is greater-than 66.66~% of the total oracles, as indicated by dividing both members of the inequality by `3` resulting in a `2/3` multiplier for the `signatures`.

Example:

contracts/Oracles.sol

SOL

Copy

```
require(
    signatures.length.mul(3) > getRoleMemberCount(ORACLE_ROLE).mul(2),
    "Oracles: invalid number of signatures"
);
```

Recommendation:

We advise this trait to be properly documented, potentially in a dedicated `pure` function, as currently value literals are directly used that can be ambiguous.

Alleviation:

The consortium calculation is now properly performed by an internal function better illustrating its purpose and optimizing the codebase.

OwnablePausable Code Style Findings

ON THIS PAGE

OPE-01C: Redundant Visibility Specifier

Type	Severity	Location
Gas Optimization	Informational	OwnablePausable.sol:L16

Description:

The linked variable is meant to be used as an internally accessible `constant` and has no use outside of the contract as it represents a static value.

Example:

contracts/presets/OwnablePausable.sol

SOL

```
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
```

Recommendation:

We advise it to be set to either `internal` or `private` to reduce the bytecode size of the contract.

Alleviation:

The Stakewise team stated that they prefer to retain the current visibility in place to ensure non-technically attuned persons can still read the status of users in the system when using basic tools such as Etherscan.

OwnablePausableUpgradeable Code Style Findings

ON THIS PAGE

OPU-01C: Redundant Visibility Specifier

Type	Severity	Location
Gas Optimization	Informational	OwnablePausableUpgradeable.sol:L16

Description:

The linked variable is meant to be used as an internally accessible `constant` and has no use outside of the contract as it represents a static value.

Example:

contracts/presets/OwnablePausableUpgradeable.sol

SOL

Copy

```sol
bytes32 public constant PAUSER_ROLE = keccak256("PAUSER_ROLE");
```

Recommendation:

We advise it to be set to either `internal` or `private` to reduce the bytecode size of the contract.

Alleviation:

The Stakewise team stated that they prefer to retain the current visibility in place to ensure non-technically attuned persons can still read the status of users in the system when using basic tools such as Etherscan.

Pool Code Style Findings

ON THIS PAGE

POO-01C: Undocumented Value Literal

Type	Severity	Location
Code Style	Informational ●	**Pool.sol:L81**, **L169**, **L182**, **L190**, **L211**

Description:

The `pendingValidatorsLimit` is meant to represent a fractional off-set with a maximum of `1e4`, indicating that the `validatorIndex` multipler can at most be `0.5` which implies that the maximum number of pending validators during which tokens are still minted as normal is at most equal to the currently activated ones.

Example:

contracts/pool/Pool.sol

SOL

Copy

```sol
210 require(
211     validatorIndex.mul(1e4) <= _activatedValidators.mul(pendingValidatorsLimit.ad
212     "Pool: validator is not active yet"
213 );
```

Recommendation:

We advise the literal `1e4` to be stored to a contract level `constant` variable that properly illustrates its purpose and the `pendingValidatorsLimit` variable to be documented in the locations it is being utilized as well as its declaration to greatly increase the legibility of the codebase.

Alleviation:

After discussing with the Stakewise team, we concluded that such a change would actually render the codebase less readable given that it would cause all instances to be replaced by a long verbose variable name. As a result, we consider this exhibit null.

PoolValidators Code Style Findings

ON THIS PAGE

PVS-01C: Redundant Root Validations

Type	Severity	Location
Gas Optimization	Informational ●	PoolValidators.sol:L73-L76

Description:

The validations performed in the linked `require` check are all redundant as they are validated one-to-one in the ensuing `require` check. In detail, the `length` of `""` casted to `bytes` is zero meaning that the length comparisons actually cover the inequality with `""` case and if the `keccak256` results of two `bytes` members are different so are their values (note: the opposite relation is not necessarily true).

Example:

contracts/pool/PoolValidators.sol

SOL

Copy

```sol
73 require(
74     initializeMerkleRoot != "" && finalizeMerkleRoot != "" && finalizeMerkleRoot
75     "PoolValidators: invalid merkle roots"
76 );
77 require(
78     bytes(initializeMerkleProofs).length != 0 && bytes(finalizeMerkleProofs).leng
79     keccak256(bytes(initializeMerkleProofs)) != keccak256(bytes(finalizeMerklePro
80     "PoolValidators: invalid merkle proofs"
81 );
```

Recommendation:

We advise the `require` check to be omitted to optimize the gas cost of the function.

Alleviation:

The Stakewise team confirmed this exhibit, however, they will retain the current implementation in place.

RewardEthToken Code Style Findings

ON THIS PAGE

RET-01C: Duplicate Event Emittance & Storage Write

Type	Severity	Location
Gas Optimization	Informational ●	RewardEthToken.sol:L216, L253-L259

Description:

The `updateTotalRewards` function contains a logic path via which the `RewardsUpdated` event is emitted twice with the same arguments and the `lastUpdateBlockNumber` is written twice to the same value.

Example:

contracts/tokens/RewardEthToken.sol

```sol
207 /**
208  * @dev See {IRewardEthToken-updateTotalRewards}.
209  */
210 function updateTotalRewards(uint256 newTotalRewards) external override {
211     require(msg.sender == oracles, "RewardEthToken: access denied");
212
213     uint256 periodRewards = newTotalRewards.sub(totalRewards);
214     if (periodRewards == 0) {
215         lastUpdateBlockNumber = block.number;
216         emit RewardsUpdated(0, newTotalRewards, rewardPerToken, 0, 0);
217     }
218
219     // calculate protocol reward and new reward per token amount
220     uint256 protocolReward = periodRewards.mul(protocolFee).div(1e4);
221     uint256 prevRewardPerToken = rewardPerToken;
222     uint256 newRewardPerToken = prevRewardPerToken.add(periodRewards.sub(protocol
223     uint128 newRewardPerToken128 = newRewardPerToken.toUint128();
```

```

    // store previous distributor rewards for period reward calculation
    uint256 prevDistributorBalance = _balanceOf(address(0), prevRewardPerToken);

    // update total rewards and new reward per token
    (totalRewards, rewardPerToken) = (newTotalRewards.toUint128(), newRewardPerTo

    uint256 newDistributorBalance = _balanceOf(address(0), newRewardPerToken);
    address _protocolFeeRecipient = protocolFeeRecipient;
    if (_protocolFeeRecipient == address(0) && protocolReward > 0) {
        // add protocol reward to the merkle distributor
        newDistributorBalance = newDistributorBalance.add(protocolReward);
    } else if (protocolReward > 0) {
        // update fee recipient's checkpoint and add its period reward
        checkpoints[_protocolFeeRecipient] = Checkpoint({
            reward: _balanceOf(_protocolFeeRecipient, newRewardPerToken).add(prot
            rewardPerToken: newRewardPerToken128
        });
    }

    // update distributor's checkpoint
    if (newDistributorBalance != prevDistributorBalance) {
        checkpoints[address(0)] = Checkpoint({
            reward: newDistributorBalance.toUint128(),
            rewardPerToken: newRewardPerToken128
        });
    }

    lastUpdateBlockNumber = block.number;
    emit RewardsUpdated(
        periodRewards,
        newTotalRewards,
        newRewardPerToken,
        newDistributorBalance.sub(prevDistributorBalance),
        _protocolFeeRecipient == address(0) ? protocolReward: 0
    );
}
```

Recommendation:

We advise a `return` event to be introduced after the first emittance to ensure that the function ends early and does not waste gas executing the ensuing statements as they will be ineffectual in the case that the `periodRewards` are zero.

Alleviation:

Alleviation:

A `return` statement was properly introduced according to our recommendation.

Roles Code Style Findings

ON THIS PAGE

ROL-01C: Unspecified Numerical Accuracy

Type	Severity	Location
Code Style	Informational	Roles.sol:L26, L43

Description:

The `setOperator` and `setPartner` functions apply a maximum limit of `1e4` for the `revenueShare`, however, the actual accuracy of `revenueShare` may be different thus causing ambiguity as to its purpose.

Example:

contracts/Roles.sol

SOL

Copy

```
21 /**
22  * @dev See {IRoles-setOperator}.
23  */
24 function setOperator(address account, uint256 revenueShare) external override onl
25     require(account != address(0), "Roles: account is the zero address");
26     require(revenueShare <= 1e4, "Roles: invalid revenue share");
27     emit OperatorUpdated(account, revenueShare);
28 }
29
30 /**
31  * @dev See {IRoles-removeOperator}.
32  */
33 function removeOperator(address account) external override onlyAdmin whenNotPause
34     require(account != address(0), "Roles: account is the zero address");
35     emit OperatorRemoved(account);
36 }
37
```

```
/**
 * @dev See {IRoles-setPartner}.
 */
function setPartner(address account, uint256 revenueShare) external override only
    require(account != address(0), "Roles: account is the zero address");
    require(revenueShare <= 1e4, "Roles: invalid revenue share");
    emit PartnerUpdated(account, revenueShare);
}
```

Recommendation:

We advise the value to be set to a contract-level `constant` that clearly depicts its purpose via surrounding comments. This does not alter the generated bytecode of the contract and increases the legibility and maintainability of the code.

Alleviation:

A `MAX_PERCENT` constant was introduced to the codebase according to our recommendation.

StakedEthToken Code Style Findings

ON THIS PAGE

SET-01C: Incorrect Gas Optimization

Type	Severity	Location
Gas Optimization	Informational ●	StakedEthToken.sol:L78-L84

Description:

The way the code is structured actually incurs more gas than simply performing a direct assignment to the storage slot as it performs redundant in-memory operations.

Example:

contracts/tokens/StakedEthToken.sol

SOL

```sol
78 uint256 _distributorPrincipal = distributorPrincipal; // gas savings
79 if (senderRewardsDisabled) {
80     _distributorPrincipal = _distributorPrincipal.sub(amount);
81 } else {
82     _distributorPrincipal = _distributorPrincipal.add(amount);
83 }
84 distributorPrincipal = _distributorPrincipal;
```

Recommendation:

We advise the code block to be reverted to the canonical implementation similarly to `toggleRewards` to reduce the gas cost of the function. In general, such optimizations are only valuable when the value that is cached in memory would have been read twice which is not the case here.

Alleviation:

The Stakewise team confirmed this exhibit, however, they will update the live implementation of the contract only when a logic update is also performed to avoid contract upgrades solely for optimizations.

SET-02C: Potential XOR Optimization

Type	Severity	Location
Gas Optimization	Informational	StakedEthToken.sol:L76

Description:

The `if` statement performs a XOR operation between the values of `senderRewardsDisabled` and `recipientRewardsDisabled`. This operation can be optimized in all cases by adjusting the statements from `(a || b) && !(a && b)` to `a ? !b : b`.

Example:

contracts/tokens/StakedEthToken.sol

SOL

Copy

```
if ((senderRewardsDisabled || recipientRewardsDisabled) && !(senderRewardsDisable
```

Recommendation:

Although the gas optimization is minimal, we advise it to be applied as such optimizations can compound to a significant reduction in gas.

Alleviation:

The Stakewise team considered this exhibit but opted to retain the current implementation in place.

Finding Types

ON THIS PAGE

A description of each finding type included in the report can be found below and is linked by each respective finding. A full list of finding types Omniscia has defined will be viewable at the central audit methodology we will publish soon.

External Call Validation

Many contracts that interact with DeFi contain a set of complex external call executions that need to happen in a particular sequence and whose execution is usually taken for granted whereby it is not always the case. External calls should always be validated, either in the form of `require` checks imposed at the contract-level or via more intricate mechanisms such as invoking an external getter-variable and ensuring that it has been properly updated.

Input Sanitization

As there are no inherent guarantees to the inputs a function accepts, a set of guards should always be in place to sanitize the values passed in to a particular function.

Indeterminate Code

These types of issues arise when a linked code segment may not behave as expected, either due to mistyped code, convoluted `if` blocks, overlapping functions / variable names and other ambiguous statements.

Language Specific

Language specific issues arise from certain peculiarities that the Solidity language boasts that discerns it from other conventional programming languages. For example, the EVM is a 256-bit machine meaning that operations on less-than-256-bit types are more costly for the EVM in terms of gas costs, meaning that loops utilizing a `uint8` variable because their limit will never exceed the 8-bit range actually cost more than redundantly using a `uint256` variable.

Code Style

An official Solidity style guide exists that is constantly under development and is adjusted on each new Solidity release, designating how the overall look and feel of a codebase should be. In these types of findings, we identify whether a project conforms to a particular naming convention and whether that convention is consistent within the codebase and legible. In case of inconsistencies, we point them out under this category. Additionally, variable shadowing falls under this category as well which is identified when a local-level variable contains the same name as a contract-level variable that is present in the inheritance chain of the local execution level's context.

Gas Optimization

Gas optimization findings relate to ways the codebase can be optimized to reduce the gas cost involved with interacting with it to various degrees. These types of findings are completely optional and are pointed out for the benefit of the project's developers.

Standard Conformity

These types of findings relate to incompatibility between a particular standard's implementation and the project's implementation, oftentimes causing significant issues in the usability of the contracts.

Mathematical Operations

In Solidity, math generally behaves differently than other programming languages due to the constraints of the EVM. A prime example of this difference is the truncation of values during a division which in turn leads to loss of precision and can cause systems to behave incorrectly when dealing with percentages and proportion calculations.

Logical Fault

This category is a bit broad and is meant to cover implementations that contain flaws in the way they are implemented, either due to unimplemented functionality, unaccounted-for edge cases or similar extraordinary scenarios.