Haedal - haWAL

Security Assessment

CertiK Assessed on Oct 13th, 2025

CertiK Assessed on Oct 13th, 2025

Haedal - haWAL

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 09/24/2025
	Final report published on 10/13/2025

Vulnerability Summary

15	5	1	9	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.
0	Major		Major risks may include logical errors that, under specific circumstances, could result in fund losses or loss of project control.
1	Medium	1 Resolved	Medium risks may not pose a direct risk to users' funds, but they can affect the overall functioning of a platform.
5	Minor	2 Resolved, 3 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.
8	Informational	2 Resolved, 1 Partially Resolved, 5 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 - HAWAL

Summary

Executive Summary

Vulnerability Summary

Codebase

Audit Scope

Approach & Methods

Findings

HAH-06 : Centralization Related Risks And Upgradability

HAH-07 : Excessive Early Withdrawal During Instant Unstake Sweeps Surplus WAL Into Internal Vault

HAH-08 : Over-Withdrawal On Partial Claims Due To Splitting By WAL (Principal+Rewards) Instead Of Principal

HAH-09 : Early-Withdraw Path Would Cause Inconsistent `pool.total_staked` With `do_validator_request_withdraw`

HAH-10 : Missing Check On `validator_reward_fee`

HAH-16 : Potential Supply Inconsistency Due To Pre-Initialization Minting

HAH-17 : Zero `hawal_supply` With Nonzero `total_wal_amount_snapshot` Causing Incorrect Minting Ratio

HAH-01 : Potential Misleading `SystemStaked` Event Emitted For Merged Stakes

HAH-02 : Unused Fields In `Staking`

HAH-03 : Concerns About Unbounded Vector/VecMap Growth And Full Scans Can Exceed Sui Object/Gas Limits

HAH-04 : Instant-Unstake Bypasses `wal_vault` Liquidity And Withdraws Early From Validators

HAH-05 : Discussion On `Cap`

HAH-11 : Outdated Reward Updates Cause Inaccurate Minting And Fee Miscalculation

HAH-13 : Missing Check On `validator`

HAH-15 : Unused Deposit Fee

Optimizations

HAH-12 : Inconsistent Event Emission in `claim_collect_protocol_fee`

HAH-14 : UserStaked logs not log actual validator

Appendix

Disclaimer

CODEBASE | HAEDAL - HAWAL

Repository

https://github.com/haedallsd/hawal-contract/tree/audit

Commit

58fb5b7ac0b7555b28114b9699ac5de3fb96f1da

ce2201991c63731c3d52db09146b4b04b2b430e4

8a6bd7d1e942543587b02c55b1f85bba6b05a9a5

Audit Scope

The file in scope is listed in the appendix.

APPROACH & METHODS | HAEDAL - HAWAL

This audit was conducted for Haedal to evaluate the security and correctness of the smart contracts associated with the Haedal - haWAL 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.

FINDINGS | HAEDAL - HAWAL

15	0	1	0	1	5	8
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 15 issues were identified. Leveraging a combination of Formal Verification, Manual Review & Static Analysis the following findings were uncovered:

ID	Title	Category	Severity	Status
HAH-06	**Centralization Related Risks And Upgradability**	**Centralization**	**Centralization**	**Acknowledged**
HAH-07	Excessive Early Withdrawal During Instant Unstake Sweeps Surplus WAL Into Internal Vault	Volatile Code	Medium	Resolved
HAH-08	Over-Withdrawal On Partial Claims Due To Splitting By WAL (Principal+Rewards) Instead Of Principal	Incorrect Calculation	Minor	Acknowledged
HAH-09	Early-Withdraw Path Would Cause Inconsistent `pool.total_staked` With `do_validator_request_withdraw`	Volatile Code	Minor	Resolved
HAH-10	Missing Check On `validator_reward_fee`	Volatile Code	Minor	Resolved
HAH-16	Potential Supply Inconsistency Due To Pre-Initialization Minting	Logical Issue	Minor	Acknowledged
HAH-17	Zero `hawal_supply` With Nonzero `total_wal_amount_snapshot` Causing Incorrect Minting Ratio	Logical Issue	Minor	Acknowledged
HAH-01	Potential Misleading `SystemStaked` Event Emitted For Merged Stakes	Volatile Code	Informational	Acknowledged
HAH-02	Unused Fields In `Staking`	Coding Style	Informational	Resolved

ID	Title	Category	Severity	Status
HAH-03	Concerns About Unbounded Vector/VecMap Growth And Full Scans Can Exceed Sui Object/Gas Limits	Volatile Code	Informational	Acknowledged
HAH-04	Instant-Unstake Bypasses `wal_vault` Liquidity And Withdraws Early From Validators	Volatile Code	Informational	Acknowledged
HAH-05	Discussion On `Cap`	Access Control	Informational	Partially Resolved
HAH-11	Outdated Reward Updates Cause Inaccurate Minting And Fee Miscalculation	Logical Issue	Informational	Acknowledged
HAH-13	Missing Check On `validator`	Volatile Code	Informational	Acknowledged
HAH-15	Unused Deposit Fee	Coding Issue	Informational	Resolved

HAH-06 | Centralization Related Risks And Upgradability

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

Description

In the module **breaker**, the role **Breaker** has authority over the functions:

- toggle_unstake_v2()
- toggle_claim_v2()

In the module **manage**, the role **AdminCap** has authority over the functions:

- initialize()
- set_operator_cap_to_address()
- request_collect_rewards_fee()
- claim_collect_rewards_fee()
- set_deposit_fee_v2()
- set_reward_fee_v2()
- set_validator_reward_fee_v2()
- set_service_fee_v2()
- claim_collect_rewards_fee_v2()
- claim_collect_protocol_fee_v2()
- share_acl()
- add_minor_signs_to_acl()
- del_minor_signs()
- add_breaker_to_acl()
- del_breaker_to_acl()
- add_robot_to_acl()
- del_robot_to_acl()
- migrate()

In the module **operate**, the role **OperatorCap** has authority over the functions:

- set_deposit_fee()
- set_reward_fee()
- set_validator_reward_fee()
- set_service_fee()
- set_withdraw_time_limit()

- set_validator_count()
- set_active_validators()
- set_walrus_epoch_start()
- toggle_stake()
- toggle_unstake()
- toggle_claim()
- update_validator_rewards()
- sort_validators()
- migrate()
- request_collect_rewards_fee()
- claim_collect_rewards_fee()
- claim_collect_protocol_fee()
- validator_offline()

In the module **minorsign**, the role **MinorSign** has authority over the functions:

- set_withdraw_time_limit_v2()
- set_validator_count_v2()
- set_active_validators_v2()
- toggle_stake_v2()

In the module **robot**, the role **Robot** has authority over the functions:

- update_validator_rewards_v2()
- validator_offline_v2()
- sort_validators_v2()

If any of these privileged accounts are compromised, an attacker could exploit their enabled authorities to alter protocol parameters, manipulate staking and unstaking processes, upgrade or pause the contract, or change operator lists.

In addition, developers working with the Sui blockchain can upgrade packages based on their software iteration requirements. However, this also means that the `UpgradeCap` and deployer'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, 10/13/2025]:

All original privileges and contract upgrade authorities are assigned to the main multisignature wallet. Other operational permissions are managed by the main multisig via ACL, with ACL addresses structured hierarchically according to the level of privilege, using different multisig configurations. **AdminCap** and **OperatorCap** privileges have been transferred to the main multisignature wallet. All other Cap roles have been **deprecated** and are no longer in use.

[CertiK, 10/13/2025]:

The finding will be updated when corresponding multi-sig information is provided. Also, CertiK strongly encourages the project team to periodically revisit the private key security management of all addresses related to centralized roles.

HAH-07 Excessive Early Withdrawal During Instant Unstake Sweeps Surplus WAL Into Internal Vault

Category	Severity	Location	Status
Volatile Code	Medium	sources/walstaking.move (hawal): 600~601	Resolved

Description

User calls `request_unstake_instant`, which computes the WAL amount from input haWAL and invokes `do_unstake_instant` with `need_amount = max_exchange_wal_amount`. `do_unstake_instant` iterates validators while `need_amount > 0`, and, per validator, iterates staked entries that are early-withdrawable.

For each early-withdrawable entry, it computes how much to withdraw via `get_split_wal_amount`, withdraws from the underlying `walrus::staking`, and:

- If withdrawn ≤ need_amount: credits the entire amount to the user's bal and decreases need_amount.
- If withdrawn > need_amount: credits exactly need_amount to the user, deposits the surplus into staking.wal_vault, sets need_amount = 0, and continues.

However, the inner per-validator loop doesn't re-check `need_amount > 0`. After `need_amount` becomes 0, it continues scanning entries.

And when `need_amount == 0`, the process will dive into partial withdrawal flow in `get_split_wal_amount`, the `withdraw_principal` will be 1, as a result:

- 1 < MIN_STAKING_THRESHOLD (1e9) and principal ≥ 2MIN: (left_principal, MIN_STAKING_THRESHOLD)
- 1 < MIN and principal < 2MIN: returns (0, principal)

```
        } else { // split StakedWal and only withdraw partial
            let mut withdraw_principal = mul_div(need_amount, principal,
staked_wal_with_rewards) + 1;
            let left_principal = if (principal >= withdraw_principal) {
                principal - withdraw_principal
            } else {
                withdraw_principal = principal;
                0
            };

            if (withdraw_principal >= MIN_STAKING_THRESHOLD) {
                if (left_principal >= MIN_STAKING_THRESHOLD) { // can be split
                    (left_principal, withdraw_principal)
                } else { // can't be split
                    (0, principal)
                }
            } else {
                if (principal >= 2*MIN_STAKING_THRESHOLD) { // can be split
                    (principal-MIN_STAKING_THRESHOLD, MIN_STAKING_THRESHOLD) //
withdraw at least 1 SUI
                } else { // can't be split
                    (0, principal)
                }
            }
        };
```

As a result, more than expected WAL will be unstaked.

Recommendation

Recommend handling the condition when `need_amount` is 0.

Alleviation

[Haedal, 09/23/2025]: The team heeded the advice and resolved the issue by adding validation on `need_amount` in commit 4a502e6c095da654e9e610c9fbe4764e345e07e9

HAH-08 Over-Withdrawal On Partial Claims Due To Splitting By WAL (Principal+Rewards) Instead Of Principal

Category	Severity	Location	Status
Incorrect Calculation	Minor	sources/walstaking.move (hawal): 743	Acknowledged

Description

To initiate the unstake process, the user burns `HAWAL` and receives an `UnstakeTicket` with `wal_amount` to be paid later:

```
        let ut = UnstakeTicket {
            id: object::new(ctx),
            unstake_timestamp_ms: now_ms,
            hawal_amount: input_hawal_amount,
            wal_amount: max_exchange_wal_amount,
            claim_epoch: claim_epoch,
            claim_timestamp_ms: claim_timestamp_ms,
        };
```

Protocol schedules withdrawals across validators using `do_validator_request_withdraw`, which converts the WAL to-pay into a principal to split via `get_split_wal_amount`, and moves positions into `withdrawing`:

```
            let staked_wal_ref = pool.staked.get_mut(&key);
            let (left_principal, withdraw_principal) =
get_split_wal_amount(wal_staking, staked_wal_ref, need_amount, current_epoch);
            // decide whether withdraw entire or partial StakedWal
            let mut withdraw = if (left_principal > 0) {
                let withdraw = staked_wal_ref.split(withdraw_principal, ctx);
                withdraw
            ...
```

On claim (`withdraw_stake`), protocol tries to pay from `wal_vault` first, then withdraws matured `withdrawing` positions via `do_validator_withdraw`. Any surplus after satisfying the exact user amount is deposited back into `wal_vault`:

```
        let (mut bal, left_amount) = vault::withdraw_max(&mut staking.wal_vault,
need_amount);
        need_amount = left_amount;

        if (need_amount > 0) {
            let mut unstaked_bal = balance::zero();
...
            assert!(need_amount == 0, EUnstakeNeedAmountIsNotZero);

            bal.join(unstaked_bal.split(left_amount));
            staking.wal_vault.deposit(unstaked_bal);
        };
```

However, in `do_validator_withdraw`, partial splitting of a matured `withdrawing` position uses `need_amount` directly as the split amount:

```
                let withdraw = if (
                    need_amount > MIN_STAKING_THRESHOLD &&
                    need_amount + MIN_STAKING_THRESHOLD < withdrawing_wal_ref_value
                ) {
                    let withdraw = withdrawing_wal_ref.split(need_amount, ctx);
                    withdraw
                } else {
                    let (_, withdraw) = pool.withdrawing.remove(&key);
                    withdraw
                };
```

The `need_amount` is the WAL to deliver to the user (principal + rewards), while `StakedWal::split` expects a principal amount:

In `local_wal/walus`:

```
public fun split(sw: &mut StakedWal, amount: u64, ctx: &mut TxContext): StakedWal {
    assert!(sw.principal.value() > amount, EInvalidAmount);
    // Both parts after the split must have a principal of at least
MIN_STAKING_THRESHOLD.
    assert!(amount >= MIN_STAKING_THRESHOLD, EStakeBelowThreshold);
    assert!(sw.principal.value() - amount >= MIN_STAKING_THRESHOLD,
EStakeBelowThreshold);
    StakedWal {
        id: object::new(ctx),
        state: sw.state, // state is preserved
        node_id: sw.node_id,
        principal: sw.principal.split(amount),
        activation_epoch: sw.activation_epoch,
    }
}
```

And `staking::withdraw_stake` returns principal + rewards for that chunk:

```
    let principal = staked_wal.into_balance();

    // Withdraw rewards...
    let rewards_amount = rewards_amount.min(pool.rewards_pool.value());
    let mut to_withdraw = pool.rewards_pool.split(rewards_amount);
    to_withdraw.join(principal);
    to_withdraw
```

As a result, the user may get extra WAL.

Recommendation

Recommend considering splitting the StakedWal based on principle in `do_validator_withdraw` .

Alleviation

[Haedal, 09/24/2025]: We understand that staking::withdraw_stake returns the principal plus rewards, so we designed a **sliding window** approach to address this.

1. The protocol will receive more rewards after performing staking::withdraw_stake, but we will deposit **these rewards into wal_vault**.

2. The next time a claim is initiated, the protocol will prioritize wal_vault, causing need_amount to become a smaller value than left_amount, which will be less than need_amount.

3. In fact, the entire protocol uses a **sliding window** reward approach, **including request_withdraw_stake, request_unstake_instant, and withdraw_stake.** This design is designed so that in extreme cases, the minimum withdrawal of 1 WAL may need to be increased, and any excess redemptions generated will need to be deposited into wal_vault.

HAH-09 Early-Withdraw Path Would Cause Inconsistent `pool.total_staked` With `do_validator_request_withdraw`

Category	Severity	Location	Status
Volatile Code	Minor	sources/walstaking.move (hawal): 466~470	Resolved

Description

The function `do_validator_request_withdraw` requests unstake from a specific validator and, if possible, performs an early withdrawal and deposits the WAL into the vault. And the `pool.total_staked` does not change during the process.

However, in `do_unstake_instant` after early-withdraw, the amount is derived from `pool.total_staked` .

Also, in the following `withdraw_stake` invocation, the user claims to pull from the vault first, which also does not adjust the `pool.total_staked` .

As a result, the functions use `pool.total_staked` would return inaccurate results, for example, `get_min_total_validator` .

Recommendation

Recommend handling `pool.total_staked` with the early withdrawal path in normal unstake.

Alleviation

[Haedal, 09/24/2025]: The team heeded the advice and resolved the issue by updating `pool.total_staked` in early withdrawal path in commit 4a502e6c095da654e9e610c9fbe4764e345e07e9

HAH-10 | Missing Check On `validator_reward_fee`

Category	Severity	Location	Status
Volatile Code	Minor	sources/config.move (hawal): 100~101	Resolved

Description

The `set_validator_reward_fee()` function does not enforce an upper limit on the `validator_reward_fee` parameter. Additionally, although this fee rate is defined, it is not referenced or utilized elsewhere in the project code.

Recommendation

If the intention is to use `validator_reward_fee` in future logic, it is recommended to implement an appropriate maximum limit to prevent misconfiguration or unintended values.

Alleviation

[Haedal, 09/24/2025]: The team heeded the advice and resolved the issue by adding validation on the upper bound of the fee in commit 4a502e6c095da654e9e610c9fbe4764e345e07e9

HAH-16 | Potential Supply Inconsistency Due To Pre-Initialization Minting

Category	Severity	Location	Status
Logical Issue	Minor	sources/hawal.move (update-0925): 7~8; sources/manage.move (update-0925): 50~51; sources/walstaking.move (update-0925): 241~242	Acknowledged

Description

According to the intended project logic, the expected call chain during initialization is:

```
hawal::init() -> manage::initialize()
```

Within this process, `manage::initialize()` internally invokes `walstaking::initialize(treasuryCap, ctx)`.

The `TreasuryCap` is stored in `Staking.hawal_treasury_cap` only inside the `walstaking::initialize()` function, as shown below:

```
public(package) fun initialize(cap: TreasuryCap<HAWAL>, ctx: &mut TxContext) {
    ...
    let staking_object = Staking {
        ...
        hawal_treasury_cap: cap,
        ...
    };
}
```

The value `coin::total_supply(&staking.hawal_treasury_cap)` is used to record the total amount of minted HAWAL coins, which is critical for calculating the amount of HAWAL received when users deposit WAL.

However, after the `hawal::init()` call and before the `manage::initialize()` call, the `TreasuryCap` remains temporarily held by the caller of `hawal::init()`. During this period, the caller still has the authority to mint new coins. The `walstaking::initialize()` function does not perform any validation to ensure that no coins have been minted externally prior to initialization.

If the holder mints additional coins before `walstaking::initialize()` is executed, the `staking.hawal_supply` will be initialized with an inflated total supply value:

```
staking.hawal_supply = coin::total_supply(&staking.hawal_treasury_cap);
```

This results in incorrect accounting during subsequent deposits, causing the subsequent depositors to receive an inconsistent amount of minted HAWAL coins.

Recommendation

It is recommended to add a validation within `walstaking::initialize()` to ensure that no additional `HAWAL` coins have been minted prior to initialization.

Alleviation

[Haedal, 10/12/2025]:

As mentioned, TreasuryCap can still be called after hawal::init() and before manage::initialize(). However, since the hawal protocol has been successfully launched on the mainnet (ensuring no calls between hawal::init() and manage::initialize()), and has been running normally for some time, we are no longer considering calling initialize() again. Therefore, we are not currently considering fixing this issue.

HAH-17 Zero `hawal_supply` With Nonzero `total_wal_amount_snapshot` Causing Incorrect Minting Ratio

Category	Severity	Location	Status
Logical Issue	Minor	sources/walstaking.move (update-0925): 1152~1153	Acknowledged

Description

The `get_hawal_by_wal()` function calculates the amount of `HAWAL` to mint based on the total `WAL` balance in the system and the total minted `HAWAL` supply. The function returns the same amount of `HAWAL` as the input `WAL` amount if the following condition is met:

```
let total_wal_amount_snapshot = get_total_wal(staking);
if (total_wal_amount_snapshot == 0 || staking.hawal_supply == 0) {
    return wal_amount
};
```

This means that whenever either `total_wal_amount_snapshot` or `staking.hawal_supply` equals zero, the minting ratio defaults to 1:1.

Since `get_hawal_by_wal()` and `get_wal_by_hawal()` serve as the core calculation functions for staking and unstaking operations, respectively, both rely on division (`/`) operations that can introduce truncation errors. Over multiple staking and unstaking cycles, repeated truncation can result in residual “dust” `WAL` left in the system, even when `staking.hawal_supply` becomes zero.

Furthermore, if the `request_withdraw_stake()` function is invoked, the corresponding `HAWAL` is burned first, which may reduce `staking.hawal_supply` to zero while the `StakedWal` remains stored in `pool.withdrawing`. If `update_validator_rewards()` is then called, it will increase `staking.total_rewards`, resulting in a state where `total_wal_amount_snapshot` becomes nonzero while `staking.hawal_supply` remains zero.

```
public fun get_total_wal(staking: &Staking): u64 {
    staking.total_staked + staking.total_rewards
    - staking.collected_protocol_fees
    - staking.uncollected_protocol_fees
    - staking.total_unstaked
}
```

In this situation, the first user to stake will receive `HAWAL` at a 1:1 ratio and, upon unstaking, can potentially claim all the remaining `WAL` “dust” and accumulated rewards, resulting in unfair benefit and incorrect system state.

Recommendation

It is recommended to abort execution when `staking.hawal_supply == 0` but `total_wal_amount_snapshot != 0`.

When updating rewards at the beginning of an epoch, the administrator should, through a controlled entry point, first “seed” a certain amount of `HAWAL` to the protocol address (for example, by minting it at a 1:1 ratio to the current `total_wal`). This establishes a valid initial exchange rate before reopening staking to users.

Alleviation

[Haedal, 10/12/2025]:

We are aware of this issue and consider it acceptable. We will not address it for the following reasons:

1. When hawal_supply is 0 and total_wal_amount_snapshot is not 0, we assume that all users have cleared the protocol's staked WAL. Therefore, when the next user stakes, they should receive all rewards, including all subsequent rewards and dust, since their stake is 100%.

2. During the protocol's operation, it is very possible for hawal_supply to become 0, leading to truncation and dust generation. However, this is an extreme case, and the truncation dust will remain in the protocol, allowing the next user to receive rewards.

3. When request_withdraw_stake causes hawal to become 0, it means that all staked WAL has been withdrawn. At this time, calling update_validator_rewards will cause Walrus to generate no rewards since no WAL is staked. Therefore, staking.total_rewards will not be accumulated.

HAH-01 Potential Misleading `SystemStaked` Event Emitted For Merged Stakes

Category	Severity	Location	Status
Volatile Code	Informational	sources/walstaking.move (hawal): 352	Acknowledged

Description

When a user initiates a stake to a validator for the first time in a particular `activation_epoch`, a new `StakedWal` object is created and stored. If an additional stake is made to the same validator during the same epoch—by the same or another user—the system optimizes storage by merging the new stake into the existing `StakedWal`. This is accomplished by creating a temporary `StakedWal` object for the incoming stake, invoking `join` to transfer its balance to the existing object, and then destroying the temporary instance.

Currently, the function records the object ID of the temporary `StakedWal` before the merge and destruction steps. After these actions, it emits a `SystemStaked` event referencing this now-destroyed object ID. As a result, the emitted event refers to a `StakedWal` object that no longer exists, leading to discrepancies between event logs and actual on-chain state. This can cause confusion or misinterpretation for users or off-chain services relying on event data to track staking activity.

Recommendation

We would like to check with the team about the usage of the `SystemStaked` and if the scenario has been considered.

Alleviation

[Haedal, 09/24/2025]: In fact, off-chain tracking is more concerned with other fields under SystemStaked. The staked_wal_id is just a redundant record. Currently, it does not cause service confusion. In the future, when services need to obtain information, we will traverse the pool instead of emit to determine the data.

HAH-02 | Unused Fields In `Staking`

Category	Severity	Location	Status
Coding Style	Informational	sources/walstaking.move (hawal): 78, 91	Resolved

Description

The following fields in `Staking` are not being used:

- `collected_protocol_fees_pending`
- `rewards_last_updated_epoch`

Recommendation

We would like to check with the team if any logic is missing.

Alleviation

[Haedal, 09/24/2025]: The current code is all up to date and there is no logic missing. This is a reserved field. Based on the implementation of the haSUI protocol, we believe that this field may be used in the future.

HAH-03 Concerns About Unbounded Vector/VecMap Growth And Full Scans Can Exceed Sui Object/Gas Limits

Category	Severity	Location	Status
Volatile Code	Informational	sources/walstaking.move (hawal): 96~97	Acknowledged

Description

The staking design stores per-validator state in child `PoolInfo` objects inside a Table, which is good for isolating size. However, each `PoolInfo` uses `VecMap<u32, StakedWal>` for both `staked` and `withdrawing`. Over time, these maps can accumulate many entries (e.g., from splits and multiple epochs). Critical flows iterate all keys via `keys()` (allocates a vector) and then linearly scan, which can grow to O(n) per pool and O(n·validators) overall. This increases per-tx gas and risks hitting Sui object-size and compute limits, potentially causing withdrawals/claims/rewards updates to fail as state grows.

```
        staked: VecMap<u32, StakedWal>, // Activation Epoch as key
        withdrawing: VecMap<u32, StakedWal>, // Withdrawing Epoch as key
```

Root vectors `active_validators` / `validators` are expected to be small based on current walrus mainnet data, but future growth is unknown:

```
        /// validators that have stakes, could be ordered by apy
        active_validators: vector<ID>,
        /// /// active validators in current epoch, updated every epoch.
        validators: vector<ID>,
```

Recommendation

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

Alleviation

[Haedal, 09/24/2025]:

Yes, we've considered this.

1. In the case of multiple epochs, when a user is using the protocol, withdrawals will prioritize the staked tokens from the smallest (oldest) epoch. Even if there are no users, the management console will proactively intervene in withdrawals.

2. Currently, the number of Walrus validators is relatively small, but it may increase in the future. However, the HaWAL protocol itself is designed to maximize user returns. When there are many Walrus nodes, the protocol will select the highest-yielding few or dozens for staking, rather than all of them. This is the purpose of active_validators.

HAH-04 Instant-Unstake Bypasses `wal_vault` Liquidity And Withdraws Early From Validators

Category	Severity	Location	Status
Volatile Code	Informational	sources/walstaking.move (hawal): 545	Acknowledged

Description

The `request_unstake_instant` function allows users to immediately withdraw their stake by paying a fee, provided there is sufficient StakedWal available for withdrawal.

Unlike the delayed claim path, which withdraws funds from the `wal_vault` first, `request_unstake_instant` does not verify whether `wal_vault` has enough liquidity before proceeding. Instead, it directly invokes `do_unstake_instant`, which attempts to withdraw funds early from validators. Any excess funds are then deposited into `wal_vault`. This approach introduces an inconsistency between the instant and delayed withdrawal flows.

Recommendation

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

Alleviation

[Haedal, 09/29/2025]: We are aware of this situation and have considered it. Due to Walrus limitations, a minimum withdrawal requirement of 1 WAL is required, so we deposit any remaining balance into the wal_vault. However, we do not consider performing operations on the wal_vault vault during instantaneous withdrawals. Since the value of wal_vault is calculated in do_validator_request_withdraw(request_unstake_delay), the value of wal_vault is also used during withdraw_wal_bal(claim).

In fact, if the wal_vault vault is withdrawn instantaneously, a situation will arise: after user A executes request_unstake_delay, user B executes request_unstake_instant in the same epoch, resulting in insufficient balance in the wal_vault vault. This could lead to a DOS error when user A claims (to meet the epoch limit). This is undesirable.

At the same time, we do not recommend using request_unstake_instant, but prefer to use request_unstake_delay>claim

---

[Haedal, 10/09/2025]: Yes, we are aware of this situation, and it's intentional.

1. First, we do not recommend using request_unstake_instant to withdraw funds, as reflected on our website. We prefer using request_withdraw_stake > withdraw_stake.

2. Second, we deposit any excess redemptions into the wal_vault, but will not use request_unstake_instant again. This is because request_withdraw_stake contains the staking::can_withdraw_staked_wal_early check. If a user requests_withdraw_stake and the staking::can_withdraw_staked_wal_early condition is met, the corresponding amount will be withdrawn and deposited into the wal_vault. After the user meets the epoch conditions (usually N+1|N+2, as the walrus cycle is one week, which is quite long), withdraw_stake will be called to withdraw funds. While the user is waiting

for the epoch, other users may request_unstake_instant, reducing the wal_vault. This will cause a normal user to get an error "wal_vault insufficient docs" when performing a withdraw_stake. This is not what we want to see, so in the request_unstake_instant, the wal_vault will not be deducted, but any excess redemption will be deposited into the wal_vault.

HAH-05 | Discussion On `Cap`

Category	Severity	Location	Status
Access Control	Informational	sources/manage.move (hawal): 20~38	Partially Resolved

Description

`MinorSignCap`, `BreakerCap`, `RobotCap`, 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.

```
20      /// `OperatorCap` is used by the offchain programs.
21      public struct OperatorCap has store, key {
22          id: UID,
23      }
24
25      // `MinorSignCap` is used by n of 2
26      public struct MinorSignCap has store, key {
27          id: UID,
28      }
29
30      // `BreakerCap` is used by the circuit breaker.
31      public struct BreakerCap has store, key {
32          id: UID,
33      }
34
35      // `RobotCap` is used by the robot.
36      public struct RobotCap has key, store {
37          id: UID,
38      }
```

Recommendation

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

Alleviation

[Haedal, 10/09/2025]: The team heeded the advice and partially resolved the issue by using ACL members in commit 98165fcd308d4211f6d9573d4962269b2f4c6f40.

[CertiK, 10/09/2025]: The team has removed the `MinorSignCap`, `BreakerCap`, and `RobotCap`, and now uses the `ACL` member instead. The `OperatorCap` is still in use.

HAH-11 Outdated Reward Updates Cause Inaccurate Minting And Fee Miscalculation

Category	Severity	Location	Status
Logical Issue	Informational	sources/operate.move (hawal): 78~79	Acknowledged

Description

The function `update_validator_rewards()` is designed for the admin to update `staking.total_rewards` and `staking.uncollected_protocol_fees` at the beginning of the epoch, which are essential for determining the exchange rate and the amount of tokens to mint, as well as for protocol fee collection.

However, user-facing functions such as `request_stake_coin()`, `request_withdraw_stake()`, and `request_unstake_instant()` can be executed before the rewards for the current epoch are updated, the protocol relies on stale reward data, which introduces several issues:

- The `request_stake_coin()` function mints `haWAL` tokens based on an outdated `staking.total_rewards` value. If new rewards have not yet been added, this can result in over-minting.
- `staking.uncollected_protocol_fees` is updated only by the aforementioned admin-only functions. If users unstake or claim before these updates, their share of rewards may not be properly reflected in `staking.total_rewards`, and protocol fees may be miscalculated, leading to potential loss of fees for the project.
- When `do_validator_withdraw()` is invoked before rewards are updated for the epoch, a portion of the rewards may be distributed or withdrawn without these amounts being added to `staking.total_rewards` while unstake from native staking system, resulting in subsequent users receiving more minted tokens than intended due to an inaccurate exchange rate.
- If rewards have not been updated for a long time, such as multiple epochs, and users withdraw before the update, the protocol can both double-count rewards for remaining stakes and fail to account for withdrawn rewards.
  - On withdraw: rewards are paid out but not added to `staking.total_rewards`. `pool.rewards` is reduced by `withdraw_rewards` (or set to 0 if insufficient).
  - On subsequent update: `increment = pool_rewards - pool.rewards`. With `pool.rewards == 0`, the update re-credits all remaining stake's historical + current rewards, including portions already credited in past updates -> double-count for remaining stakes.

Currently, only the explicit reward update functions update `staking.total_rewards` via the following chain: `update_validator_rewards()` -> `calculate_validator_pool_rewards_increase()` -> `calculate_staked_wal_rewards()`

In contrast, `do_validator_withdraw()` also calls `calculate_staked_wal_rewards()`, but its computed rewards do not update `staking.total_rewards`.

This design allows timing gaps in which users can interact with outdated protocol state.

Recommendation

It's recommended that user operations such as staking, withdrawing, and instant unstaking be restricted between the start of a new epoch and the completion of reward updates to ensure up-to-date protocol state. The audit team also recommends tracking the epoch in which `staking.total_rewards` was last updated and enforcing a check in user-facing functions, such as `request_stake_coin()`, `request_withdraw_stake()`, and `request_unstake_instant()`, to verify that `staking.rewards_last_updated_epoch` matches the current epoch before processing new user requests.

Alleviation

[Haedal, 09/24/2025]: We will do this (pause user operations during epoch changes), and for epoch synchronization, we have been doing.

HAH-13 | Missing Check On `validator`

Category	Severity	Location	Status
Volatile Code	Informational	sources/walstaking.move (update-0925): 277~278, 341~342	Acknowledged

Description

The `request_stake_coin()` function does not validate whether the provided `validator` is actually registered in the staking pool. If an unregistered validator is passed in, the subsequent call to

```
staking::stake_with_pool(wal_staking, coin::from_balance(bal, ctx), validator, ctx)
```

will abort when `stake_with_pool` attempts to access `self.pools[node_id]` for a non-existent `validator`.

```
public(package) fun stake_with_pool(
    self: &mut StakingInnerV1,
    to_stake: Coin<WAL>,
    node_id: ID,
    ctx: &mut TxContext,
): StakedWal {
    let wctx = &self.new_walrus_context();
    let pool = &mut self.pools[node_id]; // aborts if node_id (validator) is not
registered
    ...
}
```

Recommendation

Since the team also maintains an `active_validators` vector, we would like to check with the team if the scenario has been considered for early revert.

Alleviation

[Haedal, 10/09/2025]: Yes, the management platform properly controls `active_validators`. This will happen at each cycle change, and the program will automatically synchronize Walrus' validators. This includes next_committee and committee in the Walrus protocol. The program will analyze whether there are nodes offline or newly added, and then flexibly match active_validators based on the results. If a node is found to be offline, the program will automatically call validator_offline to offline the node.

HAH-15 | Unused Deposit Fee

Category	Severity	Location	Status
Coding Issue	Informational	sources/config.move (update-0925): 12	Resolved

Description

Although `StakingConfig` exposes `deposit_fee` and there are admin setters, the stake flow never charges it nor credits `protocol_wal_vault`, making the fee configuration ineffective.

Recommendation

We would like to check with the team if any logic missing.

Alleviation

[Haedal, 10/12/2025]:

This is intentional, we think we might use deposit_fee in the future.

OPTIMIZATIONS | HAEDAL - HAWAL

ID	Title	Category	Severity	Status
HAH-12	Inconsistent Event Emission In `claim_collect_protocol_fee`	Coding Style	Optimization	Resolved
HAH-14	UserStaked Logs Not Log Actual Validator	Volatile Code	Optimization	Resolved

HAH-12 | Inconsistent Event Emission In `claim_collect_protocol_fee`

Category	Severity	Location	Status
Coding Style	Optimization	sources/walstaking.move (update-0925): 944	Resolved

Description

When collecting protocol fee, the event `RewardsFeeCollected` is emitted.

Recommendation

Recommend triggering an event with the protocol fee instead to avoid confusion.

Alleviation

[Haedal, 10/09/2025]: The team heeded the advice and resolved the issue by emitting the `ProtocolFeeCollected` event in commit 1a7cf34815f4a2481217b30e476e38bf576d4693.

HAH-14 | UserStaked Logs Not Log Actual Validator

Category	Severity	Location	Status
Volatile Code	Optimization	sources/walstaking.move (update-0925): 312	Resolved

Description

When callers pass the 0x0 to request auto-selection, the callee chooses a real validator, but the emitted UserStaked event still records the original (sentinel-derived) validator argument. This makes UserStaked.validator not reflect the actual validator that received the stake, which raised the concern if any off-chain service relies on such events.

Recommendation

Recommend using an actual validator for the event if any service relies on the UserStaked event.

Alleviation

[Haedal, 10/12/2025]:

When the caller passes 0x0 to request the protocol to be automatically selected, the log under the chain will also record 0x0, and then the program will automatically use the validator in SystemStaked as the node to record it in the log. In fact, we will not use the validator in UserStaked, but the validator in SystemStaked.

APPENDIX | HAEDAL - HAWAL

Audit Scope

haedallsd/hawal-contract
sources/operate.move
sources/walstaking.move
sources/hawal.move
sources/manage.move
sources/walstaking.move
sources/manage.move
sources/config.move
sources/config.move
sources/breaker.move
sources/hawal.move
sources/interface.move
sources/minorsign.move
sources/robot.move
sources/vault.move
sources/breaker.move
sources/interface.move
sources/minorsign.move
sources/operate.move
sources/robot.move

haedallsd/hawal-contract
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.
Access Control	Access Control findings are about security vulnerabilities that make protected assets unsafe.
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.

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.