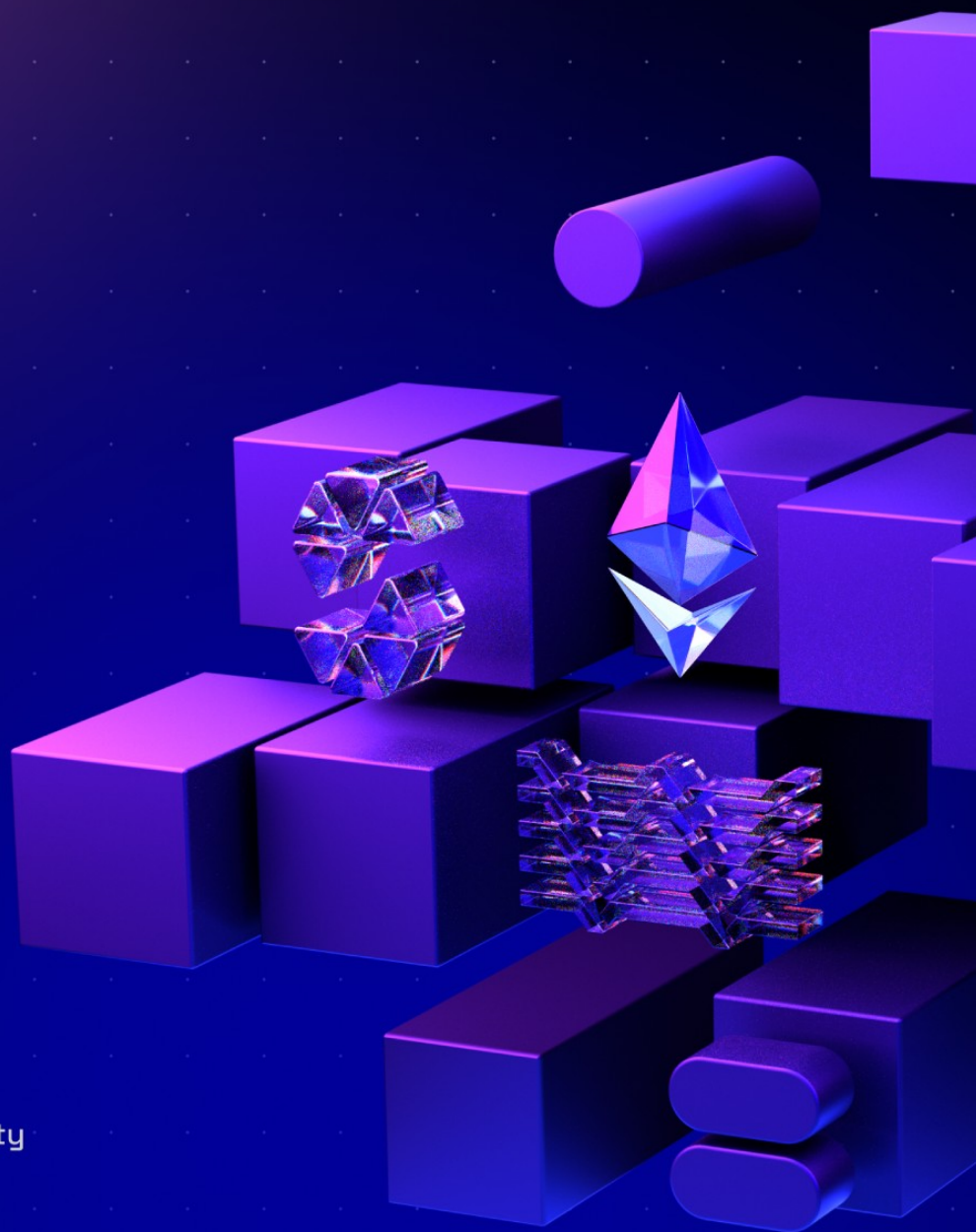
ackee
blockchain security

Let's get HAI

New core features

4.4.2025



Ackee Blockchain Security

Contents

1. Document Revisions

1.0-draft	Draft Report	14.03.2025
1.0	Final Report	25.03.2025
1.1	Final Report	04.04.2025

2. Overview

This document presents our findings in reviewed contracts.

2.1. Ackee Blockchain Security

Ackee Blockchain Security is an in-house team of security researchers performing security audits focusing on manual code reviews with extensive fuzz testing for Ethereum and Solana. Ackee is trusted by top-tier organizations in web3, securing protocols including Lido, Safe, and Axelar.

We develop open-source security and developer tooling Wake for Ethereum and Trident for Solana, supported by grants from Coinbase and the Solana Foundation. Wake and Trident help auditors in the manual review process to discover hardly recognizable edge-case vulnerabilities.

Our team teaches about blockchain security at the Czech Technical University in Prague, led by our co-founder and CEO, Josef Gattermayer, Ph.D. As the official educational partners of the Solana Foundation, we run the School of Solana and the Solana Auditors Bootcamp.

Ackee's mission is to build a stronger blockchain community by sharing our knowledge.

Ackee Blockchain a.s.
Rohanske nabrezi 717/4
186 00 Prague, Czech Republic
https://ackee.xyz
hello@ackee.xyz

2.2. Audit Methodology

1. **Verification of technical specification**
   The audit scope is confirmed with the client, and auditors are onboarded to the project. Provided documentation is reviewed and compared to the audited system.

2. **Tool-based analysis**
   A deep check with Solidity static analysis tool Wake in companion with Solidity (Wake) extension is performed, flagging potential vulnerabilities for further analysis early in the process.

3. **Manual code review**
   Auditors manually check the code line by line, identifying vulnerabilities and code quality issues. The main focus is on recognizing potential edge cases and project-specific risks.

4. **Local deployment and hacking**
   Contracts are deployed in a local Wake environment, where targeted attempts to exploit vulnerabilities are made. The contracts' resilience against various attack vectors is evaluated.

5. **Unit and fuzz testing**
   Unit tests are run to verify expected system behavior. Additional unit or fuzz tests may be written using Wake framework if any coverage gaps are identified. The goal is to verify the system's stability under real-world conditions and ensure robustness against both expected and unexpected inputs.

2.3. Finding Classification

A *Severity* rating of each finding is determined as a synthesis of two sub-ratings: *Impact* and *Likelihood*. It ranges from *Informational* to *Critical*.

If we have found a scenario in which an issue is exploitable, it will be assigned an impact rating of *High*, *Medium*, or *Low*, based on the direness of the consequences it has on the system. If we haven't found a way, or the issue is only exploitable given a change in *configuration* (system settings or parameters, such as deployment scripts, compiler configurations, using multi-signature wallets for owners, etc.) or given a change in the codebase, then it will be assigned an impact rating of *Warning* or *Info*.

Low to *High* impact issues also have a *Likelihood*, which measures the probability of exploitability during runtime.

The full definitions are as follows:

Severity

		Likelihood			
		High	**Medium**	**Low**	**N/A**
Impact	**High**	Critical	High	Medium	-
	Medium	High	Medium	Low	-
	Low	Medium	Low	Low	-
	Warning	-	-	-	Warning
	Info	-	-	-	Info

Table 1. Severity of findings

Impact

- **High** - Code that activates the issue will lead to undefined or catastrophic consequences for the system.
- **Medium** - Code that activates the issue will result in consequences of serious substance.
- **Low** - Code that activates the issue will have outcomes on the system that are either recoverable or don't jeopardize its regular functioning.
- **Warning** - The issue cannot be exploited given the current code and/or *configuration*, but could be a security vulnerability if these were to change slightly. If we haven't found a way to exploit the issue given the time constraints, it might be marked as a "Warning" or higher, based on our best estimate of whether it is currently exploitable.
- **Info** - The issue is on the borderline between code quality and security. Examples include insufficient logging for critical operations. Another example is that the issue would be security-related if code or *configuration* was to change.

Likelihood

- **High** - The issue is exploitable by virtually anyone under virtually any circumstance.
- **Medium** - Exploiting the issue currently requires non-trivial preconditions.
- **Low** - Exploiting the issue requires strict preconditions.

2.4. Review Team

The following table lists all contributors to this report. For authors of the specific revision, see the “Revision team” section in the respective “Report revision” chapter.

Member’s Name	Position
Jan Převrátil	Lead Auditor
Dmytro Khimchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

2.5. Disclaimer

We’ve put our best effort to find all vulnerabilities in the system, however our findings shouldn’t be considered as a complete list of all existing issues. The statements made in this document should not be interpreted as investment or legal advice, nor should its authors be held accountable for decisions made based on them.

3. Executive Summary

The HAI protocol serves as a framework for minting HAI stablecoins against various collateral types. The protocol's governance token, KITE, enables token holders to earn rewards through staking. The HAI protocol integrates with Velodrome, offering wrapped haiVELO tokens that are composable and generate veVELO rewards.

Revision 1.0

Let's get HAI engaged Ackee Blockchain Security to perform a security review of the Let's get HAI protocol with a total time donation of 10 engineering days in a period between March 4 and March 14, 2025, with Jan Převrátil as the lead auditor. The audit was initially set on the commit `387183d`[1].

On March 5, 2025, Let's get HAI requested to extend the audit scope with selected contracts from the `src/contracts/oracles` directory and to perform the review on commit `be45f6c`[2].

The kickoff call was held on March 5, 2025, and Let's get HAI provided additional post-meeting notes that were helpful during the review. Let's get HAI was responsive during the review and provided clarifications to the review team's questions.

The audit scope was set to the newly added features:

1. KITE staking / rewards
2. Daily rewards distribution via Merkle trees
3. HAI VELO wrapper
4. Beefy / Yearn Velo Vault Oracles

Since the scope was limited only to the newly added features, the security of the Let's get HAI protocol as a whole was not checked.

The reviewed features correspond to the following contracts:

- `src/contracts/factories/FactoryChild.sol`
- `src/contracts/factories/RewardPoolChild.sol`
- `src/contracts/factories/RewardPoolFactory.sol`
- `src/contracts/oracles/AbstractVeloVaultRelayer.sol`
- `src/contracts/oracles/BeefyVeloVaultRelayer.sol`
- `src/contracts/oracles/YearnVeloVaultRelayer.sol`
- `src/contracts/tokens/RewardDistributor.sol`
- `src/contracts/tokens/RewardPool.sol`
- `src/contracts/tokens/StakingManager.sol`
- `src/contracts/tokens/StakingToken.sol`
- `src/contracts/tokens/WrappedToken.sol`
- `src/contracts/utils/Authorizable.sol`
- `src/contracts/utils/Modifiable.sol`
- `src/libraries/Assertions.sol`
- `src/libraries/Encoding.sol`

We began our review using static analysis tools, including Wake. We then took a deep dive into the logic of the contracts. For unit testing and writing exploit scenarios, we involved the Wake testing framework. During the review, we paid special attention to:

- ensuring the arithmetic of the system is correct;
- validating accurate reward calculations;
- verifying fair reward distribution among stakers;
- preventing rewards from becoming stuck in contracts;

- detecting possible reentrancies in the code;
- ensuring claiming of rewards cannot be exploited;
- ensuring the system is not vulnerable to front-running;
- ensuring access controls are not too relaxed or too strict; and
- looking for common issues such as data validation.

Our review resulted in 21 findings, ranging from Info to Critical severity. The most critical findings were:

- C1: Users could claim rewards multiple times due to the `rewardIntegralFor` state variable not updating after reward transfers;
- H1: Malicious users could front-run merkle tree updates in the `RewardDistributor` contract to double-claim rewards; and
- H2: Incorrect interface implementation caused the `getResultWithValidity` function in `BeefyVeloVaultRelayer` to consistently revert.

The team responded promptly to these findings and collaborated effectively on developing solutions.

Ackee Blockchain Security recommends Let's get HAI:

- update state variables appropriately and ensure they are updated;
- use correct interfaces for interacting with third-party contracts;
- emit events for critical functions;
- validate input parameters to prevent unspecified errors;
- read and review the complete audit report; and
- address all identified issues.

See Report Revision 1.0 for the system overview and trust model.

Revision 1.1

Let's get HAI engaged Ackee Blockchain Security to review the fixes.

The review was performed on April 3, 2025 on the commit `11c4b4d`[3].

Let's get HAI team created a pull request PR #109 with the fixes and provided comments to most of the acknowledged findings.

From the reported 21 findings:

- 10 issues were fixed;
- 6 issues were acknowledged;
- 3 minor issues were fixed partially (I3, I4, I8); and
- 2 minor issues remained unresolved (W4, I9).

The fix of the L2 issue introduced a new finding M2.

[1] full commit hash: `387183d5949cd982222ec86fab49355eba813be2`, link to commit

[2] full commit hash: `be45f6c67d2b8557750e4a2c5e661c58f0f03dfb`, link to commit

[3] full commit hash: `11c4b4d8792b2e5c197606dd6fde6d996ce064ca`, link to commit

4. Findings Summary

The following section summarizes findings we identified during our review. Unless overridden for purposes of readability, each finding contains:

- *Description*
- *Exploit scenario* (if severity is low or higher)
- *Recommendation*
- *Fix* (if applicable).

Summary of findings:

Critical	High	Medium	Low	Warning	Info	Total
1	2	2	3	4	10	22

Table 2. Findings Count by Severity

Findings in detail:

Finding title	Severity	Reported	Status
C1: User can claim all rewards, that `StakingManager` received from `RewardPool`	Critical	1.0	Fixed
H1: Front-running `RewardDistributor.updateMerkleRoots` allows double claim	High	1.0	Acknowledged
H2: External interface `IPessimisticVeloLpOracle` is outdated	High	1.0	Acknowledged
M1: `RewardPool._totalStaked` variable updates incorrectly	Medium	1.0	Fixed

Finding title	Severity	Reported	Status
L1: Queued rewards in RewardPool can become stuck	Low	1.0	Acknowledged
L2: A user can received rewards after withdrawal process has been initiated	Low	1.0	Fixed
L3: `StakingToken.burnFrom` function does not emit `StakingTokenBurn` event	Low	1.0	Fixed
W1: Reward calculation state variables not updated in critical functions	Warning	1.0	Fixed
W2: Potential underflow in math operations leads to unspecified errors	Warning	1.0	Fixed
W3: Oracle vault relayers lack non-zero price validation	Warning	1.0	Fixed
W4: Unchecked return value of `ERC20.transfer` in `RewardDistributor`	Warning	1.0	Reported
I1: Missing event emission for reward pool token staking	Info	1.0	Fixed
I2: `RewardDistributor._claim` leaf is double hashed	Info	1.0	Acknowledged
I3: Magic numbers	Info	1.0	Partially fixed

Finding title	Severity	Reported	Status
I4: Typos and missing documentation	Info	1.0	Partially fixed
I5: Code style inconsistencies	Info	1.0	Fixed
I6: Optimization of function `_getPriceValue`	Info	1.0	Acknowledged
I7: Unused errors	Info	1.0	Fixed
I8: Unused `using-for` directives	Info	1.0	Partially fixed
I9: Unused functions	Info	1.0	Reported
I10: Variables should be immutable	Info	1.0	Acknowledged
M2: `stakeToken` can be transferred to any other address while it is still assumed staked	Medium	1.1	Reported

Table 3. Table of Findings

Report Revision 1.0

Revision Team

Member's Name	Position
Jan Převrátil	Lead Auditor
Dmytro Khimchenko	Auditor
Josef Gattermayer, Ph.D.	Audit Supervisor

System Overview

The HAI protocol is an enhanced GEB fork deployed on Optimism that functions as a framework for minting HAI stablecoins using various collateral types. Compared to the GEB protocol, HAI introduces improvements including advanced system parameter controls and enhanced deployment through a factory approach. Users can utilize the minted HAI stablecoin as collateral in other DeFi protocols.

KITE, the protocol's governance token, enables holders to earn rewards through staking. The HAI protocol integrates with Velodrome, providing wrapped haiVELO tokens that remain compatible with other protocols while generating veVELO rewards.

Trust Model

The reward distribution part of the protocol is designed as permissioned.

In the `RewardDistributor` contract, authorized addresses must provide rewards and update merkle roots for their distribution. The claiming process can be paused by authorized addresses at any time, and tokens can be withdrawn from the contract using the `emergencyWithdraw` function.

The `StakingManager` and `StakingToken` contracts are not pausable by

authorized addresses; however, they provide the `emergencyWithdraw` function that allows authorized addresses to withdraw any amount of tokens at any time.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

C1: User can claim all rewards, that `StakingManager` received from `RewardPool`

Critical severity issue

Impact:	High	Likelihood:	High
Target:	StakingManager	Type:	Logic error

Description

The `StakingManager` contract allows users to claim rewards from `RewardPool`. For reward accounting, it uses the `RewardType.rewardIntegralFor` mapping to store the reward integral for each user. This mapping is updated when users check their rewards by calling the `StakingManager.earned` function without claiming them.

However, when users claim and transfer rewards to their address using `StakingManager.getReward`, the mapping remains unchanged. This oversight allows users to call `StakingManager.getReward` multiple times, claiming all rewards that `StakingManager` received from `RewardPool`.

Exploit scenario

1. Alice, a user, stakes 1000 USDC into the `StakingManager`.
2. Bob, another user, stakes 1000 USDC into the `StakingManager`.
3. Carl, a malicious user, stakes 1 USDC into the `StakingManager`.
4. After half of the reward period passes, Carl calls `StakingManager.getReward` function multiple times and receives all rewards from `StakingManager`, depriving Alice and Bob of their rightful rewards.

Recommendation

Update the `RewardType.rewardIntegralFor` mapping in `StakingManager` after

each `StakingManager.getReward` function call.

Fix 1.1

The issue was fixed in the commit `409cb53`[1] by updating the `RewardType.rewardIntegralFor` mapping in `StakingManager` after each `StakingManager.getReward` function call.

Go back to Findings Summary

H1: Front-running `RewardDistributor.updateMerkleRoots` allows double claim

High severity issue

Impact:	High	Likelihood:	Medium
Target:	RewardDistributor	Type:	Front-running

Description

The `RewardDistributor.updateMerkleRoots` function is vulnerable to a front-running attack. The `RewardDistributor` contract maintains only one active Merkle root at a time, which represents an accumulation of all unclaimed user rewards. A malicious user can front-run the root update by claiming rewards from the old Merkle root and subsequently claim from the new root, enabling a double claim. If the contract's token balance is limited to exact reward values, this attack can prevent other users from claiming their rewards due to insufficient token balances.

Exploit scenario

1. Alice, a bad actor, accumulates a large quantity of unclaimed rewards over a long period.
2. Bob, the protocol owner with `rootSetter` authorization, calculates the new cumulative Merkle root.
3. Bob submits a transaction with `updateMerkleRoots` to the mempool.
4. Alice monitors the mempool and front-runs Bob's update with a `multiClaim` transaction, claiming all rewards from the old Merkle root.
5. Bob's transaction with the updated Merkle root is processed.
6. Alice submits another `multiClaim` transaction to claim the rewards again from the updated Merkle root.

Recommendation

Update Merkle roots using the following procedure:

1. Call `RewardDistributor.pause()`.
2. Wait for the pause transaction to be finalized.
3. Calculate the rewards from the last update to the pausing block.
4. Call `RewardDistributor.updateMerkleRoots(…)` with the updated reward Merkle roots.
5. Wait for the update transaction to be finalized.
6. Call `RewardDistributor.unpause()`.
7. Wait for the unpause transaction to be finalized before completing the procedure.

Acknowledgment 1.1

The Let's get HAI team acknowledged the issue with the following comment:

> *This functionality (pause) already exists and is tested here: https://github.com/hai-on-op/core/blob/main/test/unit/RewardDistributor.t.sol#L303-L321*
>
> — Let's get HAI team

Reply from Ackee Blockchain Security team:

> *While the modifier and pausing functions are implemented and tested, they must be used correctly. The tests do not demonstrate the usage of pausing functions around the* `updateMerkleRoots` *function. Additionally, the NatSpec documentation does not mention the requirement to pause the contract before changing the Merkle roots.*

> *The pausing functions are primarily used for emergency situations and protocol upgrades. This raised our concern that the contract is not being paused correctly before changing the Merkle roots.*
>
> — Ackee Blockchain Security team

Go back to Findings Summary

H2: External interface `IPessimisticVeloLpOracle` is outdated

High severity issue

Impact:	Medium	Likelihood:	High
Target:	IPessimisticVeloLpOracle	Type:	Denial of service

Description

The external interface `IPessimisticVeloLpOracle.sol` is incompatible with its provided implementation at address 0xDA5aA25c4110E8AE7DaBAC15fC253B84b28fdC2A. The function `getCurrentPoolPrice` in the interface was renamed to `getCurrentPrice` in the external implementation. As a result, the function `_getPriceValue` reverts on every call. This affects the contracts `YearnVeloVaultRelayer` and `BeefyVeloVaultRelayer`, which expose this functionality through their external functions `getResultWithValidity` and `read`. These functions are called by protocol components outside the audit scope.

Exploit scenario

1. Alice, a protocol owner, deploys the protocol with the current interface.
2. Bob, a user, attempts to interact with protocol components that depend on the oracle.
3. All of Bob's transactions revert, making this protocol component unusable.

Recommendation

Update the interface `IPessimisticVeloLpOracle` to match the latest implementation.

Verify that all interfaces match their implementations before deployment.

Acknowledgment 1.1

The Let's get HAI team acknowledged the issue with the following comment:

> *This finding is a function of us accidentally sharing the old* `PessimisticVeloLpOracle` *contract address that used this outdated function. The current iteration of that contract uses the proper function name that corresponds to the interface.*
>
> — Let's get HAI team

Reply from Ackee Blockchain Security team:

> *The wrong address was mentioned in the notes shared with us after the kick-off meeting.*
>
> *Since the correct address must be passed to functions* `deployBeefyVeloVaultRelayer` *and* `deployYearnVeloVaultRelayer`*, which are not called in the deployment script, we evaluated that the provided address will be used for deployment either in the deployment script or manually.*
>
> — Ackee Blockchain Security team

Go back to Findings Summary

M1: `RewardPool._totalStaked` variable updates incorrectly

Medium severity issue

Impact:	Medium	Likelihood:	Medium
Target:	RewardPool.sol	Type:	Logic error

Description

The `RewardPool` contract maintains a `_totalStaked` variable to track the total amount of staked tokens in the `StakingManager` contract. This variable updates when `StakingManager` stakes or withdraws tokens.

Listing 1. Excerpt from RewardPool

```
100 function stake(uint256 _wad) external updateReward isAuthorized {
101   if (_wad == 0) revert RewardPool_StakeNullAmount();
102   _totalStaked += _wad;
103   emit RewardPoolStaked(msg.sender, _wad);
104 }
```

Listing 2. Excerpt from RewardPool

```
107 function increaseStake(uint256 _wad) external isAuthorized {
108   if (_wad == 0) revert RewardPool_IncreaseStakeNullAmount();
109   _totalStaked += _wad;
110   emit RewardPoolIncreaseStake(msg.sender, _wad);
111 }
```

Listing 3. Excerpt from RewardPool

```
114 function decreaseStake(uint256 _wad) external isAuthorized {
115   if (_wad == 0) revert RewardPool_DecreaseStakeNullAmount();
116   if (_wad > _totalStaked) revert RewardPool_InsufficientBalance();
117   _totalStaked -= _wad;
118   emit RewardPoolDecreaseStake(msg.sender, _wad);
```

```
119 }
```

The `StakingManager` contract can connect to multiple `RewardPool` contracts. When a new `RewardPool` contract connects to `StakingManager` after tokens are already staked, the `_totalStaked` variable in the new pool does not reflect the correct total staked amount.

Exploit scenario

First scenario - Withdrawal Reversion:

1. `Reward_Pool_1` connects to the `StakingManager` contract.
2. Alice, a user, stakes 100 KITE via `StakingManager`.
3. Bob, another user, stakes 100 KITE via `StakingManager`.
4. `Reward_Pool_1` correctly reflects `_totalStaked`.
5. `Reward_Pool_2` connects to the `StakingManager` contract.
6. Alice's withdrawal attempt reverts at line 117 due to incorrect `_totalStaked`:

Listing 4. Excerpt from RewardPool

```
114 function decreaseStake(uint256 _wad) external isAuthorized {
115   if (_wad == 0) revert RewardPool_DecreaseStakeNullAmount();
116   if (_wad > _totalStaked) revert RewardPool_InsufficientBalance();
117   _totalStaked -= _wad;
118   emit RewardPoolDecreaseStake(msg.sender, _wad);
119 }
```

Second scenario - Missing Rewards:

1. `Reward_Pool_1` connects to the `StakingManager` contract.
2. Alice stakes 100 KITE via `StakingManager`.
3. Bob stakes 100 KITE via `StakingManager`.
4. `Reward_Pool_1` correctly reflects `_totalStaked`.

5. `Reward_Pool_2` connects to the `StakingManager` contract.
6. Time passes without additional staking activity.
7. Alice and Bob receive zero rewards from `Reward_Pool_2` due to `_totalStaked` being zero, causing incorrect calculations in the `rewardPerToken` function.

Recommendation

Add a state variable in the `RewardPool` contract to store the `StakingToken` contract address and use `StakingToken.totalSupply` to obtain the correct `_totalStaked` value.

Fix 1.1

The issue was fixed in the commit `800abd6`[2] by adding the ability to deploy reward pool with non zero initial `_totalStaked` value.

Go back to Findings Summary

L1: Queued rewards in RewardPool can become stuck

Low severity issue

Impact:	Medium	Likelihood:	Low
Target:	RewardDistributor.sol	Type:	Logic error

Description

The `RewardPool` contract includes functionality to queue rewards for distribution via the `queueNewRewards` function. Rewards are queued when `_params.newRewardRatio` is less than or equal to the calculated `_queuedRatio`.

Listing 5. Excerpt from RewardPool

```
172 if (_queuedRatio < _params.newRewardRatio) {
173   notifyRewardAmount(_totalRewards);
174   queuedRewards = 0;
175 } else {
176   queuedRewards = _totalRewards;
177 }
```

These queued rewards are only added to the actual rewards distribution when the `queueNewRewards` function is called again to recalculate the rewards.

If all rewards are distributed and no subsequent call to `queueNewRewards` occurs, the queued rewards remain stuck in the `RewardPool` contract and are not included in the rewards distribution.

Exploit scenario

1. Alice, a user, stakes 1000 KITE.
2. `Reward_Pool_1` is configured and connected to the `StakingManager` contract.

3. An authorized user adds rewards to `Reward_Pool_1` via `queueNewRewards`, but these rewards are queued due to ratio conditions.
4. All existing rewards are distributed, but the queued rewards are not automatically added to the distribution.
5. Without subsequent calls to `queueNewRewards`, the queued rewards remain permanently stuck in the `RewardPool` contract.

Recommendation

Implement an automatic mechanism to add queued rewards to the `RewardPool` distribution when existing rewards are depleted.

Acknowledgment 1.1

The Let's get HAI team acknowledged the issue with the following comment:

> *This is intentional and the reason the notifyRewardAmount function exists, so the DAO can trigger when the rewards are released / start.*
>
> — Let's get HAI team

Go back to Findings Summary

L2: A user can received rewards after withdrawal process has been initiated

Low severity issue

Impact:	Low	Likelihood:	Medium
Target:	StakingManager.sol	Type:	Logic error

Description

The token withdrawal process consists of the following steps:

1. User calls `StakingManager.initiateWithdrawal` function;
2. User waits for the `_cooldown` period; and
3. User withdraws tokens using `StakingManager.withdraw` function.

However, users continue to receive rewards even after their tokens are eligible for withdrawal. This allows users to initiate withdrawal immediately after staking while still collecting rewards. Once the `_cooldown` period expires, users can withdraw their funds at any time while continuing to receive rewards.

Exploit scenario

1. Alice, a user, stakes 100 KITE tokens.
2. Alice immediately calls `StakingManager.initiateWithdrawal`.
3. After the `_cooldown` period expires, Alice maintains the ability to withdraw her tokens at any time while continuing to receive and collect rewards, undermining the intended staking mechanism.

Recommendation

Modify the contract to stop reward accrual when a user initiates the

withdrawal process.

Fix 1.1

The issue was fixed in the commit `4a1663d`[3] by using the `stakedBalances` mapping to track the amount of tokens that are not withdrawn or not in the queue for withdrawal.

In addition to the fix, the Let's get HAI team provided the following comment:

> *We have fixed this but are okay with the allowing the behavior identified regardless.*
>
> — Let's get HAI team

Go back to Findings Summary

L3: `StakingToken.burnFrom` function does not emit `StakingTokenBurn` event

Low severity issue

Impact:	Low	Likelihood:	Low
Target:	StakingToken.sol	Type:	Logging

Description

The `StakingToken.burn` function emits the `StakingTokenBurn` event, while the `StakingToken.burnFrom` function does not.

Listing 6. Excerpt from StakingToken

```
70 /// @inheritdoc IStakingToken
71 function burn(uint256 _wad) public override(ERC20Burnable, IStakingToken) {
72   _burn(msg.sender, _wad);
73   emit StakingTokenBurn(msg.sender, _wad);
74 }
75
76 /// @inheritdoc IStakingToken
77 function burnFrom(address _account, uint256 _wad) public
   override(ERC20Burnable, IStakingToken) {
78   _spendAllowance(_account, msg.sender, _wad);
79   _burn(_account, _wad);
80 }
```

This inconsistency can lead to inaccurate external tracking of burned `StakingToken` tokens and total supply.

Exploit scenario

1. Alice, an approved spender, burns a significant amount of tokens on behalf of a token holder using the `burnFrom` function.
2. Bob, an external observer, monitors incorrect values of burned tokens and total supply.

3. Bob loses trust in the protocol upon discovering the discrepancy.

Recommendation

Add the `StakingTokenBurn` event emission to the `burnFrom` function to maintain consistent event logging behavior.

Fix 1.1

The issue was fixed in the commit `af61cdf`[4] by adding the `StakingTokenBurn` event emission to the `burnFrom` function.

Go back to Findings Summary

W1: Reward calculation state variables not updated in critical functions

Impact:	Warning	Likelihood:	N/A
Target:	RewardPool.sol	Type:	Logic error

Description

The following reward-related state variables are not updated during critical function calls:

- `rewardPerTokenStored`
- `lastUpdateTime`
- `rewards`
- `rewardsPerTokenPaid`

These variables are responsible for reward calculations but remain unchanged in the following functions:

Listing 7. Excerpt from RewardPool

```
107 function increaseStake(uint256 _wad) external isAuthorized {
108   if (_wad == 0) revert RewardPool_IncreaseStakeNullAmount();
109   _totalStaked += _wad;
110   emit RewardPoolIncreaseStake(msg.sender, _wad);
111 }
112
113 /// @inheritdoc IRewardPool
114 function decreaseStake(uint256 _wad) external isAuthorized {
115   if (_wad == 0) revert RewardPool_DecreaseStakeNullAmount();
116   if (_wad > _totalStaked) revert RewardPool_InsufficientBalance();
117   _totalStaked -= _wad;
118   emit RewardPoolDecreaseStake(msg.sender, _wad);
119 }
```

Recommendation

Apply the `updateReward` modifier to all functions that affect reward calculations.

Fix 1.1

The issue was fixed in the commit `8e9c0cd`[5] by applying the `updateReward` modifier to all mentioned functions that affect reward calculations.

Go back to Findings Summary

W2: Potential underflow in math operations leads to unspecified errors

Impact:	Warning	Likelihood:	N/A
Target:	RewardPool.sol, StakingManager.sol	Type:	Data validation

Description

The `RewardPool` and `StakingManager` contracts contain math operations that can potentially underflow, resulting in unspecified errors:

- In `RewardPool`, if `_params.duration` decreases and the sum of `block.timestamp` and `_params.duration` becomes less than `periodFinish`, an underflow occurs:

Listing 8. Excerpt from RewardPool

```
168 uint256 _elapsedTime = block.timestamp - (periodFinish - _params.duration);
```

- In `StakingManager`, if a user attempts to initiate a withdrawal with an `_amount` greater than their balance, an underflow occurs:

Listing 9. Excerpt from StakingManager

```
143 stakedBalances[msg.sender] -= _wad;
```

Recommendation

Add input validation checks before performing math operations:

- In `RewardPool`: Verify that `block.timestamp + _params.duration >= periodFinish`
- In `StakingManager`: Validate that `_amount` does not exceed the user's

balance

Fix 1.1

The issue was fixed in the commit `d443c20`[6] by adding input validation checks before performing math operations.

Go back to Findings Summary

W3: Oracle vault relayers lack non-zero price validation

Impact:	Warning	Likelihood:	N/A
Target:	YearnVeloVaultRelayer.sol, BeefyVeloVaultRelayer.sol	Type:	Data validation

Description

The `_getPriceValue` function in the `BeefyVeloVaultRelayer` and `YearnVeloVaultRelayer` contracts lacks validation of the result against zero. The values `_veloLpBalance` (line 54) and `_veloLpPrice` (line 57) are obtained from external contracts that implement non-zero checks. However, if the implementation of these external contracts changes, a zero value could be returned due to the missing non-zero validation. Such an occurrence would have catastrophic consequences for the protocol.

Listing 10. Excerpt from YearnVeloVaultRelayer

```
53 // # of velo LP tokens in 1 yvToken
54 uint256 _veloLpBalance = _yvTokenBalance.wmul(yearnVault.pricePerShare());
55
56 // price of 1 velo LP token in chainlink price decimals (8)
57 uint256 _veloLpPrice = veloLpOracle.getCurrentPoolPrice(address(veloPool));
58
59 return (_veloLpBalance * _veloLpPrice) / 1e8;
```

Listing 11. Excerpt from BeefyVeloVaultRelayer

```
53 // # of velo LP tokens in 1 mooToken
54 uint256 _veloLpBalance =
   _mooTokenBalance.wmul(beefyVault.getPricePerFullShare());
55
56 // price of 1 velo LP token in chainlink price decimals (8)
57 uint256 _veloLpPrice = veloLpOracle.getCurrentPoolPrice(address(veloPool));
58
59 return (_veloLpBalance * _veloLpPrice) / 1e8;
```

Recommendation

Add a non-zero validation check for the calculated price before returning the value.

Fix 1.1

The issue was fixed in the commit `7422bec`[7] by adding a non-zero validation check for the calculated price before returning the value.

Go back to Findings Summary

W4: Unchecked return value of `ERC20.transfer` in `RewardDistributor`

Impact:	Warning	Likelihood:	N/A
Target:	RewardDistributor.sol	Type:	Data validation

Description

The return value of the `ERC20.transfer` function is unchecked in the `_claim` and `emergencyWithdraw` functions of the `RewardDistributor` contract. While the standard ERC-20 implementation should revert on insufficient balance and return `true` on success, some implementations may return `false` instead. Due to the ignored return value, the withdrawal is recorded as claimed and the claim event is emitted even when the transfer fails. In such cases, the claim cannot be repeated.

Recommendation

Add a check for the return value of the `transfer` function and revert the transaction if it returns `false`.

Update 1.1

The Let's get HAI team provided the commit `f970102`[8] as a fix.

The commit adds an unused `using-for` directive for the `SafeERC20` library. Even if the `safeTransfer` function from the `SafeERC20` library were used, it would not address the issue correctly. The revert is the desired behavior when a transfer fails, allowing the claim to be repeated.

The original recommendation should be implemented to resolve the issue.

Go back to Findings Summary

I1: Missing event emission for reward pool token staking

Impact:	Info	Likelihood:	N/A
Target:	RewardPool.sol	Type:	Code quality

Description

When users stake tokens through the `StakingManager` contract's `stake` function, the contract emits an event for the overall staking operation. However, it does not emit events specifying which reward pools received the staked tokens.

Listing 12. Excerpt from StakingManager

```
127 for (uint256 _i = 0; _i < rewards; _i++) {
128   RewardType storage _rewardType = _rewardTypes[_i];
129   if (_rewardType.isActive) {
130     IRewardPool _rewardPool = IRewardPool(_rewardType.rewardPool);
131     _rewardPool.stake(_wad);
132   }
133 }
134
135 emit StakingManagerStaked(_account, _wad);
```

Recommendation

Emit an event that specifies which reward pools receive the staked tokens.

Fix 1.1

The issue was fixed in the commit `d0d6f62`[9] by emitting an event that specifies which reward pools receive the staked tokens.

Go back to Findings Summary

I2: `RewardDistributor._claim` leaf is double hashed

Impact:	Info	Likelihood:	N/A
Target:	RewardDistributor.sol	Type:	Gas optimization

Description

The `RewardDistributor._claim` function applies the `keccak256` hash function twice during the `leaf` computation. The Merkle tree proof verification requires the same double hashing in the off-chain computation, which may significantly impact computational performance.

Listing 13. Excerpt from RewardDistributor

```
116 bytes32 _leaf = keccak256(bytes.concat(keccak256(abi.encode(address
    (msg.sender), _wad))));
```

Recommendation

Remove the redundant hash function call in both on-chain and off-chain implementations.

Acknowledgment 1.1

The client acknowledged the issue without providing a comment.

Go back to Findings Summary

I3: Magic numbers

Impact:	Info	Likelihood:	N/A
Target:	StakingManager.sol, RewardPool.sol	Type:	Code quality

Description

The codebase contains multiple instances of magic numbers that lack explanatory context. The following code snippets demonstrate these occurrences:

Constant `1e18`:

Listing 14. Excerpt from StakingManager

```
335 _rewardType.rewardIntegral += (_newRewards * 1e18) / _supply;
```

Listing 15. Excerpt from StakingManager

```
351 + (_userBalance * (_rewardType.rewardIntegral - _userIntegral)) / 1e18;
```

Listing 16. Excerpt from StakingManager

```
367 + (_userBalance * (_rewardType.rewardIntegral - _userIntegral)) / 1e18;
```

Listing 17. Excerpt from RewardPool

```
150 return rewardPerTokenStored + ((_timeElapsed * rewardRate * 1e18) /
    _totalStaked);
```

Listing 18. Excerpt from RewardPool

```
155 return ((_totalStaked * (rewardPerToken() - rewardPerTokenPaid)) / 1e18) +
    rewards;
```

Constant `1000`:

Listing 19. Excerpt from RewardPool

```
170 uint256 _queuedRatio = (_currentAtNow * 1000) / _totalRewards;
```

Recommendation

Define named constants at the contract level and use them throughout the codebase instead of numeric literals. Each constant should have a descriptive name that explains its purpose and documentation comment.

Partial solution 1.1

The magic number `1e18` was replaced with the descriptive constant `WAD` in all identified locations in commit `7041ae7`[10].

The magic number `1000` remains in the codebase.

Go back to Findings Summary

I4: Typos and missing documentation

Impact:	Info	Likelihood:	N/A
Target:	*.sol	Type:	Code quality

Description

The code is well documented. However, the following errors and missing information were identified:

- the `RewardDistributor` contract does not override `_validateParameters()` and thus does not implement any parameter checks. If intentional, document it with an empty function containing an explanatory comment;
- the function `emergencyWidthdraw` in `RewardDistributor` and `IRewardDistributor` contains a typo and should be named `emergencyWithdraw`;
- interfaces in the `interfaces/external` folder lack source code references in their documentation. When external files are copied directly into a codebase, include links to their source in the comments to maintain traceability;
- the constructor of `StakingManager` contains incorrect documentation for the `_cooldownPeriod` parameter;

Listing 20. Excerpt from StakingManager

```
99 * @param  _cooldownPeriod Address of the StakingToken contract
```

- the function name `Authorizable.authorizedAccounts` does not reflect its implementation – rename it to `isAccountAuthorized` or similar;

Listing 21. Excerpt from Authorizable

```
36 function authorizedAccounts(address _account) external view returns (bool
```

```
    _authorized) {
37    return _isAuthorized(_account);
38  }
```

- the struct `StakingManagerParams` lacks documentation for the `cooldownPeriod` parameter.

Listing 22. Excerpt from IStakingManager

```
168 struct StakingManagerParams {
169   uint256 cooldownPeriod;
170 }
```

Recommendation

Review and update the documentation to fix errors and complete missing documentation according to the NatSpec standard.

Partial solution 1.1

The typos were fixed and the documentation was updated in commit `45adce1`[11]. The following remained unresolved:

- documentation was not added to interfaces in the `interfaces/external` folder; and
- function `Authorizable.authorizedAccounts` was not renamed to `isAccountAuthorized`.

Go back to Findings Summary

I5: Code style inconsistencies

Impact:	Info	Likelihood:	N/A
Target:	*.sol	Type:	Code quality

Description

The code style is generally consistent. However, several areas require improvement.

The practice of assigning casted values to variables provides no additional value when the variable is used only once and is named after the called function. Inlining these variables makes the code more concise while maintaining readability. The contracts `RewardPool` and `StakingToken` already implement this inline casting approach. To improve code consistency:

- inline the variables `_uint256` and `_address` in `RewardDistributor._modifyParameters`;

Listing 23. Excerpt from RewardDistributor

```
130 function _modifyParameters(bytes32 _param, bytes memory _data) internal
    override {
131   uint256 _uint256 = _data.toUint256();
132   address _address = _data.toAddress();
133
134   if (_param == 'epochDuration') epochDuration = _uint256;
135   else if (_param == 'rootSetter') rootSetter = _address;
136   else revert UnrecognizedParam();
137 }
```

- inline the variable `_address` in `WrappedToken._modifyParameters`;

Listing 24. Excerpt from WrappedToken

```
85 function _modifyParameters(bytes32 _param, bytes memory _data) internal
   override {
86   address _address = _data.toAddress();
```

```
87   if (_param == 'baseTokenManager') {
88     baseTokenManager = _address;
89   } else {
90     revert UnrecognizedParam();
91   }
92 }
```

- inline the variable `_uint256` in `StakingManager._modifyParameters`;
- add the missing else branch with `UnrecognizedParam` revert in `StakingManager._modifyParameters`; and

Listing 25. Excerpt from StakingManager

```
418 function _modifyParameters(bytes32 _param, bytes memory _data) internal
    override {
419   uint256 _uint256 = _data.toUint256();
420   if (_param == 'cooldownPeriod') _params.cooldownPeriod = _uint256;
421 }
```

The codebase should maintain a consistent style for single-line `if` statements, either with or without curly brackets.

Listing 26. Excerpt from WrappedToken

```
56 if (_baseToken == address(0)) revert WrappedToken_NullBaseToken();
57 if (_baseTokenManager == address(0)) {
58   revert WrappedToken_NullBaseTokenManager();
59 }
```

Recommendation

Unify the code style to ensure consistency across the codebase.

Fix 1.1

The issue was fixed in the commit `0d2dcb5`[12] by inlining the variables.

Go back to Findings Summary

I6: Optimization of function `_getPriceValue`

Impact:	Info	Likelihood:	N/A
Target:	BeefyVeloVaultRelayer.sol, YearnVeloVaultRelayer.sol	Type:	Code quality

Description

The function `_getPriceValue` is duplicated in the `BeefyVeloVaultRelayer` and `YearnVeloVaultRelayer` contracts, which share the same parent contract `AbstractVeloVaultRelayer`. The function can be moved to the `AbstractVeloVaultRelayer` contract by defining the price per full share function (line 54) as virtual in the abstract parent contract and implementing it in the child contracts.

Listing 27. Excerpt from YearnVeloVaultRelayer

```
49 function _getPriceValue() internal view override returns (uint256
   _combinedPriceValue) {
50   // 1 yvToken
51   uint256 _yvTokenBalance = 1_000_000_000_000_000_000;
52
53   // # of velo LP tokens in 1 yvToken
54   uint256 _veloLpBalance = _yvTokenBalance.wmul(yearnVault.pricePerShare());
55
56   // price of 1 velo LP token in chainlink price decimals (8)
57   uint256 _veloLpPrice = veloLpOracle.getCurrentPoolPrice(address(veloPool));
58
59   return (_veloLpBalance * _veloLpPrice) / 1e8;
60 }
```

Listing 28. Excerpt from BeefyVeloVaultRelayer

```
49 function _getPriceValue() internal view override returns (uint256
   _combinedPriceValue) {
50   // 1 mooToken
51   uint256 _mooTokenBalance = 1_000_000_000_000_000_000;
52
53   // # of velo LP tokens in 1 mooToken
```

```
54    uint256 _veloLpBalance =
    _mooTokenBalance.wmul(beefyVault.getPricePerFullShare());
55
56    // price of 1 velo LP token in chainlink price decimals (8)
57    uint256 _veloLpPrice = veloLpOracle.getCurrentPoolPrice(address(veloPool));
58
59    return (_veloLpBalance * _veloLpPrice) / 1e8;
60  }
```

The functions `IBeefyVaultV7.getPricePerFullShare` and `IYearnVault.pricePerShare` return the price per single share. This value is multiplied by one share (10^18) and then divided by the `WAD` precision constant (10^18) in the `wmul` function. Since these operations cancel each other out, the value returned from `getPricePerFullShare` (or `pricePerShare`) can be used directly, eliminating unnecessary multiplication and division operations.

Recommendation

Improve the implementation of the `_getPriceValue` according to the description.

Acknowledgment 1.1

The Let's get HAI team acknowledged the issue with the following comment:

> `_getPriceValue` *is already defined as virtual in the abstract parent contract and we feel the unnecessary multiplication and division operations make the code more readable in this instance.*
>
> — Let's get HAI team

While the function `_getPriceValue` is already defined as virtual, the recommendation was to move implementation of this function to the parent contract `AbstractVeloVaultRelayer` (so it is no longer duplicated in the child

contracts) and create a new virtual function `AbstractVeloVaultRelayer._getPricePerShare`, which would be called instead of `yearnVault.pricePerShare()` and `beefyVault.getPricePerFullShare()`.

The child contracts would implement `_getPricePerShare` as follows:

- `BeefyVeloVaultRelayer` – returns `beefyVault.getPricePerFullShare()`
- `YearnVeloVaultRelayer` – returns `yearnVault.pricePerShare()`

This modification would reduce code duplication while maintaining readability.

Go back to Findings Summary

I7: Unused errors

Impact:	Info	Likelihood:	N/A
Target:	*.sol	Type:	Unused code

Description

The following custom errors are defined but not used in the codebase:

Listing 29. Excerpt from IStakingManager

```
101 /// @notice Throws when trying to withdraw a negative amount
102 error StakingManager_WithdrawNegativeAmount();
```

Listing 30. Excerpt from IStakingManager

```
116 /// @notice Throws when trying to calculate rewards on an inactive reward
    type
117 error StakingManager_InactiveRewardType();
```

Listing 31. Excerpt from IBeefyVeloVaultRelayer

```
14 /// @notice Throws if either of the provided price sources are invalid
15 error BeefyVeloVaultRelayer_InvalidPriceSource();
```

Listing 32. Excerpt from IYearnVeloVaultRelayer

```
14 /// @notice Throws if either of the provided price sources are invalid
15 error YearnVeloVaultRelayer_InvalidPriceSource();
```

The issues were detected using Wake static analysis.

Recommendation

Review all unused errors. Either implement them in the corresponding locations or remove them to simplify the codebase.

Fix 1.1

The issue was fixed in the commit `69ce5f1`[13] by removing the unused errors.

Go back to Findings Summary

I8: Unused `using-for` directives

Impact:	Info	Likelihood:	N/A
Target:	RewardPool.sol, AbstractVeloVaultRelayer.sol	Type:	Unused code

Description

The following libraries imported by `using-for` directives are unused:

Listing 33. Excerpt from RewardPool

```
24 using Math for uint256;
```

Listing 34. Excerpt from AbstractVeloVaultRelayer

```
15 using Math for uint256;
```

The issues were detected using Wake static analysis.

Recommendation

Review the unused `using-for` directives. Either implement them in the corresponding locations or remove them to simplify the codebase.

Partial solution 1.1

The unused `using-for` directives were removed in the commit `11c4b4d`[14].

A new `using-for` directive for the `SafeERC20` library was added to the `RewardDistributor` contract, which remains unused:

Listing 35. Excerpt from RewardDistributor

```
21 contract RewardDistributor is Authorizable, Modifiable, Pausable,
   IRewardDistributor {
22   using Encoding for bytes;
```

```
23   using SafeERC20 for IERC20;
```

Go back to Findings Summary

I9: Unused functions

Impact:	Info	Likelihood:	N/A
Target:	Assertions.sol	Type:	Unused code

Description

The following function in the `Assertions.sol` library is not used in the codebase:

Listing 36. Excerpt from Assertions

```
42 /// @dev Asserts that `_x` is greater than `_y` and returns `_x`
43 function assertGt(int256 _x, int256 _y) internal pure returns (int256 __x) {
44   if (_x <= _y) revert IntNotGreaterThan(_x, _y);
45   return _x;
46 }
```

The issue was detected using Wake static analysis.

Recommendation

Review the unused function in the `Assertions` library. Either implement it in the corresponding locations or remove it to simplify the codebase.

Update 1.1

The Let's get HAI team provided the following comment:

> *This function is used in "src/contracts/TaxCollector.sol:306:5"*
>
> — Let's get HAI team

The function `assertGt` in the `Assertions` library is overloaded with `uint256` and `int256` parameters. Only the `uint256` version is used in the codebase. This

confirms the detection of unused function is correct, as only the `int256` variant was reported as unused.

Even in the referenced `TaxCollector` contract (line 306), the `assertGt` function is used with `uint256` parameters, as shown in the following code snippets:

Listing 37. Excerpt from TaxCollector

```
304 function _validateParameters() internal view override {
305   _params.primaryTaxReceiver.assertNonNull();
306   _params.maxStabilityFeeRange.assertGt(0).assertLt(RAY);
307   _params.globalStabilityFee.assertGtEq(RAY -
    _params.maxStabilityFeeRange).assertLtEq(
308     RAY + _params.maxStabilityFeeRange
309   );
310 }
```

Listing 38. Excerpt from ITaxCollector

```
70 struct TaxCollectorParams {
71   // Address of the primary tax receiver
72   address /*     */ primaryTaxReceiver;
73   // Global stability fee
74   uint256 /* RAY */ globalStabilityFee;
75   // Max stability fee range of variation
76   uint256 /* RAY */ maxStabilityFeeRange;
77   // Max number of secondary tax receivers
78   uint256 /*     */ maxSecondaryReceivers;
79 }
```

Go back to Findings Summary

I10: Variables should be immutable

Impact:	Info	Likelihood:	N/A
Target:	*.sol	Type:	Code quality

Description

The following variables should be marked as immutable:

Listing 39. Excerpt from StakingToken

```
39 IProtocolToken public protocolToken;
```

Listing 40. Excerpt from StakingManager

```
34 /// @inheritdoc IStakingManager
35 IProtocolToken public protocolToken;
36
37 /// @inheritdoc IStakingManager
38 IStakingToken public stakingToken;
```

Listing 41. Excerpt from RewardPool

```
30 IERC20 public rewardToken;
```

Listing 42. Excerpt from FactoryChild

```
14 address public factory;
```

Listing 43. Excerpt from AbstractVeloVaultRelayer

```
19 /// @inheritdoc IAbstractVeloVaultRelayer
20 IVeloPool public veloPool;
21
22 /// @inheritdoc IAbstractVeloVaultRelayer
23 IPessimisticVeloLpOracle public veloLpOracle;
```

The issues were detected using Wake static analysis.

Recommendation

Mark all the mentioned variables as immutable to indicate they are not intended to be changed after contract deployment.

Acknowledgment 1.1

The Let's get HAI team acknowledged the issue with the following comment:

> *We are aware of this convention, however it's not possible to change these values in the code.*
>
> — Let's get HAI team

Go back to Findings Summary

[1] full commit hash: `409cb53517c6c59de415ad774c9088ce91f17e4f`, link to commit PR #109

[2] full commit hash: `800abd667a658be189a77dfab22b54e062d60170`, link to commit PR #109

[3] full commit hash: `4a1663d288c7d2e6836085fe7236c2babe82a6ed`, link to commit PR #109

[4] full commit hash: `af61cdf82cdb5ecebdcd885edf8d62d1de8edefe`, link to commit PR #109

[5] full commit hash: `8e9c0cd715d89ee2c538b52c94aebc535338559e`, link to commit PR #109

[6] full commit hash: `d443c20c4c711afd180c46da7dc84e5bdf087944`, link to commit PR #109

[7] full commit hash: `7422bec312df37633f1a368237a522c68bc46663`, link to commit PR #109

[8] full commit hash: `f970102ca3d2d929d9e9172001534f4e7c27c650`, link to commit PR #109

[9] full commit hash: `d0d6f62cb9af51c7b16dfa1365872fff53d949db`, link to commit PR #109

[10] full commit hash: `7041ae7a5657def1d2e0a255c58de8bcbb0c072b`, link to commit PR #109

[11] full commit hash: `45adce109e6e776620afa8092a32ac8d38a28274`, link to commit PR #109

[12] full commit hash: `0d2dcb50db7e28ad979d1c0595e08ee21f2d430c`, link to commit PR #109

[13] full commit hash: `69ce5f1d1862fd2269d897245a907a905eec1499`, link to commit PR #109

[14] full commit hash: `11c4b4d8792b2e5c197606dd6fde6d996ce064ca`, link to commit PR #109

Report Revision 1.1

Revision Team

Revision team is the same as in Report Revision 1.0.

Overview

Since there were no comprehensive changes in this revision, the complete overview is listed in the Executive Summary section Revision 1.1.

Findings

The following section presents the list of findings discovered in this revision. For the complete list of all findings, Go back to Findings Summary

M2: `stakeToken` can be transferred to any other address while it is still assumed staked

Medium severity issue

Impact:	High	Likelihood:	Low
Target:	StakingManager.sol	Type:	Logic error

Description

The `stakedBalances` mapping is used to track the amount of tokens that a user has staked. Every time a user stakes or withdraws tokens, the value of `stakedBalances[msg.sender]` is updated. However, a user can transfer their stakeToken to any other address and their `stakedBalances[msg.sender]` value will not be changed, if the transferring of tokens is available.

Exploit scenario

1. Alice, a malicious actor, buys stakeToken, which increases her `stakedBalances[msg.sender]` value.
2. Alice sells/swaps the stakeToken on a DEX (assuming this functionality is enabled) to buy protocolToken.
3. Alice's value in the `stakedBalances[msg.sender]` mapping remains unchanged despite no longer owning the stakeToken.
4. By repeating these steps, Alice can artificially inflate her `stakedBalances[msg.sender]` value and collect more rewards than she is actually eligible for.

Recommendation

Add logic to update the `stakedBalances[msg.sender]` value when a user transfers their stakeToken to another address. Another way to mitigate this issue is to remove the functionality of transferring tokens.

Go back to Findings Summary

Appendix A: How to cite

Please cite this document as:

> Ackee Blockchain Security, Let's get HAI: New core features, 4.4.2025.

Appendix B: Wake Findings

This section lists the outputs from the Wake framework used for testing and static analysis during the audit.

B.1. Detectors

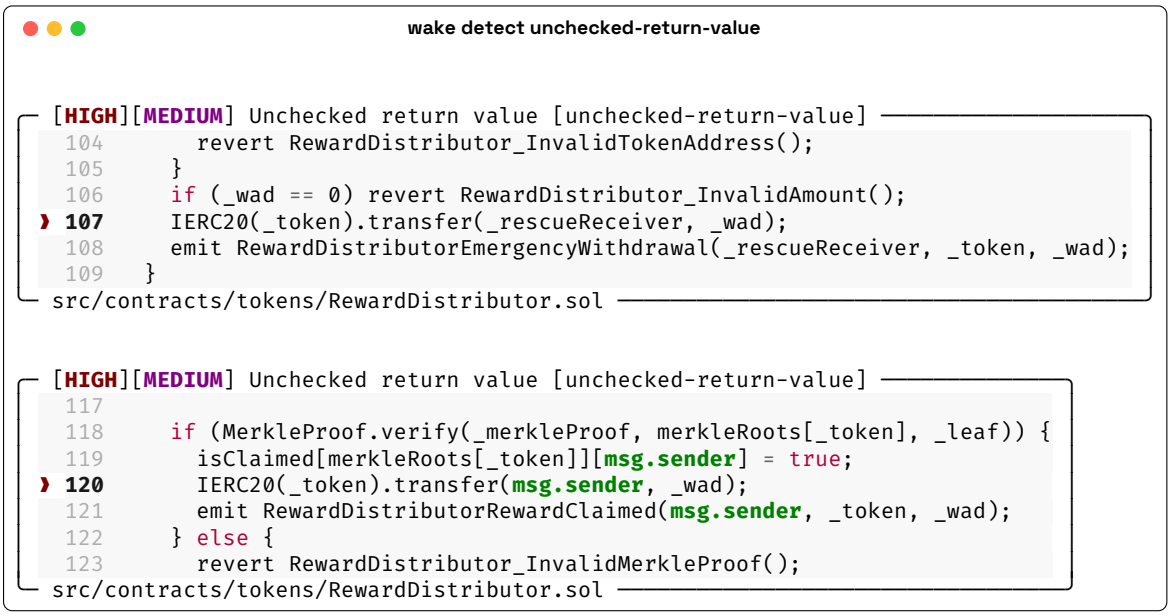


Figure 1. Unchecked return value of `ERC20.transfer` *in* `RewardDistributor`

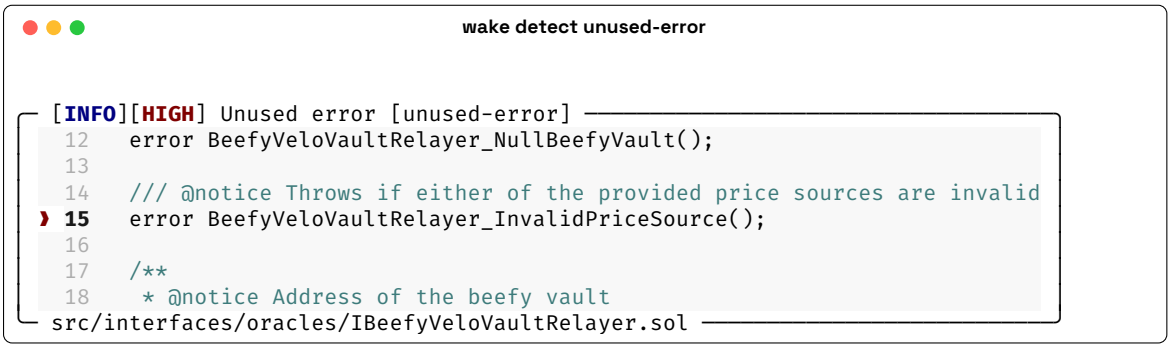


Figure 2. Unused errors in `IBeefyVeloVaultRelayer`

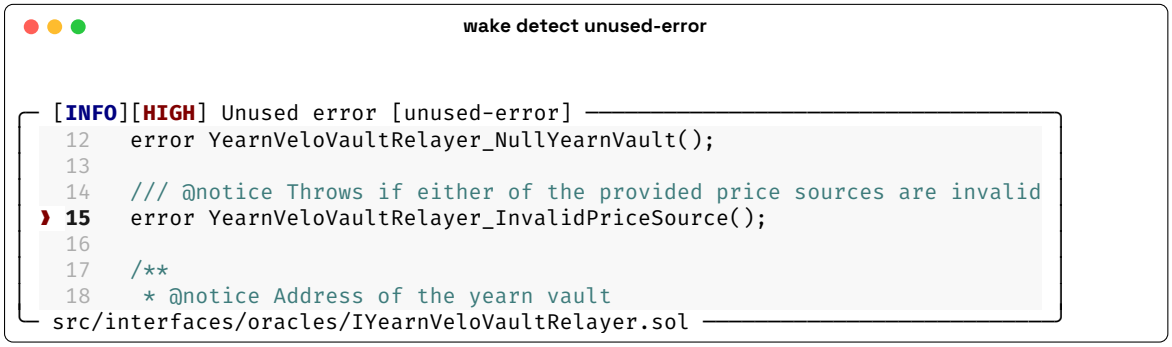


Figure 3. Unused errors in `IYearnVeloVaultRelayer`

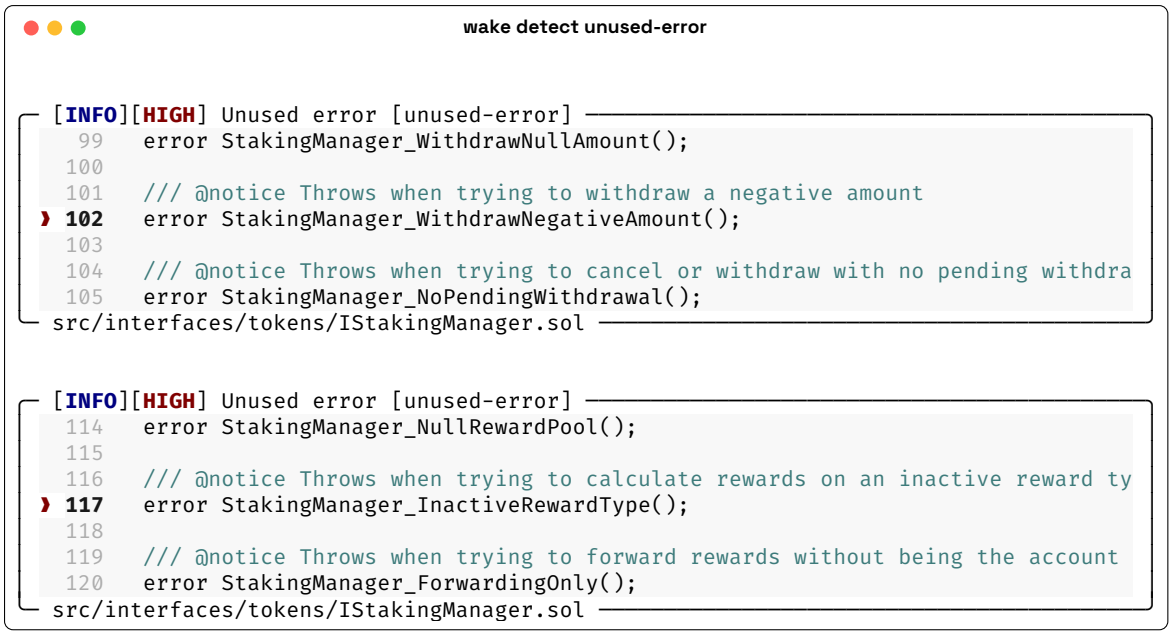


Figure 4. Unused errors in `IStakingManager`

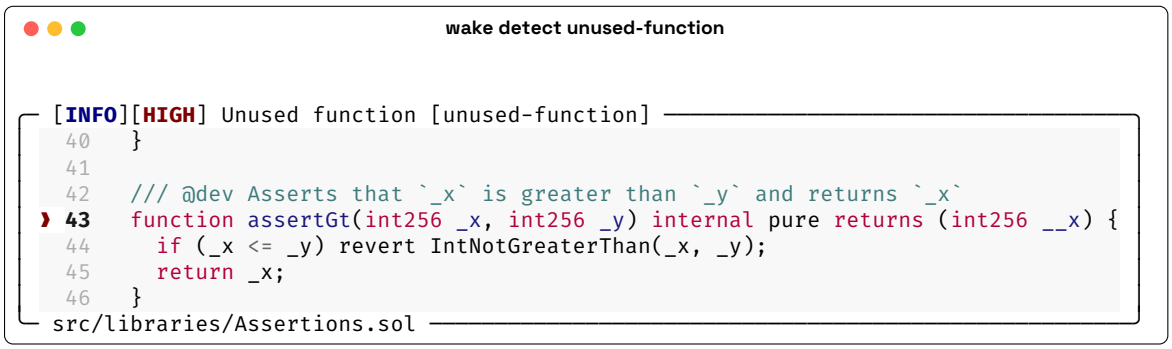


Figure 5. Unused function in `Assertions`

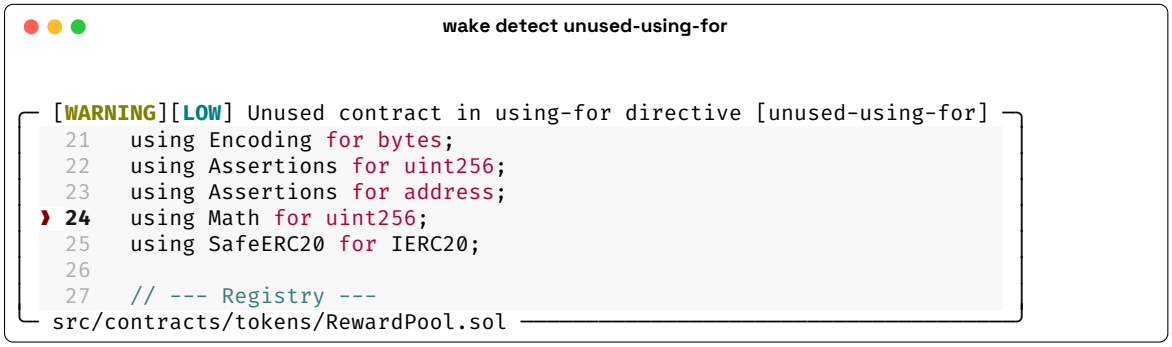


Figure 6. Unused `using-for` *directives in* `RewardPool`

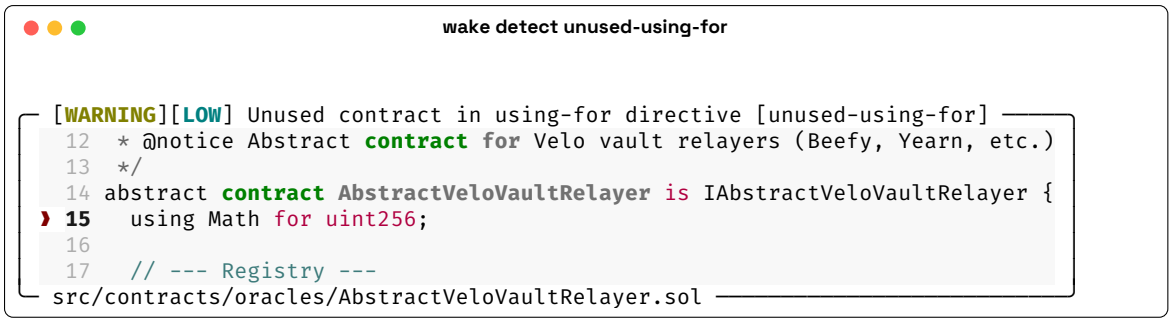


Figure 7. Unused `using-for` *directives in* `AbstractVeloVaultRelayer`

ackee
blockchain security

Thank You

Ackee Blockchain a.s.

Rohanske nabrezi 717/4
186 00 Prague
Czech Republic

hello@ackee.xyz