Haedal - vehaedal & reward distribution

Security Assessment

CertiK Assessed on Oct 24th, 2025

CertiK Assessed on Oct 24th, 2025

Haedal - vehaedal & reward distribution

The security assessment was prepared by CertiK.

Executive Summary

TYPES	ECOSYSTEM	METHODS
Staking	Sui (SUI)	Formal Verification, Manual Review, Static Analysis

LANGUAGE	TIMELINE
Move	Preliminary comments published on 10/08/2025
	Final report published on 10/24/2025

Vulnerability Summary

31	16	0	15	0
Total Findings	Resolved	Partially Resolved	Acknowledged	Declined

1	Centralization	1 Acknowledged	Centralization findings highlight privileged roles & functions and their capabilities, or instances where the project takes custody of users' assets.
0	Critical		Critical risks are those that impact the safe functioning of a platform and must be addressed before launch. Users should not invest in any project with outstanding critical risks.
1	Major	1 Resolved	Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
1	Medium	1 Acknowledged	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
14	Minor	9 Resolved, 5 Acknowledged	Minor risks can be any of the above, but on a smaller scale. They generally do not compromise the overall integrity of the project, but they may be less efficient than other solutions.
14	Informational	6 Resolved, 8 Acknowledged	Informational errors are often recommendations to improve the style of the code or certain operations to fall within industry best practices. They usually do not affect the overall functioning of the code.

TABLE OF CONTENTS

HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

CODEBASE | HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

Repository

https://github.com/haedallsd/vehaedal/tree/0ee35fd446bde4d4564a96be71912bd966ee01d7

https://github.com/haedallsd/reward_distribution/tree/01a2df3318827367b3e4be8c579880be64cf3fb6

https://github.com/haedallsd/vehaedal/tree/691edab91615bad4b61475cd34a0464455559049

https://github.com/haedallsd/reward_distribution/tree/5d54ee98b028bf16ef470c240bc1c5b51fc915b2

https://github.com/haedallsd/vehaedal/tree/fc6a033784ef88c5edf266e804566e2fa4060d61

https://github.com/haedallsd/reward_distribution/tree/0119a9fcf683507ab57f8bf1422f21af93587192

https://github.com/haedallsd/vehaedal/tree/43dbc3d9b6c5f04014dffb7b7b5d74d030092e7c

https://github.com/haedallsd/vehaedal/tree/ddac75540cf59304f9d2550fc00bc95cb1bbadc5

https://github.com/haedallsd/vehaedal/tree/b6eccacfbb81434cbc99165c3820128270657764

Commit

0ee35fd446bde4d4564a96be71912bd966ee01d7

01a2df3318827367b3e4be8c579880be64cf3fb6

691edab91615bad4b61475cd34a0464455559049

5d54ee98b028bf16ef470c240bc1c5b51fc915b2

fc6a033784ef88c5edf266e804566e2fa4060d61

0119a9fcf683507ab57f8bf1422f21af93587192

43dbc3d9b6c5f04014dffb7b7b5d74d030092e7c

ddac75540cf59304f9d2550fc00bc95cb1bbadc5

b6eccacfbb81434cbc99165c3820128270657764

Audit Scope

The file in scope is listed in the appendix.

APPROACH & METHODS

HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

This audit was conducted for Haedal to evaluate the security and correctness of the smart contracts associated with the Haedal - vehaedal & reward distribution project. The assessment included a comprehensive review of the in-scope smart contracts. The audit was performed using a combination of Formal Verification, Manual Review, and Static Analysis.

The review process emphasized the following areas:

- Architecture review and threat modeling to understand systemic risks and identify design-level flaws.
- Identification of vulnerabilities through both common and edge-case attack vectors.
- Manual verification of contract logic to ensure alignment with intended design and business requirements.
- Dynamic testing to validate runtime behavior and assess execution risks.
- Assessment of code quality and maintainability, including adherence to current best practices and industry standards.

The audit resulted in findings categorized across multiple severity levels, from informational to critical. To enhance the project's security and long-term robustness, we recommend addressing the identified issues and considering the following general improvements:

- Improve code readability and maintainability by adopting a clean architectural pattern and modular design.
- Strengthen testing coverage, including unit and integration tests for key functionalities and edge cases.
- Maintain meaningful inline comments and documentations.
- Implement clear and transparent documentation for privileged roles and sensitive protocol operations.
- Regularly review and simulate contract behavior against newly emerging attack vectors.

REVIEW NOTES | HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

Overview

The **Haedal - vehaedal & reward distribution** includes implementation of the veHAEDAL token and reward distribution. Users obtain veHAEDAL tokens by locking HAEDAL tokens, thereby gaining governance voting rights, reward boosts, and other privileges. The amount of veHAEDAL decays over time and returns to zero at the end of the lock period. Users can extend the lock period or increase the locked amount to raise the veHAEDAL amount.

External Dependencies

The project is developed using the Move language and running on the top of the Sui blockchain. The vulnerability and the updates of the language/Sui framework may affect the project as a whole. As the Sui network is rapidly evolving, to avoid any potential compatibility issues and take advantage of new features and improvements, the client should upgrade the Sui framework to the most recent version. Additionally, staying informed about any upcoming updates or changes to the language or framework can help ensure the project remains secure and compatible.

Privileged Roles

To set up the project correctly and ensure that the project functions properly, owners of the following objects are able to use privileged functions, more details in **Centralization Related Risks and Upgradability**.

The advantage of the privileged role in the codebase is that the client reserves the ability to adjust the protocol according to the runtime required to serve the community best. It is also worthy of note the potential drawbacks of these functions, which should be clearly stated through the client's action/plan. Additionally, if the key pairs of privileged accounts are compromised, the project could have devastating consequences.

To improve the trustworthiness of the project, dynamic runtime updates in the project should be notified to the community. Furthermore, any plan to invoke the aforementioned functions should also be considered to move to the execution queue of the `Timelock` contract.

Upgradeability

Developers working with the Sui blockchain have the ability to upgrade packages based on their software iteration requirements. However, this also means that the admin's key store should be handled with caution to prevent any unexpected loss. Additionally, it is important to inform the community about any upgrade plans to address concerns related to centralization and ensure transparency.

Reference:

- Sui Package Upgrades
- Custom Policies

FINDINGS | HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

31	0	1	1	1	14	14
Total Findings	Critical	Centralization	Major	Medium	Minor	Informational

This report has been prepared for Haedal to identify potential vulnerabilities and security issues within the reviewed codebase. During the course of the audit, a total of 31 issues were identified. Leveraging a combination of Formal Verification, Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
HVR-19	**Centralization Related Risks And Upgradability**	**Centralization**	**Centralization**	**Acknowledged**
HVR-33	Duplicate Addition Of `locked_amount` And Missing Access Control For `get_add_vehaedal_amount`	Incorrect Calculation	Major	Resolved
HVR-04	Decay Toggle May Inflate VeHaedal Amount And Extend Lock Unexpectedly	Design Issue	Medium	Acknowledged
HVR-01	Admin Can Create Multiple Pools For The Same Coin Type	Logical Issue	Minor	Acknowledged
HVR-05	Incorrect `SECONDS_PER_DAY` Constant	Incorrect Calculation	Minor	Resolved
HVR-06	Inconsistent Naming Of `HEADAL` Vs. `HAEDAL`	Coding Style	Minor	Acknowledged
HVR-07	Missing Check Lock Expiry In Non-Decaying Mode	Logical Issue	Minor	Resolved
HVR-20	Ownership Field Desynchronization In `VeHAEDAL<T>`	Logical Issue	Minor	Acknowledged
HVR-21	Non-Decaying “Current Lock Weeks” Uses A Stale Saved Value Instead Of Real Time Remaining	Volatile Code	Minor	Resolved
HVR-22	Potential Incorrect Removal Condition In Reward Claim Logic	Logical Issue	Minor	Resolved

ID	Title	Category	Severity	Status
HVR-23	Incorrect Addition To Claimed Rewards List Without Vault Verification	Logical Issue	Minor	Resolved
HVR-24	Missing Check On `pool` And `veheadal`	Logical Issue	Minor	Acknowledged
HVR-26	RewardDistributor'S Use Of A Vector Field Has An Upper Limit	Denial of Service	Minor	Acknowledged
HVR-27	Functions Using `check_all_coins_claimed()` May Cause DoS Due To High Time Complexity	Denial of Service	Minor	Resolved
HVR-28	`veheadal` 'S `initial_amount` May Exceed `locked_amount` In `extend_existing_lock()`	Coding Issue	Minor	Resolved
HVR-29	`update_user_stake()` Does Not Update `total_veheadal` Of Pool	Inconsistency	Minor	Resolved
HVR-34	Incorrect `total_locked` Update In `add_to_existing_stake`	Volatile Code	Minor	Resolved
HVR-02	Potential Inconsistent Minimum Stake Amount Check	Volatile Code	Informational	Resolved
HVR-10	Potential Unused `pending_rewards` And Incomplete Reward Logic	Logical Issue	Informational	Resolved
HVR-11	Unused Functions	Logical Issue	Informational	Resolved
HVR-12	Potential Inconsistent Scale	Volatile Code	Informational	Acknowledged
HVR-13	Concerns When Multiple `PauseClaimConfig` / `ACL` Exists	Volatile Code	Informational	Acknowledged
HVR-14	User Cannot Claim Additional Reward Deposits After Being Removed From `period_rewards`	Design Issue	Informational	Acknowledged
HVR-15	`extend_lock()` May Cause Unintended Value Jump	Logical Issue	Informational	Acknowledged

ID	Title	Category	Severity	Status
HVR-16	Period Mutations After Claims May Cause Unfair Payouts	Volatile Code	Informational	Resolved
HVR-17	Discussion On `Cap`	Design Issue	Informational	Acknowledged
HVR-18	Discussion On Version Control	Logical Issue	Informational	Acknowledged
HVR-25	Operator Withdrawal Before User Claims Causes Reward Loss	Logical Issue	Informational	Acknowledged
HVR-30	`original_lock_weeks` May Be Incorrectly Modified In `extend_lock()`	Coding Issue	Informational	Resolved
HVR-31	Event-Emitting Functions Callable By Anyone	Access Control	Informational	Acknowledged
HVR-32	Potential Inconsistent Logic And Error Code	Volatile Code	Informational	Resolved

HVR-19 | Centralization Related Risks And Upgradability

Category	Severity	Location	Status
Centralization	**Centralization**		**Acknowledged**

Description

Vehaedal

In the `admin` module, the role `AdminCap` holds authority over the following functions:

- `set_min_stake_amount<T>()`
- `update_pool_version<T>()`
- `transfer_admin_cap()`
- `share_acl()`
- `add_minor_signs_to_acl()`
- `del_minor_signs()`

ACL member `minor_signs` have control over:

- `set_min_stake_amount_v2<T>()`

In the `staking` module, the role `AdminCap` holds authority over the following function:

- `create_and_share_pool<T>()`

Reward Distribution

In the `breaker` module, ACL member `breaker` has control over:

- `deposit_reward()`

In the `reward_distribution` module, the role `AdminCap` holds authority over the following functions:

- `init_pause_reward()`
- `share_acl()`
- `set_operator_cap_to_address()`
- `add_breaker_to_acl()`
- `del_breaker_to_acl()`
- `add_robot_to_acl()`
- `del_robot_to_acl()`

In the `reward_distribution` module, the role `OperatorCap` holds authority over the following functions:

- `admin_withdraw_unclaimed_rewards()`
- `add_user_rewards()`
- `deposit_reward()`

In the `robot` module, ACL member `robot` have control over:

- `add_user_rewards()`

If any of these privileged accounts are compromised, an attacker could leverage their granted permissions to alter protocol states, create new pools, and upgrade the contract.

It is worth noting that the rewards distribution relies on the actual rewards remaining in the pool, for example, if the team withdraws the rewards via `admin_withdraw_unclaimed_rewards` , the user will not be able to claim the rewards.

In addition, developers working with the Sui blockchain can upgrade packages based on their software iteration requirements. However, this also means that the admin's key store should be handled carefully to prevent any unexpected losses. It is important to inform the community about any upgrade plans to address concerns related to centralization and ensure transparency.

More information can be found:

- Sui Package Upgrades
- Third-Party Package upgrades

Recommendation

The risk describes the current project design and potentially makes iterations to improve in the security operation and level of decentralization, which in most cases cannot be resolved entirely at the present stage. We advise the client to carefully manage the privileged account's private key to avoid any potential risks of being hacked. In general, we strongly recommend centralized privileges or roles in the protocol be improved via a decentralized mechanism or smart-contract-based accounts with enhanced security practices, e.g., multisignature wallets.

Indicatively, here are some feasible suggestions that would also mitigate the potential risk at a different level in terms of short-term, long-term and permanent:

Short Term:

Timelock and Multi sign (⅔, ⅗) combination *mitigate* by delaying the sensitive operation and avoiding a single point of key management failure.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Assignment of privileged roles to multi-signature wallets to prevent a single point of failure due to the private key compromised;
  AND

- A medium/blog link for sharing the timelock contract and multi-signers addresses information with the public audience.

Long Term:

Timelock and DAO, the combination, *mitigate* by applying decentralization and transparency.

- Time-lock with reasonable latency, e.g., 48 hours, for awareness on privileged operations;
  AND
- Introduction of a DAO/governance/voting module to increase transparency and user involvement.
  AND
- A medium/blog link for sharing the timelock contract, multi-signers addresses, and DAO information with the public audience.

Permanent:

Renouncing the ownership or removing the function can be considered *fully resolved*.

- Renounce the ownership and never claim back the privileged roles.
  OR
- Remove the risky functionality.

Alleviation

[Haedal, 10/10/2025]:

1. Use ACL address lists to prevent privilege evasion.
2. AdminCap and OperatorCap will ensure multi-signature support by at least four administrators.
3. The team will also regularly review the private key security of all addresses associated with centralized roles.

[CertiK, 10/10/2025]: It is suggested to implement the aforementioned methods to avoid centralized failure. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

HVR-33 Duplicate Addition Of `locked_amount` And Missing Access Control For `get_add_vehaedal_amount`

Category	Severity	Location	Status
Incorrect Calculation	Major	sources/VeToken.move (vehaedal-1018): 153, 164	Resolved

Description

Based on the updated code 43dbc3d, the following issues has been indentified with `get_add_vehaedal_amount`.

- The `get_add_vehaedal_amount` is missing access control, which allows any VeHAEDAL holder manipulate the `locked_amount`.

`VeToken.move`

```
153  @>  public fun get_add_vehaedal_amount<T>(
154          veheadal: &mut VeHAEDAL<T>,
155          additional_amount: u64,
156          clock: &Clock
157      ): u64 {
158          let now = clock::timestamp_ms(clock);
159          let remaining_lock_time = if (now >= veheadal.lock_end_time) {
160              0
161          } else {
162
           (veheadal.lock_end_time - now) / (DAYS_PER_WEEK * SECONDS_PER_DAY *
1000)
163          };
164  @>      veheadal.locked_amount = veheadal.locked_amount + additional_amount;
165
166          (((additional_amount * remaining_lock_time) / WEEKS_PER_YEAR) as u64)
167      }
```

- Also, In `add_to_existing_stake()`, the function `get_add_vehaedal_amount()` is called twice — once directly, and once indirectly via `vetoken::add_locked_amount()`.This means that when a user stakes through `add_to_existing_stake()`, `veheadal.locked_amount` may be increased twice. As a result, the user may eventually receive funds in `unstake_and_claim()` that do not actually belong to them.

`VeToken.move`

```
241      public(package) fun add_locked_amount<T>(
242          veheadal: &mut VeHAEDAL<T>,
243          additional_amount: u64,
244          clock: &Clock
245      ): u64 {
246
@>       let additional_veheadal =
get_add_vehaedal_amount(veheadal,additional_amount,clock);
247          let new_initial_amount = veheadal.initial_amount + additional_veheadal;
248          veheadal.initial_amount = new_initial_amount;
249          veheadal.current_amount = get_current_value(veheadal, clock);
250          new_initial_amount
251      }
```

`Staking.move`

```
206
@>       let additional_veheadal =
vetoken::get_add_vehaedal_amount(veheadal,amount,clock);
207          staking_pool::update_user_stake<T>(
208              pool,
209              veheadal,
210              balance,
211              additional_veheadal,
212              ctx
213          );
214
@>       let new_veheadal_amount = vetoken::add_locked_amount(veheadal, amount,
clock);
215          Events::emit_stake_added_event<T>(
216              sender,
217              object::id(veheadal),
218              amount,
219              new_veheadal_amount,
220          );
```

Proof of Concept

We simulated the following scenario: User 1 staked 100,000 HAEDAL, and User 2 staked 10,000 HAEDAL and then called `add_to_existing_stake` with an additional 90,000 HAEDAL. In the end, after the lock period, User 2 was able to withdraw 190,000 HAEDAL from the contract — exceeding the amount originally staked.

```
#[test_only]
module veheadal::staking_test_audit {
    use sui::test_utils::assert_eq;
    use sui::test_scenario as ts;
    use sui::coin::{Self, Coin};
    use sui::object::{Self, ID};
    use sui::clock::{Self, Clock};
    use sui::test_utils;
    use std::debug;
    use std::string;
    use sui::transfer;
    use sui::test_scenario::Scenario;
    use veheadal::admin::{Self, AdminCap};
    use veheadal::staking::{Self};
    use veheadal::staking_pool::{Self, StakingPool};
    use veheadal::vetoken::{Self, VeHAEDAL};
    //
    public struct HAEDAL has drop {}
    //
    const ADMIN: address = @0xAD;
    const USER1: address = @0x1;
    const USER2: address = @0x2;
    //
    const STAKE_AMOUNT: u64 = 10000;
    //
    const WEEKS_PER_YEAR: u64 = 52;
    //
    const DAYS_PER_WEEK: u64 = 7;
    //
    const SECONDS_PER_DAY: u64 = 86400;
    // : AdminCap
    #[test_only]
    fun init_staking_pool(scenario: &mut Scenario): AdminCap {
        //  AdminCap
        let admin_cap = admin::create_admin_cap_for_testing(ts::ctx(scenario));
        //
        ts::next_tx(scenario, ADMIN);
        {
            //
            let pool = staking_pool::create_pool<HAEDAL>(ts::ctx(scenario));
            //
            staking_pool::share_pool(pool);
        };
        //
        admin_cap
    }
    // :
    #[test_only]
```

```
    fun setup(): (Scenario, Clock, AdminCap) {
        //
        let mut scenario = ts::begin(ADMIN);
        //
        let clock = clock::create_for_testing(ts::ctx(&mut scenario));
        //
        let admin_cap = init_staking_pool(&mut scenario);
        //
        ts::next_tx(&mut scenario, ADMIN);
        {
            let coin = coin::mint_for_testing<HAEDAL>(100000, ts::ctx(&mut
scenario));
            transfer::public_transfer(coin, USER1);
        };
        ts::next_tx(&mut scenario, ADMIN);
        {
            let coin = coin::mint_for_testing<HAEDAL>(100000, ts::ctx(&mut
scenario));
            transfer::public_transfer(coin, USER2);
        };
        (scenario, clock, admin_cap)
    }
    // :
    #[test_only]
    fun get_pool(scenario: &Scenario): StakingPool<HAEDAL> {
        ts::take_shared<StakingPool<HAEDAL>>(scenario)
    }
    // :
    #[test_only]
    fun return_pool(scenario: &mut Scenario, pool: StakingPool<HAEDAL>) {
        ts::return_shared(pool)
    }
    #[test]
    fun test_add_more_stake() {
        let (mut scenario, mut clock, admin_cap) = setup();
        // user1 stake 100_000 token
        ts::next_tx(&mut scenario, USER1);
        {
            let mut pool = get_pool(&scenario);
            let mut coin = ts::take_from_sender<Coin<HAEDAL>>(&scenario);
            debug::print(&string::utf8(b"user1 stake 100_000 token"));
            //  add_stake
            let stake_coin = coin::split(&mut coin,  10 * STAKE_AMOUNT, ts::ctx(&mut
scenario));
            staking::add_stake<HAEDAL>(
                &mut pool,
                stake_coin,
                26, //
                true, //
```

```
                &clock,
                ts::ctx(&mut scenario)
            );
            ts::return_to_sender(&scenario, coin);
            return_pool(&mut scenario, pool);
        };
        // user2 stake 10_000 token
        ts::next_tx(&mut scenario, USER2);
        {
            let mut pool = get_pool(&scenario);
            let mut coin = ts::take_from_sender<Coin<HAEDAL>>(&scenario);
            debug::print(&string::utf8(b"user2 stake 10_000 token"));
            //  add_stake
            let stake_coin = coin::split(&mut coin, STAKE_AMOUNT, ts::ctx(&mut
scenario));
            staking::add_stake<HAEDAL>(
                &mut pool,
                stake_coin,
                26, //
                true, //
                &clock,
                ts::ctx(&mut scenario)
            );
            ts::return_to_sender(&scenario, coin);
            return_pool(&mut scenario, pool);
        };
        // user2 add 90_000 token stake
        ts::next_tx(&mut scenario, USER2);
        {
            let mut pool = get_pool(&scenario);
            let mut coin = ts::take_from_sender<Coin<HAEDAL>>(&scenario);
            let mut vehaedal = ts::take_from_sender<VeHAEDAL<HAEDAL>>(&scenario);
            debug::print(&string::utf8(b"user2 add staking 90_000 token"));
            //  add_stake
            let stake_coin = coin::split(&mut coin,  9 * STAKE_AMOUNT, ts::ctx(&mut
scenario));
            staking::add_to_existing_stake<HAEDAL>(
                &mut pool,
                &mut vehaedal,
                stake_coin,
                &clock,
                ts::ctx(&mut scenario)
            );
            ts::return_to_sender(&scenario, coin);
            ts::return_to_sender(&scenario, vehaedal);
            return_pool(&mut scenario, pool);
        };
        // after the lock period
        debug::print(&string::utf8(b"after the 26 lock period"));
```

```
        clock::increment_for_testing(&mut clock, 26 * 7 * 24 * 60 * 60 * 1000); //
26
        //user2 unstake and claim all token
        ts::next_tx(&mut scenario, USER2);
        {
            let mut pool = get_pool(&scenario);
            let veheadal = ts::take_from_sender<VeHAEDAL<HAEDAL>>(&scenario);
            debug::print(&string::utf8(b"user2 unstake and claim all token"));
            let coin = staking::unstake_and_claim_coin<HAEDAL>(
                &mut pool,
                veheadal,
                &clock,
                ts::ctx(&mut scenario)
            );
            debug::print(&string::utf8(b"user2 unstake and claim token amount:"));
            debug::print(&coin.value());
            return_pool(&mut scenario, pool);
            transfer::public_transfer(coin, USER2);
        };
        //  AdminCap
        admin::destroy_admin_cap_for_testing(admin_cap);
        clock::destroy_for_testing(clock);
        ts::end(scenario);
    }
}
```

```
Running Move unit tests
[debug] "user1 stake 100_000 token"
[debug] "user2 stake 10_000 token"
[debug] "user2 add staking 90_000 token"
[debug] "after the 26 lock period"
[debug] "user2 unstake and claim all token"
[debug] "user2 unstake and claim token amount:"
[debug] 190000
[ PASS    ] veheadal::staking_test_audit::test_add_more_stake
Test result: OK. Total tests: 1; passed: 1; failed: 0
```

Recommendation

It is recommended to ensure that `veheadal.locked_amount` is updated only once within `add_to_existing_stake()` and adding access control.

Alleviation

[Haedal, 10/21/2025]: The team heeded the advice and resolved the issue in commit b2fa4d232eaa02d22daa952f0d3710ea6fb5cbee.

HVR-04 Decay Toggle May Inflate VeHaedal Amount And Extend Lock Unexpectedly

Category	Severity	Location	Status
Design Issue	Medium	sources/VeToken.move (vehaedal): 190, 205	Acknowledged

Description

Stopping decay when at least 1 week remains freezes ve weight at the full `initial_amount`, and stores the remaining time rounded up to whole weeks. Starting decay later resets the countdown from "now + saved weeks," effectively extending the lock and restarting decay from the `initial_amount` again, which inflates the weight.

```
public(package) fun get_current_value<T>(veheadal: &VeHAEDAL<T>, clock: &Clock): u64
{
    if (!veheadal.is_decaying) {
        return veheadal.initial_amount
    };
    let now = clock::timestamp_ms(clock);
    if (now >= veheadal.lock_end_time) {
        return 0
    };
```

If rewards or governance weights depend on current ve, users can game distribution/votes by toggling near the end to spike weight and/or delay expiry.

Scenario

1. Stake with decay enabled: amount = 100, lock_weeks = 10. initial_amount = 100 * 10 / 52 = 19 (integer division).
2. Advance time to 1.2 weeks before lock_end. Current ve ≈ initial_amount × (remaining/total) ≈ 19 × 1.2/10 ≈ 2 (int).
3. Call `stop_decay`:

- Allowed (remaining weeks floored = 1 ≥ 1).
- Saved remaining weeks = ceil(1.2) = 2.
- is_decaying = false, and get_current_value now returns initial_amount = 19 (instant weight jump from ~2 to 19).

4. Later call `start_decay`:

- lock_start_time = now; lock_end_time = now + 2 weeks (extends original maturity by ~0.8 week).
- Weight restarts decaying from initial_amount again.

Actual result:

- Weight can be boosted to initial_amount on demand when ≥1 week remains.
- Lock end effectively extended by up to just under 1 week due to ceil.

Recommendation

Recommend that the team reconsider this design.

Alleviation

[Haedal, 10/10/2025]:

Regarding initial_amount, this is a redundant field left over from history.

In fact, all current logic will not use functions related to initial_amount, including get_current_veheadal and the new_veheadal_amount event generated by add_to_existing_stake.

This is because in the initial version of Veheadal, we used the protocol to perform staking calculations and real-time decay. However, we later discovered that frequent doc errors prevented users from receiving rewards or correctly claiming their share of rewards. Therefore, we have migrated the protocol's real-time decay functionality to the management platform. This allows for real-time and accurate allocation of user equity by scanning user operation records, and also applies restrictions based on user staking objects in the protocol. This ensures the most accurate, timely, and secure staking method.

Of course, in subsequent upgrades, we will consider optimizing redundant fields and functions related to get_current_value and initial_amount. This will be implemented in the near future.

[CertiK, 10/10/2025]: Thank you for the update! The team mentioned that the real-time decay feature of the protocol has been migrated to the management platform. We would appreciate more details on how the management platform manages the decay logic to ensure consistency. Specifically, is this an off-chain platform, and how is the current amount calculated within the platform? Also, how does the team handle scenarios where the user starts, stops, and then restarts decay? This includes details on the calculation of the amount and the lock day.

[Haedal, 10/13/2025]: Thank you for your response. The current off-chain platform approach is:

1. Based on user operation logs, we synchronize important operation logs such as staking, lockup extensions, stake increases, and unlocks in real time.

2. We perform one-to-one user binding and verification based on operation logs and user-staked object data to ensure the authenticity and security of user stake amounts and data.

3. Based on this data, the platform calculates decay and stake percentage within the rules.

4. Based on the latest user percentage, we recalculate and verify, ensuring accuracy before calling the reward contract to set the user reward ratio.

5. Users claim their earned rewards through the official website.

We have implemented verification and protection mechanisms in each of these steps, ensuring that all important data is

sourced on-chain to ensure user rights and interests.

HVR-01 | Admin Can Create Multiple Pools For The Same Coin Type

Category	Severity	Location	Status
Logical Issue	Minor	sources/Staking.move (vehaedal): 25~26	Acknowledged

Description

The current implementation does not enforce a uniqueness constraint for staking pools. Specifically, the `create_and_share_pool<T>` function calls `create_pool<T>` without checking whether a pool of the same token type already exists:

```
public entry fun create_and_share_pool<T>(_admin_cap: &AdminCap, ctx: &mut
TxContext) {
    let mut pool = staking_pool::create_pool<T>(ctx);
    ...
}

public(package) fun create_pool<T>(ctx: &mut TxContext): StakingPool<T> {
    StakingPool<T> {
```

As a result, administrators can deploy multiple pools for the same token (e.g., `StakingPool<SUI>`). This may lead to fragmented staking liquidity, inconsistent reward accounting, and potential confusion for integrators and end-users.

We would like to discuss this finding with the team to ensure this implementation is correct and aligns with the intended business logic.

Recommendation

We would like to discuss this finding with the team to ensure this implementation is correct and aligns with the intended business logic.

Alleviation

[Haedal, 10/10/2025]: This is a known issue, but since user staked interests are concentrated in VeHAEDAL, there is no consideration of modifying the public struct VeHAEDAL in the short term. Secondly, the public struct VEHEADAL has been abandoned.

[CertiK, 10/10/2025]:

Thanks for the update, however, the finding is trying to describe the scenario that multiple pools with the same coin type may exist, could you please double check.

[Haedal, 10/13/2025]: Thank you for your response.

1. Currently, our business only implements the Vehaedal protocol for Haedal tokens. There will be no future calls to create_and_share_pool.

2. Because create_and_share_pool requires AdminCap, which is the highest permission, this will require simultaneous confirmation from at least four (or even more) administrators (multi-sig call). This prevents the re-creation of the same pool and the creation of redundant pool information.

In fact, the Vehaedal protocol was originally designed solely for Haedal tokens.

HVR-05 | Incorrect `SECONDS_PER_DAY` Constant

Category	Severity	Location	Status
Incorrect Calculation	Minor	sources/VeToken.move (vehaedal): 6~7	Resolved

Description

The constant `SECONDS_PER_DAY` is incorrectly defined as `3600` instead of the correct value `86400` (`60 * 60 * 24`). This constant is used in the `create_veheadal()` function to calculate the lock end time:

```
let lock_end_time = now + (lock_weeks * DAYS_PER_WEEK * SECONDS_PER_DAY * 1000); // convert to milliseconds
```

Because of the incorrect constant, the computed lock duration is significantly shorter than expected. This results in premature unlocks and breaks the intended locking mechanism. Depending on the system's design, this may allow users to bypass lockup periods, undermining the protocol's security and economic assumptions.

Recommendation

It is recommended to update the definition of `SECONDS_PER_DAY` to the correct value of `86400`.

Alleviation

[Haedal, 10/10/2025]: The team heeded the advice and resolved the issue by updating the value in commit 291cbff9265727ec0b6a0ca58a946bce36968eb4

HVR-06 | Inconsistent Naming Of `HEADAL` Vs. `HAEDAL`

Category	Severity	Location	Status
Coding Style	Minor	sources/VeToken.move (vehaedal): 11~14	Acknowledged

Description

The project contains inconsistent spellings (`HEADAL` vs. `HAEDAL`) and mixed case usage (e.g., `veHEADAL`, `VeHAEDAL`), which leads to ambiguity:

```
[package]
name = "veHEADAL"
```

```
/// veHEADAL token type
public struct VEHEADAL has drop {}

/// veHEADAL token object
public struct VeHAEDAL<phantom T> has key, store {}
```

These inconsistencies cause confusion and increase the likelihood of incorrect usage. For example:

- Developers, external scripts, front-end code, or indexers may mistakenly reference the wrong type or import name. This can lead to failed transactions (`abort`) or miscounted statistics in dashboards.
- Since interfaces widely rely on generics (`T` and `VeHAEDAL<T>`), inconsistent naming may cause mixed usage. Errors may only surface at compile time or during execution, increasing operational risk.
- Events or type-based identification may be misinterpreted, leading to monitoring and integration errors.

Naming inconsistencies do not directly compromise protocol security but introduce a significant risk of operational mistakes, transaction failures, and incorrect external integrations. This undermines developer efficiency and system reliability.

Recommendation

It is recommended to standardize naming conventions for all related types and objects (e.g., consistently use `HEADAL` or `HAEDAL` across all modules and types).

Alleviation

[Haedal, 10/10/2025]: This is a known issue, but since user staked interests are concentrated in VeHAEDAL, there is no consideration of modifying the public struct VeHAEDAL in the short term. Secondly, the public struct VEHEADAL has been abandoned.

HVR-07 | Missing Check Lock Expiry In Non-Decaying Mode

Category	Severity	Location	Status
Logical Issue	Minor	sources/Staking.move (vehaedal): 201~203	Resolved

Description

The function `add_to_existing_stake()` allows users to increase their stake in an existing position. However, it only performs an expiry check when the stake is in **decaying mode**:

`Staking.move`

```
201  if (vetoken::is_decaying(veheadal)) {
202      assert!(!vetoken::is_expired(veheadal, clock), ELockShouldBeExpired);
203  };
```

When `vetoken::is_decaying(veheadal) == false`, no validation is performed against the current time and the `lock_end_time`. This creates a scenario where the stake's lock period may already have expired, yet the function still accepts additional deposits.

In such cases, the calculation of `additional_veheadal` results in zero because `remaining_lock_time` becomes zero:

`VeToken.move`

```
262  let now = clock::timestamp_ms(clock);
263  let remaining_lock_time = if (now >= veheadal.lock_end_time) {
264      0
265  } else {
266      (veheadal.lock_end_time - now) / (DAYS_PER_WEEK * SECONDS_PER_DAY * 1000)
267  };
268
269
let additional_veheadal = (additional_amount * remaining_lock_time) /
WEEKS_PER_YEAR;
```

Despite this, the user's assets are still transferred into the `pool.rewards_pool`, effectively locking funds without granting any `veheadal` staking benefits.

Recommendation

It is recommended to refactor the function logic to guarantee that users always receive a corresponding amount of `veheadal` when adding to an existing stake.

Alleviation

[Haedal, 10/10/2025]: When vetoken::is_decaying(veheadal) == false, the protocol will determine current_lock_weeks. If it is less than 1, no action will be taken. Therefore, it will not happen that when vetoken::is_decaying(veheadal) == false and current_lock_weeks = 0, the user can successfully add more coins to the stake.

[Certik，10/10/2025]: Thanks for the update. When `vetoken::is_decaying(veheadal) == false` and `now >= veheadal.lock_end_time`, users can still stake by calling `add_to_existing_stake()`. This increases pool.total_staked but does not increase the veheadal amount. In other words, under the non-decay mode, users' voting power does not increase. We'd like to confirm whether the team has considered this scenario.

[Haedal, 10/17/2025]: Thank you for your reply. Please consult the code. When vetoken::is_decaying(veheadal) == false, current_lock_weeks = vetoken::get_remaining_lock_weeks_when_stopped_decay(veheadal) instead of 0.

[CertiK, 10/17/2025]: Thank you for your response. As you mentioned, in the `add_to_existing_stake()` function, the `current_lock_weeks` used to verify that the remaining time is sufficient is indeed not zero when `vetoken::is_decaying(veheadal) == false`.

`Staking.move`

```
190         let current_lock_weeks = if (!vetoken::is_decaying(veheadal)) {
191             vetoken::get_remaining_lock_weeks_when_stopped_decay(veheadal)
192         } else {
193             let now = sui::clock::timestamp_ms(clock);
194
           let remaining_lock_time = if (now >=
vetoken::get_lock_end_time(veheadal)) {
195                 0
196             } else {
197
               (vetoken::get_lock_end_time(veheadal) - now) / (DAYS_PER_WEEK *
SECONDS_PER_DAY * 1000)
198             };
199             remaining_lock_time
200         };
201         assert!(current_lock_weeks >= 1, EInvalidLockDuration);
```

However, in the invoked functions `vetoken::add_locked_amount()` and `get_add_vehaedal_amount()`, the case where `vetoken::is_decaying()` is false is not differentiated — instead, the `additional_vehaedal` is directly computed using the remaining lock duration.

`Staking.move`

```
214
@>      let new_veheadal_amount = vetoken::add_locked_amount(veheadal, amount,
clock);
```

`VeToken.move`

```
241     public(package) fun add_locked_amount<T>(
242         veheadal: &mut VeHAEDAL<T>,
243         additional_amount: u64,
244         clock: &Clock
245     ): u64 {
246
@>      let additional_veheadal =
get_add_vehaedal_amount(veheadal,additional_amount,clock);
247         let new_initial_amount = veheadal.initial_amount + additional_veheadal;
248         veheadal.initial_amount = new_initial_amount;
249         veheadal.current_amount = get_current_value(veheadal, clock);
250         new_initial_amount
251     }
```

VeToken.move

```
152     /// Check if a veHEADAL token is in decay mode
153     public fun get_add_vehaedal_amount<T>(
154         veheadal: &mut VeHAEDAL<T>,
155         additional_amount: u64,
156         clock: &Clock
157     ): u64 {
158         let now = clock::timestamp_ms(clock);
159         let remaining_lock_time = if (now >= veheadal.lock_end_time) {
160             0
161         } else {
162
           (veheadal.lock_end_time - now) / (DAYS_PER_WEEK * SECONDS_PER_DAY *
1000)
163         };
164         veheadal.locked_amount = veheadal.locked_amount + additional_amount;
165
166         (((additional_amount * remaining_lock_time) / WEEKS_PER_YEAR) as u64)
167     }
```

We recommend modifying `get_add_vehaedal_amount()` to behave consistently with the intended logic in `add_to_existing_stake()`, by differentiating the `vetoken::is_decaying()` case and correctly computing the `additional_veheadal`.

[Haedal, 10/22/2025]: The team heeded the advice and resolved the issue by modifying `get_add_vehaedal_amount()` to behave consistently with the intended logic in `add_to_existing_stake()` in commit ddac75540cf59304f9d2550fc00bc95cb1bbadc5

HVR-20 | Ownership Field Desynchronization In `VeHAEDAL<T>`

Category	Severity	Location	Status
Logical Issue	Minor	sources/VeToken.move (vehaedal): 14~15	Acknowledged

Description

The struct `VeHAEDAL<T>` is declared with the `key` and `store` ability, which makes it freely transferable with `transfer::public_transfer`:

```
public struct VeHAEDAL<phantom T> has key, store {
    id: UID,
    ...
    /// Owner address
    owner: address,
    ...
}
```

As a result, the `owner` field is not automatically updated when the object is transferred on-chain. This creates a risk that the **recorded owner** (`owner` field) diverges from the **actual owner** (derived from object possession).

Any logic that relies on `get_owner()` for permission checks may therefore become inaccurate. For example:

- Access control depending on `get_owner()` could be rejected.
- Application-level logic (such as reward distribution or voting rights) could be assigned to the wrong party.

Recommendation

It is recommended to either remove the redundant `owner` field or remove the `store` ability from the `VeHAEDAL` struct.

Alleviation

[Haedal, 10/10/2025]: This is a known issue. The agreement stipulates that once a user pledges haedal, the rights and interests cannot be transferred.

HVR-21 Non-Decaying “Current Lock Weeks” Uses A Stale Saved Value Instead Of Real Time Remaining

Category	Severity	Location	Status
Volatile Code	Minor	sources/Staking.move (vehaedal): 189~191, 232~234	Resolved

Description

In non-decaying mode, both functions `add_to_existing_stake` and `extend_existing_lock` use `remaining_lock_weeks_when_stopped_decay` to derive current lock weeks instead of using `lock_end_time` or `lock_end_time + remaining_lock_weeks_when_stopped_decay`.

For positions that never entered decay, this field is initialized to the full original weeks and never decreases, so validations (e.g., max 52 weeks on extend) can be inaccurate relative to the true time remaining until `lock_end_time`.

Recommendation

Recommend reconsidering the calculation for `current_lock_weeks` with non non-decaying position.

Alleviation

[Haedal, 10/10/2025]: The agreement stipulates that if the market never enters a decline, the maximum period is also 52 weeks.

HVR-22 | Potential Incorrect Removal Condition In Reward Claim Logic

Category	Severity	Location	Status
Logical Issue	Minor	sources/reward_distribution.move (reward_distribution): 665	Resolved

Description

In the `claim_rewards_process` function, users are only removed from `period_rewards` if they have claimed all globally supported coin types (via `distributor.supported_coin_types`).

```
            let all_claimed =
check_all_coins_claimed(&distributor.supported_coin_types,
&distributor.claimed_rewards, sender, period);
```

```
fun check_all_coins_claimed(
        period_vaults: &vector<String>,
        claimed_rewards: &Table<u32, Table<address, vector<String>>>,
        user: address,
        period: u32
    ): bool {
        let available_coins = period_vaults;
        ...
    }
```

However, this check uses the global list instead of the period-specific coin types (from `period_vaults`), leading to inconsistencies. For periods with fewer coins than the global list, users can never be removed even after claiming all available rewards for that period.

Recommendation

Update the `check_all_coins_claimed` call to pass `period_vaults` instead of `supported_coin_types` to ensure the check is period-specific.

Alleviation

[Haedal, 10/10/2025]: The team heeded the advice and resolved the issue by using `period_vaults` in commit 5d54ee98b028bf16ef470c240bc1c5b51fc915b2

HVR-23 Incorrect Addition To Claimed Rewards List Without Vault Verification

Category	Severity	Location	Status
Logical Issue	Minor	sources/reward_distribution.move (reward_distribution): 625~626	Resolved

Description

In the `claim_rewards_process` function, the code adds a coin type to the user's claimed rewards list for a period even if no corresponding vault exists (i.e., when the vault check fails). This occurs regardless of whether rewards are actually withdrawn, potentially polluting the state by marking non-existent rewards as claimed.

```
    public(package) fun claim_rewards_process<CoinType>(
        distributor: &mut RewardDistributor,
        periods: vector<u32>,
        ctx: &mut TxContext
    ) {
        ...
        let coin_type = get_coin_type_name<CoinType>();
        ...
        while (i < periods_len) {
            ...
            let coin_type_key = CoinTypeKey<CoinType> { period };
            let mut already_claimed = false;
            if (table::contains(&distributor.claimed_rewards, period)) {
                let period_claimed = table::borrow(&distributor.claimed_rewards,
period);
                if (table::contains(period_claimed, sender)) {
                    let user_claimed = table::borrow(period_claimed, sender);
                    already_claimed = vector::contains(user_claimed, &coin_type);
                };
            };

            if (!already_claimed) {
                if (vector::contains(period_vaults, &coin_type) &&
                    df::exists_with_type<CoinTypeKey<CoinType>, CoinVault<CoinType>>
(&distributor.id, coin_type_key)) {
                    ...
                };

                // @audit the following logic will continue and process
                if (!table::contains(&distributor.claimed_rewards, period)) {
                    table::add(&mut distributor.claimed_rewards, period,
table::new(ctx));
                };
                let period_claimed = table::borrow_mut(&mut
distributor.claimed_rewards, period);
                if (!table::contains(period_claimed, sender)) {
                    table::add(period_claimed, sender,
vector::singleton(coin_type));
                } else {
                    let user_claimed = table::borrow_mut(period_claimed, sender);
                    vector::push_back(user_claimed, coin_type);
                };
            };
            ...
        };
        ...
    }
```

Additionally, even when a vault exists but has a zero balance, the function still records the reward as claimed. As a result, users may be prevented from re-claiming legitimate rewards later if the vault balance is later replenished.

Recommendation

Ensure that the records are only added to `claimed_rewards` if rewards are successfully processed.

Alleviation

[Haedal, 10/10/2025]: The team heeded the advice and resolved the issue in commit 5d54ee98b028bf16ef470c240bc1c5b51fc915b2

HVR-24 | Missing Check On `pool` And `veheadal`

Category	Severity	Location	Status
Logical Issue	Minor	sources/Staking.move (vehaedal): 221~222, 249~250	Acknowledged

Description

The `extend_existing_lock()` function allows users to extend the lock duration of an existing stake.

However, the function does **not** verify whether the provided `veheadal` object was created within the specified `pool`.

There may be multiple staking pools associated with the same phantom type `<T>`. If the `veheadal` does not belong to that pool, the function will incorrectly update the wrong pool's state.

```
public(package) fun update_lock_period<T>(
    pool: &mut StakingPool<T>,
    original_veheadal_amount: u64,
    new_veheadal_amount: u64,
) {
    pool.total_veheadal = pool.total_veheadal - original_veheadal_amount +
new_veheadal_amount;
}
```

The function directly modifies `pool.total_veheadal` without validating that the `veheadal` corresponds to the given `pool`.

Recommendation

It is recommended to verify that the input `veheadal` corresponds to the specified `pool` before updating the lock period.

Alleviation

[Haedal, 10/10/2025]: Currently, the Vehaedal staking protocol only supports the Haedal single token, and no different coins will be pledged to the protocol in the future.

HVR-26 | RewardDistributor'S Use Of A Vector Field Has An Upper Limit

Category	Severity	Location	Status
Denial of Service	Minor	sources/reward_distribution.move (reward_distribution): 47~53	Acknowledged

Description

In the `RewardDistributor`, the fields `period_vaults`, `supported_coin_types`, `claimed_rewards`, and `period_keys` all use vectors. Vectors have a length limit of 1000, and once this limit is reached, adding new elements will fail, potentially causing a DoS. In particular, the `period_keys` field restricts the reward distribution to a maximum of 1000 periods.

Recommendation

We would like to confirm with the team whether this is the intended design. If not, it will be necessary to upgrade and migrate the `RewardDistributor`.

Alleviation

[Haedal, 10/15/2025]: The team acknowledged the issue and confirmed this is intended.

HVR-27 Functions Using `check_all_coins_claimed()` May Cause DoS Due To High Time Complexity

Category	Severity	Location	Status
Denial of Service	Minor	sources/reward_distribution.move (reward_distribution): 285, 390~397, 665	Resolved

Description

In `check_all_coins_claimed()`, there is a nested loop: the function first iterates over all elements in `period_vaults`, and within this iteration, it loops through all elements in `user_claimed` for checking.

`reward_distribution.move`

```
390         while (i < available_len) {
391             let coin_type = *vector::borrow(available_coins, i);
392             if (!vector::contains(user_claimed, &coin_type)) {
393                 return false
394             };
395
396             i = i + 1;
397         };
```

Furthermore, the function `claim_rewards_process()` that calls it also contains its own loop, resulting in a **triple nested loop** structure.

`reward_distribution.move`

```
605         while (i < periods_len) {
606             ......
607
            let all_claimed =
check_all_coins_claimed(&distributor.supported_coin_types,
&distributor.claimed_rewards, sender, period);
608             if (all_claimed) {
609                 table::remove(period_rewards, sender);
610             };
611
612             i = i + 1;
613         };
```

This leads to excessively high time complexity and increases the risk of a DoS.

A similar issue also exists in `get_unclaimed_periods()`, though it is a read-only function and therefore less severe than `claim_rewards_process()`.

`reward_distribution.move`

```
277          while (i < periods_len) {
278              let period = *vector::borrow(&periods, i);
279
280              if (table::contains(&distributor.period_rewards, period)) {
281
                 let period_rewards = table::borrow(&distributor.period_rewards,
period);
282                  if (table::contains(period_rewards, user)) {
283                      if (table::contains(&distributor.period_vaults, period)) {
284
                        let period_vaults =
table::borrow(&distributor.period_vaults, period);
285
                        let all_claimed =
check_all_coins_claimed(&distributor.supported_coin_types,
&distributor.claimed_rewards, user, period);
286                          if (!all_claimed) {
287                              vector::push_back(&mut result, period);
288                          }
289                      } else {
290                          vector::push_back(&mut result, period);
291                      }
292                  };
293              };
294
295              i = i + 1;
296          };
```

Recommendation

We recommend that the official team review the design of these functions to prevent potential DoS issues caused by high computational complexity.

Alleviation

[Haedal, 10/15/2025]:

We've optimized check_all_coins_claimed , but we believe the outer loop is necessary. This is because the protocol must maintain period_rewards , which directly affects whether users can claim rewards repeatedly.

1. In check_all_coins_claimed , reward currencies will never exceed 5. The original design was for each period to contain only one currency, so the length of available_coins within the period will always be 1. In fact, all future reward currencies will be Haedal, as the protocol's original intention was to reward Haedal stakers.

2. In the outer loop of claim_rewards_process , the length of periods_len depends on the value passed by the caller. On the official website, we limit its length to no more than 50. For calls with multiple periods, we use multiple batches.

3. In extreme cases, if a user goes unclaimed for a long period (a period is 1 week, assuming more than 500 weeks or more than 9 years), get_unclaimed_periods may fail. At this point, the interface will replace get_all_periods to return all periods,

and then initiate transactions in an iterative loop, initiating a transaction every 50 periods.

[Haedal, 10/16/2025]: The team heeded the advice and resolved the issue by simplifying the loop complexity in commit 0119a9fcf683507ab57f8bf1422f21af93587192

HVR-28 `veheadal` 'S `initial_amount` May Exceed `locked_amount` In `extend_existing_lock()`

Category	Severity	Location	Status
Coding Issue	Minor	sources/Staking.move (vehaedal-1010): 243	Resolved

Description

In `extend_existing_lock()`, when `is_decaying == true`, the `additional_weeks` upper limit check is that the **current remaining lock weeks** plus the `additional_weeks` to extend do not exceed `WEEKS_PER_YEAR`.

`Staking.move`

```
 232         let current_lock_weeks = if (!vetoken::is_decaying(veheadal)) {
 233             vetoken::get_remaining_lock_weeks_when_stopped_decay(veheadal)
 234         } else {
 235             let now = sui::clock::timestamp_ms(clock);
 236
            let remaining_lock_time = if (now >=
vetoken::get_lock_end_time(veheadal)) {
 237                 0
 238             } else {
 239
                (vetoken::get_lock_end_time(veheadal) - now) / (DAYS_PER_WEEK *
SECONDS_PER_DAY * 1000)
 240             };
 241             remaining_lock_time
 242         };
 243
@>      assert!(current_lock_weeks + additional_weeks <= WEEKS_PER_YEAR,
EInvalidLockDuration);
```

However, in `vetoken::extend_lock`, when calculating `new_initial_amount`, the calculation uses the **entire lock period** (`lock_end_time - lock_start_time`) plus the `additional_weeks` to determine the new lock period.

`VeToken.move`

```
238          if (!veheadal.is_decaying) {
239
            let remaining_ms = veheadal.remaining_lock_weeks_when_stopped_decay *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000;
240
            new_lock_end_time = now + remaining_ms + (additional_weeks *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000);
241
            new_total_weeks = veheadal.remaining_lock_weeks_when_stopped_decay +
additional_weeks;
242          } else {
243
@>          new_lock_end_time = veheadal.lock_end_time + (additional_weeks *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000);
244  @>           start_time = veheadal.lock_start_time;
245
@>          new_total_weeks = (new_lock_end_time - start_time) / (DAYS_PER_WEEK *
SECONDS_PER_DAY * 1000);
246          };
```

As a result, the corresponding number of weeks for the new lock period may exceed `WEEKS_PER_YEAR`, causing the newly calculated `new_initial_amount` to exceed `locked_amount`.

`VeToken.move`

```
247          // Calculate new initial veHEADAL amount
248
       let new_initial_amount = (veheadal.locked_amount * new_total_weeks) /
WEEKS_PER_YEAR;
249          // Update the token
250          veheadal.initial_amount = new_initial_amount;
```

In other words, the pre-decay `veheadal` amount could exceed the staked token amount, which does not align with the intended design.

Recommendation

In `extend_existing_lock()`, when `is_decaying == true`, the calculation should use the **entire lock period** rather than the current remaining lock weeks as `current_lock_weeks`.

Alleviation

[Haedal, 10/16/2025]: The team heeded the advice and resolved the issue by modifying `new_total_weeks` in `extend_lock()` to be the remaining locked weeks plus `additional_weeks` in commit fc6a033784ef88c5edf266e804566e2fa4060d61

HVR-29 `update_user_stake()` Does Not Update `total_veheadal` Of Pool

Category	Severity	Location	Status
Inconsistency	Minor	sources/StakingPool.move (vehaedal-1010): 215~228	Resolved

Description

In the `add_to_existing_stake()` function, `update_user_stake()` is called to update the corresponding `StakingPool` information. However, after adding a stake in `add_to_existing_stake()`, the amount of the corresponding `veheadal` (i.e., `initial_amount`) also changes.

This new amount is **not reflected in the pool's `total_veheadal` field** via `update_user_stake()`.

`StakingPool.move`

```
215     public(package) fun update_user_stake<T>(
216         pool: &mut StakingPool<T>,
217         veheadal_token: &VeHAEDAL<T>,
218         additional_balance: Balance<T>,
219         additional_amount: u64,
220         _ctx: &TxContext,
221     ) {
222         balance::join(&mut pool.rewards_pool, additional_balance);
223         pool.total_staked = pool.total_staked + additional_amount;
224         let user = vetoken::get_owner(veheadal_token);
225         let token_id = object::id(veheadal_token);
226         let user_info = table::borrow_mut(&mut pool.staking_records, user);
227
       user_staking_info::update_stake_amount<T>(user_info, token_id,
additional_amount);
228     }
```

Recommendation

We recommend updating the `StakingPool` to include the additional `veheadal` amount accordingly.

Alleviation

[Haedal, 10/15/2025]: fix
https://github.com/haedallsd/reward_distribution/commit/0119a9fcf683507ab57f8bf1422f21af93587192

[CertiK, 10/16/2025]: Thank you for your revision and response. However, the updated amount is incorrect. In `add_locked_amount()`, the increase in `vehaedal` is `additional_veheadal`; **`VeToken.move`**

```
261     public(package) fun add_locked_amount<T>(
262         veheadal: &mut VeHAEDAL<T>,
263         additional_amount: u64,
264         clock: &Clock
265     ): u64 {
266         let now = clock::timestamp_ms(clock);
267         let remaining_lock_time = if (now >= veheadal.lock_end_time) {
268             0
269         } else {
270
          (veheadal.lock_end_time - now) / (DAYS_PER_WEEK * SECONDS_PER_DAY *
1000)
271         };
272         veheadal.locked_amount = veheadal.locked_amount + additional_amount;
273
@>      let additional_veheadal = (additional_amount * remaining_lock_time) /
WEEKS_PER_YEAR;
274         let new_initial_amount = veheadal.initial_amount + additional_veheadal;
275         veheadal.initial_amount = new_initial_amount;
276         veheadal.current_amount = get_current_value(veheadal, clock);
277         new_initial_amount
278     }
```

however, in `update_user_stake()`, `additional_amount` is directly added to `pool.total_veheadal`, which is incorrect.

`StakingPool.move`

```
215     public(package) fun update_user_stake<T>(
216         pool: &mut StakingPool<T>,
217         veheadal_token: &VeHAEDAL<T>,
218         additional_balance: Balance<T>,
219         additional_amount: u64,
220         _ctx: &TxContext,
221     ) {
222         balance::join(&mut pool.rewards_pool, additional_balance);
223         pool.total_staked = pool.total_staked + additional_amount;
224         pool.total_veheadal = pool.total_veheadal + additional_amount;
225         let user = vetoken::get_owner(veheadal_token);
226         let token_id = object::id(veheadal_token);
227         let user_info = table::borrow_mut(&mut pool.staking_records, user);
228
        user_staking_info::update_stake_amount<T>(user_info, token_id,
additional_amount);
229     }
```

[Haedal, 10/20/2025]: The team heeded the advice and resolved the issue by updating the `total_veheadal` in commit 43dbc3d9b6c5f04014dffb7b7b5d74d030092e7c.

HVR-34 | Incorrect `total_locked` Update In `add_to_existing_stake`

Category	Severity	Location	Status
Volatile Code	Minor	sources/StakingPool.move (vehaedal-1018): 230~231; sources/UserStakingInfo.move (vehaedal-1018): 75~76	Resolved

Description

The `add_to_existing_stake()` function incorrectly updates the `UserStakingInfo.total_locked` field using a **ve-token amount** instead of the actual **staked coin amount**.

In `add_to_existing_stake()`, the function calls:

```
staking_pool::update_user_stake(..., additional_veheadal);
```

which internally results in:

```
user_info.total_locked = user_info.total_locked + additional_amount;
```

Here, `additional_amount` corresponds to `additional_veheadal`, derived from:

```
let additional_veheadal = vetoken::get_add_vehaedal_amount(veheadal, amount, clock);
```

This value represents the **additional ve-token** generated, not the actual staked coin amount.

Using the ve-token value to update `total_locked` results in inaccurate accounting of the user's staked balance, which may cause significant inconsistencies between the recorded stake data and the actual staked amount.

Recommendation

It is recommended to pass the correct input coin amount when updating `total_locked`.

Alleviation

[Haedal, 10/22/2025]: The team heeded the advice and resolved the issue by using `staked_amount` in `user_staking_info::update_stake_amount()` in commit ddac75540cf59304f9d2550fc00bc95cb1bbadc5

HVR-02 | Potential Inconsistent Minimum Stake Amount Check

Category	Severity	Location	Status
Volatile Code	Informational	sources/Staking.move (vehaedal): 177~178	Resolved

Description

The `add_stake()` function enforces a minimum stake requirement when users create a new stake:

```
if (min_amount > 0) {
    assert!(amount >= min_amount, EMinStakeAmount);
};
```

In contrast, the `add_to_existing_stake()` function, which allows users to increase an existing stake, does **not** enforce this minimum stake amount check. As a result, additional deposits may be smaller than the minimum threshold required for new stakes.

Recommendation

We recommend discussing this behavior with the development team to confirm whether the absence of a minimum stake check in `add_to_existing_stake()` is by design. If it is not intended, align the logic between both functions by enforcing the same minimum stake requirement.

Alleviation

[Haedal, 10/10/2025]: The agreement stipulates this. Since the minimum amount for new holdings is already specified, the increase in holdings can only be greater than the minimum amount, not less, so it is meaningless to increase the minimum amount verification.

HVR-10 Potential Unused `pending_rewards` And Incomplete Reward Logic

Category	Severity	Location	Status
Logical Issue	Informational	sources/UserStakingInfo.move (vehaedal): 11~12	Resolved

Description

There is a `pending_rewards` field in the `UserStakingInfo` struct, which is reset to zero whenever a user creates a new stake. However, this field is never calculated or referenced anywhere else in the project. Additionally, when a user calls the `unstake_and_claim()` function, it only returns the originally staked coins, it neither computes nor distributes any rewards. `rewards_pool` currently acts as a principal vault (stake in, split out on unstake), not a reward pool.

Recommendation

We would like to discuss this finding with the team to confirm whether this behavior is intentional and consistent with the intended business logic.

Alleviation

[Haedal, 10/22/2025]: Conforms to expected business logic

[CertiK, 10/22/2025]: Thanks for your response. We'd like to confirm whether it's intentional and consistent with the business logic that the `total_staked` field in `UserStakingInfo` is incremented in both `add_stake()` and `update_stake_amount()`, but not decremented in `remove_stake()`. If this behavior is not intended, we recommend updating the `remove_stake()` function to decrease the `total_staked` value to maintain data consistency.

[Haedal, 10/23/2025]: The team heeded the advice and resolved the issue in commit b6eccacfbb81434cbc99165c3820128270657764.

HVR-11 | Unused Functions

Category	Severity	Location	Status
Logical Issue	Informational	sources/StakingPool.move (vehaedal): 136~137, 226~227, 238~239	Resolved

Description

The following functions are declared with the `public(package)` visibility modifier, which restricts their usage to within the same package:

- `unstake`
- `claim_unstaked_tokens`
- `claim_unstaked_tokens_by_id`

Similar issue also hold for the following functions:

- `grant_admin_cap`
- `create_or_update_staking_info`
- `update_current_amount`

These functions appear to be related to the unstaking process. If they are intended to be integrated into the unstake workflow in the future, we recommend completing their implementation before the on-chain deployment. However, if these functions are not part of the future roadmap, we suggest removing them to improve code clarity and maintainability.

Recommendation

These functions appear to be related to the unstaking process. If they are intended to be integrated into the unstake workflow in the future, we recommend completing their implementation before the on-chain deployment. However, if these functions are not part of the future roadmap, we suggest removing them to improve code clarity and maintainability.

Alleviation

[Haedal, 10/10/2025]: The team heeded the advice and resolved the issue by removing unused functions in commit 291cbff9265727ec0b6a0ca58a946bce36968eb4.

HVR-12 | Potential Inconsistent Scale

Category	Severity	Location	Status
Volatile Code	Informational	sources/reward_distribution.move (reward_distribution): 105~106	Acknowledged

Description

When adding rewards, the given percentage for each entry is using PERCENTAGE_PRECISION:

```
        let mut i = 0;
        while (i < users_len) {
            let user = *vector::borrow(&users, i);
            let percentage = *vector::borrow(&percentages, i);
            if (table::contains(period_rewards, user)) {
                let old_percentage = *table::borrow(period_rewards, user);
                total_percentage = total_percentage - old_percentage;
            };

            total_percentage = total_percentage + percentage;
            assert!(total_percentage <= PERCENTAGE_PRECISION,
ETotalPercentageExceeded);
```

However, when claiming the reward, the `period<=2` and `period > 2` are sharing different precision; if the percentages are set unexpectedly, the user may get a 1000 times reward increase.

```
                    let percentages = if (period <= 2) {
                        PERCENTAGE_PRECISION_OLD
                    } else {
                        PERCENTAGE_PRECISION
                    };
                    let reward_amount_u128 = ((coin_vault.total_rewards as u128) *
(user_percentage as u128)) / (percentages as u128);
                    let mut reward_amount = (reward_amount_u128 as u64);
                    if (reward_amount > 0) {
                        if (reward_amount >= vault::vault_amount(&coin_vault.vault))
{
                            reward_amount = vault::vault_amount(&coin_vault.vault);
                        };
```

Recommendation

We would like to check with the team if the scenario has been considered.

Alleviation

[Haedal, 10/10/2025]: This is a known issue and is intentional. It is caused by the upgrade. The accuracy of period 1 and week 2 is different from the accuracy of all subsequent periods.

HVR-13 | Concerns When Multiple `PauseClaimConfig` / `ACL` Exists

Category	Severity	Location	Status
Volatile Code	Informational	sources/reward_distribution.move (reward_distribution): 127, 135; sources/Admin.move (vehaedal): 35~40	Acknowledged

Description

Multiple `PauseClaimConfig` can be created with `AdminCap` in `init_pause_reward`, if the records are not synced between different `PauseClaimConfig`, the pause mechanism can be bypassed.

The similar concern also raised for ACL.

Recommendation

We would like to check with the team if the scenario has been considered.

Alleviation

[Haedal, 10/10/2025]: Due to AdminCap's control (each call requires at least 4 administrators to sign multiple times), there will not be multiple PauseClaimConfig/ACL.

HVR-14 User Cannot Claim Additional Reward Deposits After Being Removed From `period_rewards`

Category	Severity	Location	Status
Design Issue	Informational	sources/reward_distribution.move (reward_distribution): 657 ~663, 667	Acknowledged

Description

In the `claim_rewards_process` function, after a user claims all available coins for a given period, they are removed from the `period_rewards` table if `check_all_coins_claimed` returns `true`. However, if an operator subsequently deposits additional rewards for the same period via `deposit_reward_process` (either increasing the total for an existing coin type or adding a new coin type), the user is no longer eligible to claim these new rewards because they have been removed from `period_rewards`.

`reward_distribution.move`

```
665
            let all_claimed =
check_all_coins_claimed(&distributor.supported_coin_types,
&distributor.claimed_rewards, sender, period);
666             if (all_claimed) {
667                 table::remove(period_rewards, sender);
668             };
```

In addition, after claiming the tokens, the corresponding `coin_type` will be added to the user's `distributor.claimed_rewards`. If additional rewards of that token type are added to the same period afterward, the user will no longer be able to claim them.

`reward_distribution.move`

```
657
                let period_claimed = table::borrow_mut(&mut
distributor.claimed_rewards, period);
658                 if (!table::contains(period_claimed, sender)) {
659
                    table::add(period_claimed, sender,
vector::singleton(coin_type));
660                 } else {
661
                    let user_claimed = table::borrow_mut(period_claimed, sender);
662                     vector::push_back(user_claimed, coin_type);
663                 };
```

Recommendation

We would like to know if this is the intended design.

Alleviation

[Haedal, 10/10/2025]:

1. The entire staking agreement will stipulate that rewards will only be issued once per cycle, and rewards will not be issued repeatedly for previous cycles.

2. If multiple rewards are available in a cycle, the operator will deposit multiple reward tokens simultaneously, rather than depositing rewards for previous cycles.

HVR-15 | `extend_lock()` May Cause Unintended Value Jump

Category	Severity	Location	Status
Logical Issue	Informational	sources/VeToken.move (vehaedal): 229~230	Acknowledged

Description

The `extend_lock()` function directly resets `current_amount` to `new_initial_amount`:

```
// Update the token
veheadal.initial_amount = new_initial_amount;
veheadal.current_amount = new_initial_amount; // Reset to new initial amount
```

According to the logic in `add_locked_amount()`, the `current_amount` should **decrease over time** as defined by:

```
veheadal.current_amount = get_current_value(veheadal, clock);
```

In `get_current_value()`, the current value is calculated based on the remaining lock time:

```
// Calculate current value based on remaining time
let current_value_u128 =
    ((veheadal.initial_amount as u128) * (remaining_time_percentage as u128))
        / (PERCENTAGE_BASE as u128);
```

However, under the current **decay model**, the implementation in `extend_lock()` resets `current_amount` to the full initial value, effectively causing a **"ve boost"**. This sudden jump is especially pronounced when the lock is extended near the end of its original period.

Recommendation

We recommend discussing this finding with the team to confirm that the current implementation aligns with the intended business logic and design objectives.

Alleviation

[Haedal, 10/10/2025]: This issue is the same as issue HVR-04. Regarding initial_amount, this is a redundant, legacy field.

In fact, the latest logic does not use numerical calculations related to initial_amount.

In subsequent upgrades, we will consider optimizing the redundant fields and functions related to get_current_value and initial_amount. This will be implemented in the near future.

HVR-16 | Period Mutations After Claims May Cause Unfair Payouts

Category	Severity	Location	Status
Volatile Code	Informational	sources/reward_distribution.move (reward_distribution): 490, 539	Resolved

Description

According to the current implementation, both `deposit_reward_process` and `add_user_rewards_process` functions allow operator/breaker/robot to process them even if the user has already started the claim.

As a result, if the period rewards/vault has been updated after the user's claim, the user will not be able to claim newly added rewards.

Recommendation

We would like to confirm with the team whether this situation can occur, as well as request the related process documentation.

Alleviation

[Haedal, 10/10/2025]: In fact, we will transfer OperatorCap to the black hole address later, so as to distinguish the breaker operation deposit_reward and the robot operation add_user_rewards

[CertiK, 10/10/2025]: Thank you for the update. We would like to confirm with the team that the `deposit_reward_process` will not be triggered again after the `add_user_rewards_process` has been triggered for the same period.

[Haedal, 10/17/2025]: Yeah, that's not going to happen based on current management.

HVR-17 | Discussion On `Cap`

Category	Severity	Location	Status
Design Issue	Informational	sources/reward_distribution.move (reward_distribution): 17 ~25	Acknowledged

Description

`AdminCap` and `OperatorCap` have the `key` and `store` abilities, allowing them to be directly `public_transfer`ed by the holding account to other accounts. This means the granting and revocation of these privileged roles are not determined by the `AdminCap` holder but instead by the respective holders themselves. This would result in these roles not being managed or controlled by the `AdminCap` holder. We would like to ask whether this is the intended design.

```
/// `AdminCap` is used by an administrator.
public struct AdminCap has store, key {
    id: UID,
}

/// `OperatorCap` is used by the offchain programs.
public struct OperatorCap has store, key {
    id: UID,
}
```

Recommendation

We would like to ask whether this is the intended design.

Alleviation

[CertiK, 10/10/2025]: Thanks for the update. The team mentioned that the OperatorCap will be transferred to a black hole address. However, the current functions admin_withdraw_unclaimed_rewards, add_user_rewards, and deposit_reward still rely on the OperatorCap.

[Haedal, 10/10/2025]:

As expected, AdminCap is a multi-signature wallet address, which requires the consent of at least 4 administrators to execute, and OperatorCap will consider transferring to a black hole address.

[Haedal, 10/12/2025]: Thank you for your response.

Yes, the team will do that.

[Haedal, 10/13/2025]: This time, no repair will be made; it will be finalized directly.

HVR-18 | Discussion On Version Control

Category	Severity	Location	Status
Logical Issue	Informational	sources/reward_distribution.move (reward_distribution): 1~2	Acknowledged

Description

In the `reward_distribution` module, the original `claim_rewards()` function has been deprecated (its body now contains only an `abort` statement) and refactored into a new function, `claim_rewards_v2()`. Additionally, the constant `PERCENTAGE_PRECISION` has two versions:

```
const PERCENTAGE_PRECISION: u64 = 1000000000; // 100% after cycle 2
const PERCENTAGE_PRECISION_OLD: u64 = 1000000; // 100% (old) before cycle 2
```

It appears that this project is an upgraded version. However, none of the functions in the module enforce version checking or control logic. As a result, functions from the deprecated version can still be invoked, potentially leading to inconsistent or unexpected behavior.

Due to the upgrade mechanism on Sui, the older version will remain on the chain and can be used in the future. It is recommended to implement a version control mechanism across all functions at the **very first** version to ensure that only the appropriate versioned logic is executed, preventing unintended cross-version interactions.

Recommendation

We would like to check with the team if the scenario has been considered.

Alleviation

[Haedal, 10/12/2025]:

Yes, since the contract has been applied to the main network, we will consider re-releasing the contract if necessary, and add version control that complies with the SUI upgrade mechanism.

[CertiK, 10/10/2025]: Thanks for the update. We would like to remind the team that none of the functions in the module enforce version checking or control logic. As a result, functions from the deprecated version can still be invoked, potentially leading to inconsistent or unexpected behavior.

Due to the upgrade mechanism on Sui, the older version will remain on the chain and can be used in the future. It is recommended to implement a version control mechanism across all functions at the very first version to ensure that only the appropriate versioned logic is executed, preventing unintended cross-version interactions.

[Haedal, 10/13/2025]: Thank you for your response

Yes, since the contract has been applied to the main network, we will consider re-releasing the contract if necessary, and add version control that complies with the SUI upgrade mechanism.

[Haedal, 10/13/2025]: This time, no repair will be made; it will be finalized directly.

HVR-25 | Operator Withdrawal Before User Claims Causes Reward Loss

Category	Severity	Location	Status
Logical Issue	Informational	sources/reward_distribution.move (reward_distribution): 186~187	Acknowledged

Description

The `admin_withdraw_unclaimed_rewards()` function allows an `OperatorCap` holder to **withdraw all remaining funds** from the vault intended for user reward distribution.

If this operation is executed **after** `deposit_reward()` but **before some users call** `claim_rewards_v2()`, it creates a scenario:

- Eligible users will still be marked as having claimed their rewards (as indicated in the `claim_rewards_v2()` logic),
- But the vault will have a **zero balance**, meaning those users **receive no actual reward**.

As a result, users who were legitimately entitled to rewards may permanently lose them due to administrative withdrawal timing.

Recommendation

It is recommended to enforce a restriction that `admin_withdraw_unclaimed_rewards()` may only be executed once all users have completed their reward claims.

Alleviation

[Haedal, 10/10/2025]:

1. In reality, the admin_withdraw_unclaimed_rewards function is only called in emergency situations to redeem deposited rewards. If all users have claimed their rewards, the reward pool will be zero, which contradicts the function's original design and is not the intended functionality.

2. The OperatorCap in the admin_withdraw_unclaimed_rewards function is multi-signed by four or more administrators and cannot be called maliciously. It is only used to withdraw rewards in emergencies. In such situations, this function is used in conjunction with pause_claim_config.pause_claim, so that user unclaimed and claimed rewards are logged to facilitate user data recovery.

HVR-30 `original_lock_weeks` May Be Incorrectly Modified In `extend_lock()`

Category	Severity	Location	Status
Coding Issue	Informational	sources/VeToken.move (vehaedal-1010): 253	Resolved

Description

In `extend_lock()`, if the `veheadal` has ever called `stop_decay()` (regardless of the current `is_decaying` status), the `original_lock_weeks` will be updated to `remaining_lock_weeks_when_stopped_decay + additional_weeks` instead of `original_lock_weeks + additional_weeks`. This behavior does not align with the field meaning of the `original_lock_weeks`.

`VeToken.move`

```
229      public(package) fun extend_lock<T>(
230          veheadal: &mut VeHAEDAL<T>,
231          additional_weeks: u64,
232          clock: &Clock
233      ): u64 {
234          let now = clock::timestamp_ms(clock);
235          let new_lock_end_time;
236          let start_time;
237          let new_total_weeks;
238          if (!veheadal.is_decaying) {
239
            let remaining_ms = veheadal.remaining_lock_weeks_when_stopped_decay *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000;
240
            new_lock_end_time = now + remaining_ms + (additional_weeks *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000);
241
@>          new_total_weeks = veheadal.remaining_lock_weeks_when_stopped_decay +
additional_weeks;
242          } else {
243
            new_lock_end_time = veheadal.lock_end_time + (additional_weeks *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000);
244              start_time = veheadal.lock_start_time;
245
@>          new_total_weeks = (new_lock_end_time - start_time) / (DAYS_PER_WEEK *
SECONDS_PER_DAY * 1000);
246          };
247          // Calculate new initial veHEADAL amount
248
        let new_initial_amount = (veheadal.locked_amount * new_total_weeks) /
WEEKS_PER_YEAR;
249          // Update the token
250          veheadal.initial_amount = new_initial_amount;
251
        veheadal.current_amount = new_initial_amount; // Reset to new initial amount
252          veheadal.lock_end_time = new_lock_end_time;
253  @>      veheadal.original_lock_weeks = new_total_weeks;
254          if (!veheadal.is_decaying) {
255              veheadal.remaining_lock_weeks_when_stopped_decay = new_total_weeks;
256          };
257          // Return new amount
258          new_initial_amount
259      }
```

Recommendation

We would like to confirm whether this is the intended design.

Alleviation

[Haedal, 10/15/2025]: fix https://github.com/haedallsd/reward_distribution/commit/0119a9fcf683507ab57f8bf1422f21af93587192

[CertiK, 10/16/2025]: Thank you for your reply. Please carefully check the links and updates you provided in your response.

[Haedal, 10/17/2025]: fix new commit https://github.com/haedallsd/vehaedal/commit/43dbc3d9b6c5f04014dffb7b7b5d74d030092e7c

---

[CertiK, 10/16/2025]: Thank you for your reply. We have not yet seen the fix in the given commit. Could you please share more insight?

[Haedal, 10/21/2025]: The latest update may not be in the same commit, but the code update has been uploaded. Please pull the latest code and check the branch in the extend_lock function where veheadal.is_decaying==true. We have updated new_total_weeks to the correct latest calculation method.

[CertiK, 10/22/2025]: Thank you for your reply. In the current update, `veheadal.original_lock_weeks` is still being updated to remaining time adding `additional_weeks`. We would like to confirm whether this is the intended design.

`VeToken.move`

```
217     public(package) fun extend_lock<T>(
218         veheadal: &mut VeHAEDAL<T>,
219         additional_weeks: u64,
220         clock: &Clock
221     ): u64 {
222         let now = clock::timestamp_ms(clock);
223         let new_lock_end_time;
224         let start_time;
225         let new_total_weeks;
226         if (!veheadal.is_decaying) {
227
           let remaining_ms = veheadal.remaining_lock_weeks_when_stopped_decay *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000;
228
           new_lock_end_time = now + remaining_ms + (additional_weeks *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000);
229
           new_total_weeks = veheadal.remaining_lock_weeks_when_stopped_decay +
additional_weeks;
230         } else {
231
           new_lock_end_time = veheadal.lock_end_time + (additional_weeks *
DAYS_PER_WEEK * SECONDS_PER_DAY * 1000);
232
           new_total_weeks = (new_lock_end_time - now) / (DAYS_PER_WEEK *
SECONDS_PER_DAY * 1000);
233         };
234         // Calculate new initial veHEADAL amount
235
       let new_initial_amount = (veheadal.locked_amount * new_total_weeks) /
WEEKS_PER_YEAR;
236         // Update the token
237         veheadal.initial_amount = new_initial_amount;
238
       veheadal.current_amount = new_initial_amount; // Reset to new initial amount
239         veheadal.lock_end_time = new_lock_end_time;
240         veheadal.original_lock_weeks = new_total_weeks;
241         if (!veheadal.is_decaying) {
242             veheadal.remaining_lock_weeks_when_stopped_decay = new_total_weeks;
243         };
244         // Return new amount
245         new_initial_amount
246     }
```

[Haedal, 10/23/2025]: Thank you for your response.

First, this is in line with our expectations.

Second, I'll explain the reasoning in detail: Initially, using start_time for calculations will cause the new_total_weeks value to be incorrectly calculated (most likely exceeding 52) when veheadal.is_decaying == true. However, our primary goal is to obtain the latest remaining time, not the cumulative time since the initial lockout. Therefore, we need to use the formula ((stake expiration time + latest extension time) - current timestamp) to obtain the number of periods after the extension. This

new_total_weeks value is then used in the subsequent calculations of new_initial_amount and original_lock_weeks. The variable naming is certainly misleading. However, given the context, we're looking for the latest period, not the cumulative time since the initial lockout.

HVR-31 | Event-Emitting Functions Callable By Anyone

Category	Severity	Location	Status
Access Control	Informational	sources/Events.move (vehaedal-1010): 1	Acknowledged

Description

All events can be triggered by anyone. If any program relies on these events, an attacker could directly call the corresponding event-emitting functions to emit events, introducing potential risks to the downstream programs.

Recommendation

We recommend that the team either avoid relying on events in the relevant programs or change these event functions to `public (package)`.

Alleviation

[Haedal, 10/15/2025]:

We rely on events, but we do not recognize records generated by directly calling event functions. The program will exclude this situation.

HVR-32 | Potential Inconsistent Logic And Error Code

Category	Severity	Location	Status
Volatile Code	Informational	sources/Staking.move (vehaedal-1014): 201~203	Resolved

Description

The error code `ELockShouldBeExpired` (line 17) has a misleading name that contradicts its usage.

- The assertion checks: `!vetoken::is_expired()` - meaning the lock should **NOT** be expired
- The error name `ELockShouldBeExpired` suggests "the lock should be expired (but isn't)"
- However, the error is thrown when the lock **IS** expired (when it shouldn't be)
- This creates confusion about what the actual error condition is

```
const ELockShouldBeExpired: u64 = 6;  // Line 17

// Usage in three locations:
// Line 109 (stop_decay)
assert!(!vetoken::is_expired(veheadal, clock), ELockShouldBeExpired);

// Line 202 (add_to_existing_stake)
if (vetoken::is_decaying(veheadal)) {
    assert!(!vetoken::is_expired(veheadal, clock), ELockShouldBeExpired);
};

// Line 245 (extend_existing_lock)
if (vetoken::is_decaying(veheadal)) {
    assert!(!vetoken::is_expired(veheadal, clock), ELockShouldBeExpired);
};
```

Recommendation

Recommend updating the error code.

Alleviation

[Haedal, 10/19/2025]: The team heeded the advice and resolved the issue by updating the error code in commit 43dbc3d9b6c5f04014dffb7b7b5d74d030092e7c

OPTIMIZATIONS | HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

ID	Title	Category	Severity	Status
HVR-08	Unreachable Branch	Coding Style	Optimization	Acknowledged
HVR-09	Unused ACL Member	Coding Style	Optimization	Acknowledged
HVR-35	Inconsistent Code And Comment	Volatile Code	Optimization	Resolved

HVR-08 | Unreachable Branch

Category	Severity	Location	Status
Coding Style	Optimization	sources/Staking.move (vehaedal): 111~112, 193~195	Acknowledged

Description

The `stop_decay()` function allows a user to disable the decay mode for their own `veHEADAL` token. The function enforces that the current time **has not yet reached the lock end time**, as shown below:

```
assert!(!vetoken::is_expired(veheadal, clock), ELockShouldBeExpired);
```

The `is_expired()` function determines whether the lock period has ended:

```
public fun is_expired<T>(veheadal: &VeHAEDAL<T>, clock: &Clock): bool {
    clock::timestamp_ms(clock) >= veheadal.lock_end_time
}
```

This means that if the assertion passes, the lock has **not expired yet**, so the following conditional branch will **never be executed**:

```
let remaining_lock_time = if (now >= vetoken::get_lock_end_time(veheadal)) {
    0
}
```

In other words, because `stop_decay()` only runs when the lock is still active, the code path handling the `lock expired` case cannot be reached.

A similar issue exists in the `add_to_existing_stake()` function. The following branch is unreachable because the assertion below ensures that, in decay mode, the current time **has not yet reached the lock end time**:

```
let remaining_lock_time = if (now >= vetoken::get_lock_end_time(veheadal)) {
    0
}
```

```
if (vetoken::is_decaying(veheadal)) {
    assert!(!vetoken::is_expired(veheadal, clock), ELockShouldBeExpired);
};
```

Recommendation

It is recommended to refactor the function to remove or simplify this redundant conditional branch in order to improve code efficiency and maintainability.

Alleviation

[Haedal, 10/10/2025]: The team acknowledged the issue and decided not to implement the recommended change in the current engagement.

HVR-09 | Unused ACL Member

Category	Severity	Location	Status
Coding Style	Optimization	sources/reward_distribution.move (reward_distribution): 138~139; sources/Admin.move (vehaedal): 38~39	Acknowledged

Description

In **reward_distribution**,the `ACL` structure allows the `AdminCap` to manage multiple privileged roles. However, the `minor_signs` field is **initialized but never used** anywhere in the project.

```
let acl = ACL {
    id: object::new(ctx),
    minor_signs: vector::empty<address>(),
    breakers: vector::empty<address>(),
    robots: vector::empty<address>(),
};
```

The same situation also occurs in **vehaedal**, where `Admin.move` similarly initializes `breakers` and `robots`, but they are not actually used in the project.

```
    public entry fun share_acl(_: &AdminCap, ctx: &mut TxContext) {
        let acl = ACL {
            id: object::new(ctx),
            minor_signs: vector::empty<address>(),
            breakers: vector::empty<address>(),
            robots: vector::empty<address>(),
        };
        transfer::share_object<ACL>(acl);
    }
```

This suggests that the field may be **a remnant of an incomplete or deprecated feature** intended for additional role-based access control. Since it is never read or modified, it adds unnecessary complexity and may confuse future maintainers about its purpose.

Recommendation

If the roles are not part of the current access control design, remove the field entirely to simplify the ACL definition. If it is planned for future use, document its intended purpose and ensure corresponding logic (e.g., validation, permission checks, or management functions) is implemented.

Alleviation

[Haedal, 10/10/2025]: Maybe minor_signs will be used in the future.

[CertiK, 10/13/2025]: Thank you for your reply. We would like to ask whether the `breakers` and `robots` in **vehaedal** are in the same situation as well?

HVR-35 | Inconsistent Code And Comment

Category	Severity	Location	Status
Volatile Code	Optimization	sources/VeToken.move (vehaedal-1018): 118~119	Resolved

Description

The code and comment are inconsistent, with unclear semantics and naming.

- The comment mentions checking expiry “after midnight of the day after staking starts,” but the code uses `lock_end_time` instead of the staking start time.
- The logic actually checks if the current time is after midnight of the day containing `lock_end_time`, which can mark it expired up to 24 hours early.
- The naming `veHEADAL` vs. `VeHAEDAL<T>` is inconsistent.

Recommendation

It is recommended to revise either the comment or the code to ensure they are consistent with each other.

Alleviation

[Haedal, 10/21/2025]:

The team heeded the advice and resolved the issue in commit b2fa4d232eaa02d22daa952f0d3710ea6fb5cbee.

APPENDIX | HAEDAL - VEHAEDAL & REWARD DISTRIBUTION

Audit Scope

haedallsd/vehaedal
sources/Admin.move
sources/Staking.move
sources/VeToken.move
sources/Events.move
sources/StakingPool.move
sources/UserStakingInfo.move
sources/Staking.move
sources/StakingPool.move
sources/VeToken.move
sources/Staking.move
sources/StakingPool.move
sources/UserStakingInfo.move
sources/VeToken.move
sources/Events.move
sources/Version.move
sources/Admin.move
sources/UserStakingInfo.move
sources/Version.move
sources/Admin.move

haedallsd/vehaedal
sources/Events.move
sources/StakingPool.move
sources/UserStakingInfo.move
sources/VeToken.move
sources/Version.move
sources/Admin.move
sources/Events.move
sources/Staking.move
sources/Version.move
sources/Admin.move
sources/Events.move
sources/Staking.move
sources/StakingPool.move
sources/UserStakingInfo.move
sources/VeToken.move
sources/Version.move
sources/Admin.move
sources/Events.move
sources/Staking.move
sources/StakingPool.move
sources/UserStakingInfo.move
sources/VeToken.move

haedallsd/vehaedal
sources/Version.move

haedallsd/reward_distribution
sources/reward_distribution.move
sources/breaker.move
sources/robot.move
sources/vault.move
sources/breaker.move
sources/reward_distribution.move
sources/robot.move
sources/vault.move
sources/breaker.move
sources/reward_distribution.move
sources/robot.move
sources/vault.move

Finding Categories

Categories	Description
Coding Style	Coding Style findings may not affect code behavior, but indicate areas where coding practices can be improved to make the code more understandable and maintainable.
Coding Issue	Coding Issue findings are about general code quality including, but not limited to, coding mistakes, compile errors, and performance issues.
Incorrect Calculation	Incorrect Calculation findings are about issues in numeric computation such as rounding errors, overflows, out-of-bounds and any computation that is not intended.

Categories	Description
Denial of Service	Denial of Service findings indicate that an attacker may prevent the program from operating correctly or responding to legitimate requests.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
Inconsistency	Inconsistency findings refer to different parts of code that are not consistent or code that does not behave according to its specification.
Volatile Code	Volatile Code findings refer to segments of code that behave unexpectedly on certain edge cases and may result in vulnerabilities.
Logical Issue	Logical Issue findings indicate general implementation issues related to the program logic.
Centralization	Centralization findings detail the design choices of designating privileged roles or other centralized controls over the code.
Design Issue	Design Issue findings indicate general issues at the design level beyond program logic that are not covered by other finding categories.

DISCLAIMER | CERTIK

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability) set forth in the Services Agreement, or the scope of services, and terms and conditions provided to you ("Customer" or the "Company") in connection with the Agreement. This report provided in connection with the Services set forth in the Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement. This report may not be transmitted, disclosed, referred to or relied upon by any person for any purposes, nor may copies be delivered to any other person other than the Company, without CertiK's prior written consent in each instance.

This report is not, nor should be considered, an "endorsement" or "disapproval" of any particular project or team. This report is not, nor should be considered, an indication of the economics or value of any "product" or "asset" created by any team or project that contracts CertiK to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor do they provide any indication of the technologies proprietors, business, business model or legal compliance.

This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should be leveraged as investment advice of any sort. This report represents an extensive assessing process intending to help our customers increase the quality of their code while reducing the high level of risk presented by cryptographic tokens and blockchain technology.

Blockchain technology and cryptographic assets present a high level of ongoing risk. CertiK's position is that each company and individual are responsible for their own due diligence and continuous security. CertiK's goal is to help reduce the attack vectors and the high level of variance associated with utilizing new and consistently changing technologies, and in no way claims any guarantee of security or functionality of the technology we agree to analyze.

The assessment services provided by CertiK is subject to dependencies and under continuing development. You agree that your access and/or use, including but not limited to any services, reports, and materials, will be at your sole risk on an as-is, where-is, and as-available basis. Cryptographic tokens are emergent technologies and carry with them high levels of technical risk and uncertainty. The assessment reports could include false positives, false negatives, and other unpredictable results. The services may access, and depend upon, multiple layers of third-parties.

ALL SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF ARE PROVIDED "AS IS" AND "AS AVAILABLE" AND WITH ALL FAULTS AND DEFECTS WITHOUT WARRANTY OF ANY KIND. TO THE MAXIMUM EXTENT PERMITTED UNDER APPLICABLE LAW, CERTIK HEREBY DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED, STATUTORY, OR OTHERWISE WITH RESPECT TO THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS. WITHOUT LIMITING THE FOREGOING, CERTIK SPECIFICALLY DISCLAIMS ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT, AND ALL WARRANTIES ARISING FROM COURSE OF DEALING, USAGE, OR TRADE PRACTICE. WITHOUT LIMITING THE FOREGOING, CERTIK MAKES NO WARRANTY OF ANY KIND THAT THE SERVICES, THE LABELS, THE ASSESSMENT REPORT, WORK PRODUCT, OR OTHER MATERIALS, OR ANY PRODUCTS OR RESULTS OF THE USE THEREOF, WILL MEET CUSTOMER'S OR ANY OTHER PERSON'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULT, BE COMPATIBLE OR WORK WITH ANY SOFTWARE, SYSTEM, OR OTHER SERVICES, OR BE SECURE, ACCURATE, COMPLETE, FREE OF HARMFUL CODE, OR ERROR-FREE. WITHOUT LIMITATION TO THE FOREGOING, CERTIK PROVIDES NO WARRANTY OR

UNDERTAKING, AND MAKES NO REPRESENTATION OF ANY KIND THAT THE SERVICE WILL MEET CUSTOMER'S REQUIREMENTS, ACHIEVE ANY INTENDED RESULTS, BE COMPATIBLE OR WORK WITH ANY OTHER SOFTWARE, APPLICATIONS, SYSTEMS OR SERVICES, OPERATE WITHOUT INTERRUPTION, MEET ANY PERFORMANCE OR RELIABILITY STANDARDS OR BE ERROR FREE OR THAT ANY ERRORS OR DEFECTS CAN OR WILL BE CORRECTED.

WITHOUT LIMITING THE FOREGOING, NEITHER CERTIK NOR ANY OF CERTIK'S AGENTS MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND, EXPRESS OR IMPLIED AS TO THE ACCURACY, RELIABILITY, OR CURRENCY OF ANY INFORMATION OR CONTENT PROVIDED THROUGH THE SERVICE. CERTIK WILL ASSUME NO LIABILITY OR RESPONSIBILITY FOR (I) ANY ERRORS, MISTAKES, OR INACCURACIES OF CONTENT AND MATERIALS OR FOR ANY LOSS OR DAMAGE OF ANY KIND INCURRED AS A RESULT OF THE USE OF ANY CONTENT, OR (II) ANY PERSONAL INJURY OR PROPERTY DAMAGE, OF ANY NATURE WHATSOEVER, RESULTING FROM CUSTOMER'S ACCESS TO OR USE OF THE SERVICES, ASSESSMENT REPORT, OR OTHER MATERIALS.

ALL THIRD-PARTY MATERIALS ARE PROVIDED "AS IS" AND ANY REPRESENTATION OR WARRANTY OF OR CONCERNING ANY THIRD-PARTY MATERIALS IS STRICTLY BETWEEN CUSTOMER AND THE THIRD-PARTY OWNER OR DISTRIBUTOR OF THE THIRD-PARTY MATERIALS.

THE SERVICES, ASSESSMENT REPORT, AND ANY OTHER MATERIALS HEREUNDER ARE SOLELY PROVIDED TO CUSTOMER AND MAY NOT BE RELIED ON BY ANY OTHER PERSON OR FOR ANY PURPOSE NOT SPECIFICALLY IDENTIFIED IN THIS AGREEMENT, NOR MAY COPIES BE DELIVERED TO, ANY OTHER PERSON WITHOUT CERTIK'S PRIOR WRITTEN CONSENT IN EACH INSTANCE.

NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH SERVICES, ASSESSMENT REPORT, AND ANY ACCOMPANYING MATERIALS.

THE REPRESENTATIONS AND WARRANTIES OF CERTIK CONTAINED IN THIS AGREEMENT ARE SOLELY FOR THE BENEFIT OF CUSTOMER. ACCORDINGLY, NO THIRD PARTY OR ANYONE ACTING ON BEHALF OF ANY THEREOF, SHALL BE A THIRD PARTY OR OTHER BENEFICIARY OF SUCH REPRESENTATIONS AND WARRANTIES AND NO SUCH THIRD PARTY SHALL HAVE ANY RIGHTS OF CONTRIBUTION AGAINST CERTIK WITH RESPECT TO SUCH REPRESENTATIONS OR WARRANTIES OR ANY MATTER SUBJECT TO OR RESULTING IN INDEMNIFICATION UNDER THIS AGREEMENT OR OTHERWISE.

FOR AVOIDANCE OF DOUBT, THE SERVICES, INCLUDING ANY ASSOCIATED ASSESSMENT REPORTS OR MATERIALS, SHALL NOT BE CONSIDERED OR RELIED UPON AS ANY FORM OF FINANCIAL, TAX, LEGAL, REGULATORY, OR OTHER ADVICE.

Elevating Your Web3 Journey

Founded in 2017 by leading academics in the field of Computer Science from both Yale and Columbia University, CertiK is the largest blockchain security company that serves to verify the security and correctness of smart contracts and blockchain-based protocols. Through the utilization of our world-class technical expertise, alongside our proprietary, innovative tech, we're able to support the success of our clients with best-in-class security, all whilst realizing our overarching vision; provable trust for all throughout all facets of blockchain.