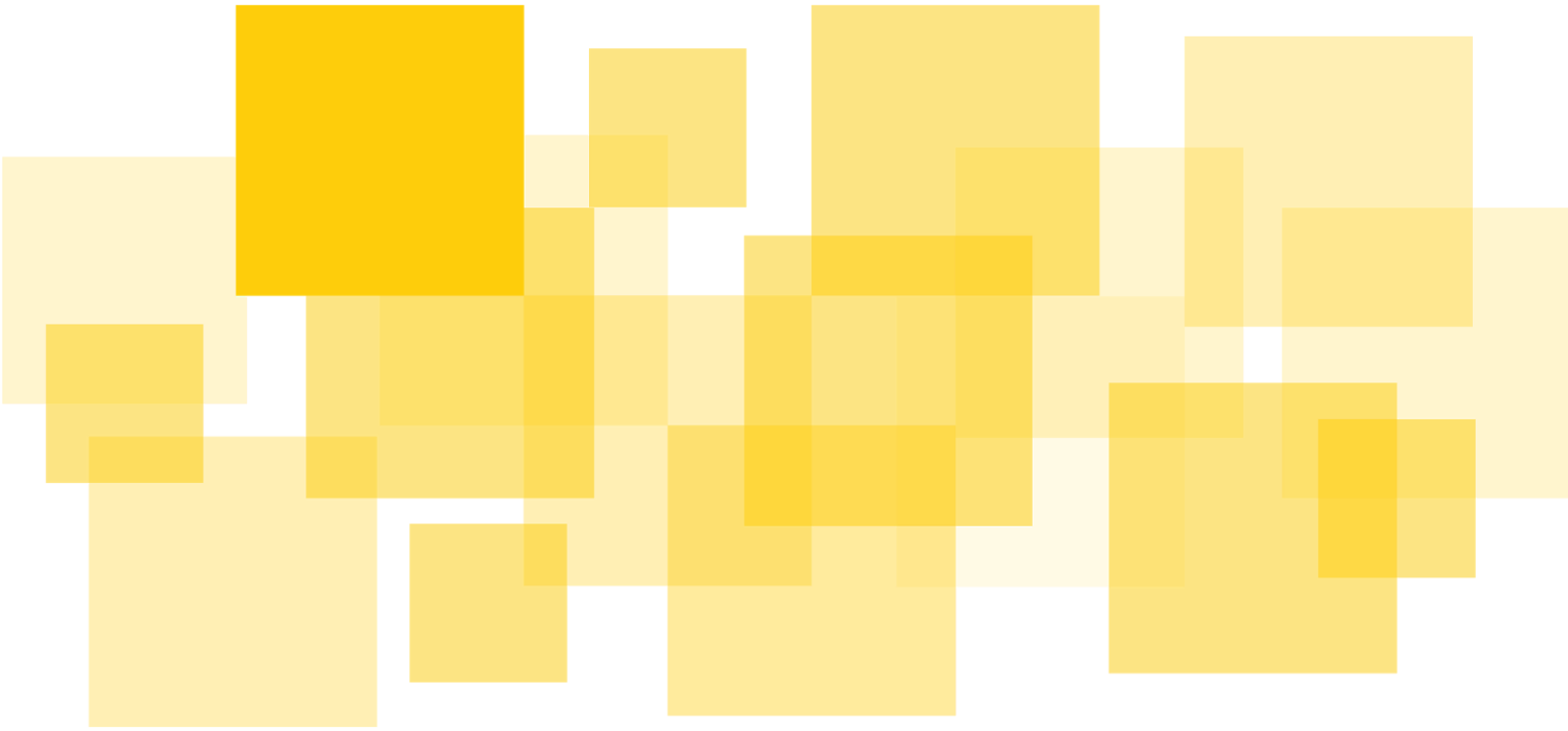
# Security Audit Report

## StakeWise Contract

**Delivered: December 23th, 2020**

**Updated: January 14th, 2021**



**Prepared for StakeWise by**

runtime verification

# Summary

Runtime Verification, Inc. conducted a security audit on the StakeWise smart contracts.

The first iteration of the audit was conducted by Daejun Park over the course of two calendar weeks. This focused on reviewing the business logic of the contracts and identifying any logical loopholes that could cause the system to malfunction or be exploited.

**Updated (Jan 12)**: The second iteration was conducted by Daejun Park and Yi Zhang over the course of two person-weeks. This focused on reviewing the lower level details of both source code and bytecode. This also included reviewing the code changes for fixing the issues found in the first iteration.

The audit led to six findings and nine informative findings and recommendations. The six findings include four implementation errors (A01, A02, A03, and A05) and two business logic flaws (A04 and A06). The nine informative findings and recommendations are about arithmetic (B01, B02, and B09), access controls (B03, B04, B07, and B08), business logic (B05), and gas optimization (B06). All findings have been fixed by the StakeWise team, and all informative findings and recommendations were acknowledged or adopted.

### Scope

The target of the audit is the smart contracts source files at git-commit-id 19da824f44079c2b94c8cca84de77f5bdf3f0e54. Below is the list of the source files:

- contracts/BalanceReporters.sol
- contracts/collectors/Pool.sol
- contracts/collectors/Solos.sol
- contracts/interfaces/*.sol
- contracts/presets/OwnablePausable.sol
- contracts/presets/OwnablePausableUpgradeable.sol
- contracts/tokens/ERC20.sol
- contracts/tokens/RewardEthToken.sol
- contracts/tokens/StakedEthToken.sol
- contracts/tokens/StakedTokens.sol
- contracts/Validators.sol

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase as well as deployment and upgrade scripts are *not* in the scope of this engagement.

Moreover, in this engagement, we gave a higher priority to reviewing the internal logic of the contracts, rather than comprehensively identifying the potential behaviors of external interactions with liquidity providers such as Uniswap.

**Updated (Jan 12)**: In the second iteration of the audit, the additional code changes made up to f2d9985131fd6c3143a4b0df272fca62cbe0d464 were also independently reviewed. However, the review focused more on the correctness of the code changes themselves rather than completely analyzing their impact on other parts of the contracts.

## Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Third, we symbolically executed the bytecode of the contract to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level, that are due to EVM quirks or Solidity compiler bugs. Finally, we employed Firefly to measure the test coverage at the bytecode level, identifying missing test scenarios, and helping to improve the quality of tests.

# Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

# Assumptions

This audit is based on the following assumptions and trust model.

The authorized users (i.e., Owners, Admins, Pausers, Operators, and Reporters) are assumed to behave correctly and honestly, where they are given the following authority:

- The Owners will initially deploy the contracts of the system, setting up all system parameters. They only can upgrade the contracts later except Solos.
- Only the Admins can change the system parameters via setter functions. Only the Admins can add or remove Pausers, Operators, and Reporters, as well as other Admins. Only the Admins can enable or disable LP tokens to be supported in StakedTokens.
- Only the Pausers can pause or unpause the system contracts. The Admins are not necessarily a Pauser.
- Only the Operators can transfer users' funds to the Ethereum 2.0 deposit contract and register validators for Solos and Pool. The Admins are not necessarily an Operator.
- Only the Reporters can update the total rewards via voteForTotalRewards().

The liquidity providers associated with StakedTokens are trusted to behave correctly and honestly. Specifically, they are assumed to not transfer the collateralized stETH/rwETH tokens to others without approval from the users, nor mint their LP tokens without proper collateral.

The deployment and upgrade scripts are assumed to be correct. Specifically, they are assumed to correctly set up the system parameters and the addresses to the trusted external contracts and the other system contracts. Also the upgrade scripts are assumed to be able to detect any inconsistencies between the old and new storage layouts, so that the existing storage is never corrupted in the upgrade process.

The OpenZeppelin libraries are assumed to be correct. Below is the list of dependencies:

- @openzeppelin/contracts-upgradeable/access/AccessControlUpgradeable.sol
- @openzeppelin/contracts-upgradeable/math/SafeMathUpgradeable.sol
- @openzeppelin/contracts-upgradeable/proxy/Initializable.sol
- @openzeppelin/contracts-upgradeable/token/ERC20/IERC20Upgradeable.sol
- @openzeppelin/contracts-upgradeable/utils/CountersUpgradeable.sol

- @openzeppelin/contracts-upgradeable/utils/PausableUpgradeable.sol
- @openzeppelin/contracts-upgradeable/utils/ReentrancyGuardUpgradeable.sol
- @openzeppelin/contracts/access/AccessControl.sol
- @openzeppelin/contracts/math/SafeMath.sol
- @openzeppelin/contracts/token/ERC20/IERC20.sol
- @openzeppelin/contracts/token/ERC20/SafeERC20.sol
- @openzeppelin/contracts/utils/Address.sol
- @openzeppelin/contracts/utils/Pausable.sol
- @openzeppelin/contracts/utils/ReentrancyGuard.sol
- @uniswap/v2-core/contracts/interfaces/IUniswapV2Pair.sol

**Updated (Jan 12)**: The code changes include the following additional dependencies:

- @openzeppelin/contracts-upgradeable/utils/SafeCastUpgradeable.sol
- @openzeppelin/contracts-upgradeable/cryptography/ECDSA.sol
- @openzeppelin/contracts-upgradeable/drafts/EIP712Upgradeable.sol
- @openzeppelin/contracts-upgradeable/drafts/ERC20PermitUpgradeable.sol

# Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and need to be properly addressed.

## A01: Flaws in StakedTokens contract implementation

The `rewardRates[]` mapping in StakedTokens does not store the user's reward rate separately for each token, and thus StakedTokens._withdrawRewards() will not work correctly when there are multiple tokens.

### Recommendation

Fix StakedTokens to store the reward rate for each (token, user) pair, as it does with `balances[][]`.

More specifically,

- Add a new field, say rewardRate, in the Token struct, that stores the token's reward rate.
- Modify `rewardRates[]` to be a nested mapping like `balances[][]`, and store the reward rate for each (token, user) pair.

### Status

Fixed as recommended in the latest version.

## A02: Error in RewardEthToken._transfer() implementation

When their rwETH balance is negative, users can increase the balance from a negative value to zero by sending a transaction of `RewardEthToken.transfer(recipient, 0)` for any non-zero recipient address.

This is because of a bug in the RewardEthToken._transfer() function, in which the balance of the sender is updated to `balanceOf(sender).sub(amount)`, where `balanceOf(sender)` returns zero even when the sender's balance is negative.

### Recommendation

Revert when the sender's balance is negative, or simply replace `balanceOf(sender)` with `rewardOf(sender).toUint256()`.

### Status

Fixed as recommended in the latest version.

## A03: Off-by-one error in consensus condition for balance reporting oracles

The amount of total rewards is periodically updated by oracles (called balance reporters) only when ⅔ of reporters agree on the same value. The implementation of the consensus condition has an off-by-one error, allowing conflicting values to be updated on different chains in case of ⅓ of reporters being malicious under the asynchronous network condition. The off-by-one error is critical because the total number of reporters is rather small, i.e., 3 to 5 in the initial phase of operation.

### Recommendation

For the Byzantine fault tolerance (i.e., to be tolerant up to ⅓ of reporters being Byzantine failed), the consensus requires *strictly* more than ⅔ of reporters to agree on the same value.

That is, fix the consensus condition to be simply as follows:

```
candidates[candidateId].mul(3) > totalReporters.mul(2)
```

In case that votesThreshold needs to be kept as a fraction of 10^18, fix it to be as follows:

```
votesThreshold = 666666666666666667;
```

(not 666666666666666666), and fix the consensus condition to be as follows:

```
candidates[candidateId].mul(1e18) > totalReporters.mul(votesThreshold)
```

Note that the condition requires > instead of >=, and no division is needed.

### Status

Fixed as recommended in the latest version.

## A04: Total supply of stETH/rwETH tokens could exceed the actual balance of the validator pool

The total supply of stETH/rwETH tokens could exceed the actual balance of the validator pool, in case that there exist users whose balance of stETH/rwETH tokens falls below zero.

### Scenario

Suppose that Alice initially has (10 stETH, 0 rwETH) tokens at the beginning of the first month, and she is the sole staker for the validator pool. Suppose that the validator pool gets penalties -1 rwETH over the first month, which in turn are given to Alice, and thus her balance at the end of the first month will be (10 stETH, -1 rwETH). Suppose that she transfers 9 stETH to Bob at the beginning of the second month, thus Alice's balance at that point will be (1 stETH, -1 rwETH), and Bob's balance will be (9 stETH, 0 rwETH). Note that at that point, the total supply of stETH/rwETH tokens is 9, which is equal to the actual balance of the validator pool. Now suppose that the validator pool gets further penalties of another -1 rwETH over the second month, which in turn are distributed to Alice and Bob proportionally, that is, -0.1 rwETH to Alice, and -0.9 rwETH to Bob. Thus, at the end of the second month, Alice's balance will be (1 stETH, -1.1 rwETH), and Bob's balance will be (9 stETH, -0.9 rwETH), which means that Alice's balance falls below zero, and the total supply exceeds the actual balance. In other words, Bob can request to redeem up to his 8.1 tokens but it cannot be fully paid back by the validator pool.

### Recommendation

Invest sufficient effort for carefully (re-)designing and properly implementing the instant penalties distribution feature. In the meantime, remove the feature.

### Status

The instant penalties distribution feature has been removed in the latest version, and thus this issue is no longer applicable.

## A05: StakedEthToken.balanceOf() could revert

StakedEthToken.balanceOf() will revert (at the last toUint256() call) for an user whose balance of stETH + rwETH tokens falls below zero. Since StakedEthToken is an ERC20 token, this behavior is problematic for any other contracts that interact with StakedEthToken expecting balanceOf() will never fail except the out-of-gas failure.

### Recommendation

For the short term, return 0 when the actual balance is negative, being consistent with the other similar functions.

For the long term, invest sufficient effort for carefully (re-)designing and properly implementing the instant penalties distribution feature. In the meantime, remove the feature.

### Status

The instant penalties distribution feature has been removed in the latest version, and thus this issue is no longer applicable.

## A06: Newly generated rewards are vulnerable to attack by malicious balance reporters

### Exploit scenario

Suppose the total rewards update threshold is *k*. A malicious balance reporter waits for *k-1* votes to arrive, after which he immediately executes the following sequence of multiple contract calls in a *single* transaction:

1. Flash loan/swap to borrow a sufficiently large amount of LP tokens, say *n* tokens.
2. Stake the borrowed *n* LP tokens by calling `StakedTokens.stakeTokens()`. Let the total amount of staked LP tokens in StakedTokens be *N*.
3. Cast the *k*-th vote by calling `BalanceReporters.voteForTotalRewards()`, which in turn will update the total reward by executing `RewardEthToken.updateTotalRewards()`.
4. Withdraw the *n* LP tokens (staked in step 2) by calling `StakedTokens.withdrawTokens()`, which will distribute the newly generated rewards multiplied by *n/N* to the malicious reporter.
5. Pay back the *n* LP tokens to the flash loan/swap provider (used in step 1).

This allows the malicious balance reporter to earn free rewards without staking any LP tokens, by stealing (a large portion of) newly generated rewards that should have been distributed to other users. Note that the malicious balance reporter does *not* need to violate any rules of the balance update protocol for this attack.

A similar but simpler attack is also possible using stETH tokens. In that case, the malicious balance reporter can simply borrow stETH tokens, make the *k*-th vote, and pay back the tokens, all in a single transaction, which doesn't require the stake and unstake steps.

### Difficulty

This attack vector requires the balance reporting role that is given only to a small number of authorized and trusted parties, thus is exploitable in rather limited circumstances.

### Recommendation

Further investigate the practical impacts of this vulnerability, and other potential issues, if any, along this line.

### Status

The StakeWise team reported that PR 62 fixed this issue by disallowing token transfers during the total rewards update, but we have not comprehensively reviewed its impact on other parts of the contracts.

# Informative Findings & Recommendations

## B01: Rounding errors in instant penalty distribution

In Solidity, the integer division rounds towards zero, which means that the negative result of division is rounded to a bigger value, e.g., “-5 / 3” results in “-1”. Thus the use of the integer division could be problematic in a place where the flooring division is required, e.g., “-5 / 3” needs to be “-2”.

Specifically, RewardEthToken.updateTotalRewards() computes rewardPerToken, the amount of new rewards (in wei) per token (in ether) to be distributed to users, by dividing the new rewards amount (periodRewards) by the total supply of tokens (totalDeposits), as follows (simplified for explanatory purpose):

```
rewardPerToken = periodRewards.mul(1e18).div(totalDeposits());
```

Later, RewardEthToken.rewardOf() computes curReward, the amount of rewards to be distributed to an individual user, by multiplying the balance of the user (deposit) to rewardPerToken computed before, as follows (simplified for explanatory purpose):

```
curReward = deposit.mul(rewardPerToken).div(1e18);
```

When the new rewards amount is negative, i.e., periodRewards < 0, the rounding-towards-zero division could cause the sum of per-user reward reductions to be less than the total reward reductions (i.e., penalties), which means that the total supply of rwETH tokens could be bigger than the actual amount of rewards awarded to the validator pool. Although each rounding error (for each user and each reward distribution) is small, they could be accumulated over a long period, being non-negligible.

For example, suppose periodRewards is -1 ether, and totalDeposits is 30 ether.

- When there are 3 users, each staking 10 ether: the sum of the users’ curReward is 10 wei less than periodRewards. In this case, the major factor is the inaccuracy of the rewardPerToken computation.
- When there are 300 users, each staking 0.1 ether: the sum of the users’ curReward is 100 wei less than periodRewards. In this case, the major factor is the inaccuracy of the curReward computation.

### Recommendation

Use the flooring division instead of the rounding-towards-zero division.

Note that Solidity does not natively support the flooring division, but it can be implemented using the rounding-towards-zero division. The idea is to subtract 1 from the result, if the result is negative and non-exact, that is, if the signs of the arguments are different, and the numerator is not exactly divided by the denominator. Below is an example implementation, following the Java Math.floorDiv() implementation. (Note that the division-by-zero and signed-division-overflow are handled by the SignedSafeMath.div() function, which can be inlined to further save gas.)

```
function floorDiv(int256 a, int256 b) returns (int256) {
    int256 r = SignedSafeMath.div(a, b);
    if ((a ^ b) < 0 && (r * b != a)) r--;
    return r;
}
```

### Status

The negative rewards distribution feature has been removed in the latest version, and thus this issue is no longer applicable.

## B02: Precision losses in rewards distribution computation

Precision losses in the computation of individual rewards distributions can lead to distributing less amount of rewards to users. Specifically, for any user who staked *x* ether (i.e., *x* * 10^18 wei) of ETH or LP tokens, each reward distribution (i.e., each RewardEthToken.updateRewardCheckpoint() or StakedTokens._withdrawRewards() call) could result in distributing at worst *x* wei less than the exact amount of rwETH tokens.

**Details**

Let *R* be the total amount of rewards to be distributed, $D \geq 0$ be the total amount of user stakes, and $d \geq 0$ denotes the amount of stakes for a given user. Then, the best (fixed-point) approximation of the individual reward for the user is:

$$X = floor(R * d / D)$$

In the code base, however, the individual reward distribution is computed over two separate function calls, and the result is essentially represented as follows:

$$Y = floor(floor(R * 10^n / D) * d / 10^n)$$

Now, we have the following:

$$\text{If } d < 10^n, \text{ then } X - 1 \leq Y \leq X$$

That is, if the max amount of per-user stakes is representable within the given precision (i.e., *n* decimal digits), then the actual computation of rewards distributions is guaranteed to be equal or only 1 wei less than the optimal value. (For an example of $Y = X - 1$, if $R = 98$, $d = 800$, $D = 900$, and $n = 3$, then $X = 87$ but $Y = 86$.)

The proof is as follows. Let *Z* be $floor(R * 10^n / D)$. By the definition of the floor operation, we have:

- $X \leq R * d / D < X + 1$
- $Z \leq R * 10^n / D < Z + 1$
- $Y \leq Z * d / 10^n < Y + 1$

Then we conclude as follows:

- First, $Y \leq Z * d / 10^n \leq R * d / D < X + 1$. Thus $Y \leq X$, since both *X* and *Y* are an integer.

- Next, *Y > Z * d / 10^n - 1 > (R * 10^n / D - 1) * d / 10^n - 1 = R * d / D - d / 10^n - 1 > R * d / D - 2 ≥ X - 2*. Thus, *Y ≥ X - 1*, since both *X* and *Y* are an integer.

Note that the key factor in the proof is "*d / 10^n < 1*". Otherwise, *Y* could be at worst (*d / 10^n*) less than *X*. For example, if *n* = 18 (as in the current code base), *d* = 8 ether, *D* = 9 ether, and *R* = 0.98 ether, then the individual rewards for the user *Y* = 871111111111111104, which is 7 wei less than the optimal amount of reward *X* = 871111111111111111.

### Recommendation for RewardEthToken

Use a higher precision in the computation of rewardPerToken and curReward.

Specifically, it is recommended to use *n* = 27 instead of *n* = 18, because the total supply of ETH is less than 10^27 wei, and thus so is *d*. Also, note that it will not lead to the multiplication overflow, because "|*R*| * *10^n*" is less than 10^54, and "*floor(|R| * 10^n / D) * d*" is less than 10^63 assuming *D* is at least 10^18 (1 ether), where 2^255 > 10^76.

### Recommendation for StakedTokens

Unlike RewardEthToken, it is not straightforward to fix the precision losses in StakedTokens, because the total supply and the decimals of LP tokens could be arbitrarily different from each other. Thus, the optimal precision that minimizes the precision errors without causing potential multiplication overflows would be different for each LP token.

For the short term, it is recommended to evaluate potential LP tokens to be supported, identify their total supply, and determine the best possible precision that works for all the potential LP tokens. Then, update StakedTokens to use the determined precision uniformly.

For the long term, it is recommended to upgrade StakedTokens to employ different precisions for different LP tokens.

### Status

The StakeWise team acknowledged the recommendations.

## B03: StakedTokens.withdrawTokens() does not update token rewards when token is disabled

StakedTokens.withdrawTokens() skips updateTokenRewards(_token) when token is disabled. It makes the user receive less amount when they claim their rewards, even if the token keeps getting rewards even when it is disabled.

For example, suppose (only) two users *A* and *B* stake the same amount of *X* token, and later, *X* is disabled on January 1st. Suppose the user *A* withdraws all his tokens on February 1st, and later, *X* is enabled on March 1st. Suppose that the token *X* gets *N* rewards for January, and another *N* rewards for February. Then, even if the user *A*'s tokens were staked during January, he doesn't get his share of the rewards, *N/2*, when he claims. Instead, the user *B* will get the whole amount of rewards for the two months, *2N*, once *X* is enabled again.

### Recommendation

Do not skip updateTokenRewards(_token) even if _token is disabled.

### Status

Fixed as suggested in the latest version.

## B04: Permission for RewardEthToken.updateRewardCheckpoint()

While only the StakedEthToken contract is allowed to execute RewardEthToken.updateRewardCheckpoint(), the same effect can be made by the self `RewardEthToken.transfer()` call. In other words, `RewardEthToken.updateRewardCheckpoint(x)` and `RewardEthToken.transfer(x, x, 0)` have the same effect (as long as `rewardOf(x) >= 0`).

### Recommendation

Remove the non-effective permission requirement that is not necessary.

### Status

Fixed as suggested in the latest version.

## B05: No service fees are refunded in penalty distribution

While the maintainers charge a service fee for each reward distribution, they do not return (a portion of) the fees when penalties are imposed to users even if it is the maintainers' fault, which is unfair from users' perspective.

### Scenario

Consider the following two cases:

- Case 1: The validator pool gets 1 ETH rewards in the first month, and another 1 ETH rewards in the second month.
- Case 2: The validator pool gets 3 ETH rewards in the first month, and -1 ETH penalties in the second month.

Suppose the maintainer fee is 0.1%. The total amount of accumulated rewards is the same for the above two cases, which is 2 ETH, but the maintainers will end up charging more service fees in the second case (i.e., 0.002 ETH for the first case, and 0.003 ETH for the second case), while they should be the same for both cases to be fair.

### Recommendation

Implement the maintainers to receive a "negative" service fee when the validator pool is penalized.

### Status

The instant penalty distribution feature has been removed in the latest version, and thus this is no longer applicable.

## B06: Gas optimization suggestion

If stETH or rwETH tokens are provided as liquidity to Uniswap, the _transfer() function will be extremely frequently executed, and it would be desired to optimize the gas cost for that. The dominating factor is the storage updates and reads.

StakedEthToken._transfer() requires two checkpoints[] updates and two deposits[] updates, where each checkpoints[] update involves two storage updates, thus total 6 storage updates for each StakedEthToken._transfer() call, which spends at least 30,000 gas (= 6 * 5,000) for the storage updates. Similarly, RewardEthToken._transfer() requires two checkpoints[] updates, which involves 4 storage updates, thus spends at least 20,000 gas (= 4 * 5,000). Here, for each transfer, 10,000 gas (i.e., two storage updates) can be saved by packing the Checkpoint struct into a single word, that is, having rewardPerToken and reward to be of int128. Since int128 can represent up to 38 decimal digits, it seems possible to refactor the code to use the smaller typed integer (although it should be carefully done to make sure no precision issues are introduced.) Note that the 10,000 gas saving corresponds to ~10% of each transfer function's gas cost.

Regarding the storage read, rewardOf() can be gas-optimized to not read deposits[account] when rewardPerToken.sub(cp.rewardPerToken) is zero, which can save 800 gas. Note that the case rewardPerToken.sub(cp.rewardPerToken) == 0 will happen very frequently, because the trade volume of the Uniswap pairs is very high. Also note that rewardOf() is executed three times per StakedEthToken._transfer(), and twice per RewardEthToken._transfer(), thus 2,400 gas and 1,600 gas can be saved respectively, by this optimization.

On the other hand, rewardPerToken is read 5 times per StakedEthToken._transfer(), and 3 times per RewardEthToken._transfer(), which in theory, could be optimized to save 3,200 gas and 1,600 gas, respectively, but such a refactoring would significantly deteriorate the code readability and simplicity, so is not recommended.

**Status**

Fixed as suggested in the latest version.

**Updated (Jan 12)**: Further gas optimization suggestion was made and adopted in the latest version.

## B07: Access control analysis

Each of the contracts (i.e., Solos, Pool, StakedEthToken, RewardEthToken, StakedTokens, BalanceReporters, and Validators) can have their own admins, and can be only independently paused by their own pausers who may not be an admin.

When StakedEthToken is paused but Pool is *not* paused, new stETH tokens can continue to be minted.

When RewardEthToken is paused but BalanceReporters is *not* paused, the total rewards as well as individual rewards can continue to be updated, meaning that new rwETH tokens can continue to be minted.

When all contracts are paused, users can continue to approve spending of their stETH/rwETH tokens via `approve()`, `increaseAllowance()`, or `permit()`.

When Solos is paused, users can continue to cancel their deposits.

**Status**

The StakeWise team acknowledged, and also said:

> *"It is intentional to allow users to cancel their deposits even when Solos is paused. Otherwise, it causes the user to lose custody over the funds if we pause the contract and he won't be able to cancel his deposit."*

## B08: Non-systematic uses of nonReentrant modifier

`Solos.addDeposit()` is not annotated with `nonReentrant`. This allows the caller of `cancelDeposit()` to re-enter Solos via `addDeposit()`, and so does other external contracts called by other contracts (i.e., the token contracts of StakedTokens, and the Uniswap pairs of BalanceReporters). Although we could not find any exploits of this, it could be considered to add `nonReentrant` to `addDeposit()` as done with other public state-changing functions in other contracts, to be more systematic.

Moreover, each contract has their own `nonReentrant` modifier that is not shared with other contracts. This means that external contracts could re-enter to other contracts within the same transaction. Although we could not find any exploits of this, it could be considered to have a single global `nonReentrant` modifier to lock all contracts from any external contract calls.

However, any of these changes will cause additional gas usage, and the trade-offs need to be carefully evaluated before adopting the changes.

**Status**

Acknowledged by the StakeWise team.

## B09: Potential arithmetic overflows

In Solos, “`block.timestamp + cancelLockDuration`” could overflow, in case that `cancelLockDuration` is set to a very large value by mistake.

In BalanceReporters, “`getRoleMemberCount(REPORTER_ROLE) * 2`” could overflow, in case that the library AccessControlUpgradeable contract somehow misbehaves due to hidden bugs.

It is considered to be a better practice to systematically use SafeMath for every arithmetic.

### Status

Fixed in PR 63.

# Bytecode Test Coverage Analysis

The bytecode-level test coverage analysis powered by Firefly revealed missing test scenarios. Specifically:

- Certain (auto-generated) getter functions are not tested.
- Certain functions inherited from the dependencies are not tested.
- Negative tests for certain `require()` assertions were missed, specifically:
  - No tests for the case of "`msg.sender != stakedTokens`" in claimRewards().
- No tests for executing the external call of UniswapV2Pair.sync() inside voteForTotalRewards().
- No tests for the case when no rewards are newly generated in updateTotalRewards().
- No tests for rewardRateOf().

The coverage report for the later version 11366ec5c772f098b70cbb966e9ae5fe50aa2801 is available here.

### Recommendation

Add more tests to cover missing cases.

### Status

Acknowledged by the StakeWise team.

# Appendix: Contract Diagram