Haedal

Audit

Presented by:

OtterSec contact@osec.io

Michał Bochnak embe221ed@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Haedal engaged OtterSec to perform an assessment of the `liquid staking protocol` program. This assessment was conducted between November 1st and November 10th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Over the course of this audit engagement, we produced 9 findings in total.

In particular, we identified several vulnerabilities, including one regarding the ability to claim rewards, before approving the corresponding Epoch Claim object (OS-HDL-ADV-02). Additionally, we emphasized the risk of increasing the withdrawal time limit beyond the current epoch timestamp, resulting in the minting of unstake tickets with timestamps violating the new limit. This further results in non-ascending epoch claims (OS-HDL-ADV-01).

We also made recommendations concerning the lack of safeguards, potentially allowing the setting of the validator count to zero (OS-HDL-SUG-02), and suggested the utilization of specific functions within the code base to enhance code readability and eliminate duplicated code (OS-HDL-SUG-03). Lastly, we advised incorporating a pause functionality (OS-HDL-SUG-01).

02 | Scope

The source code was delivered to us in a git repository at github.com/haedallsd/haedal-protocol/. This audit was performed against commit 239deaf.

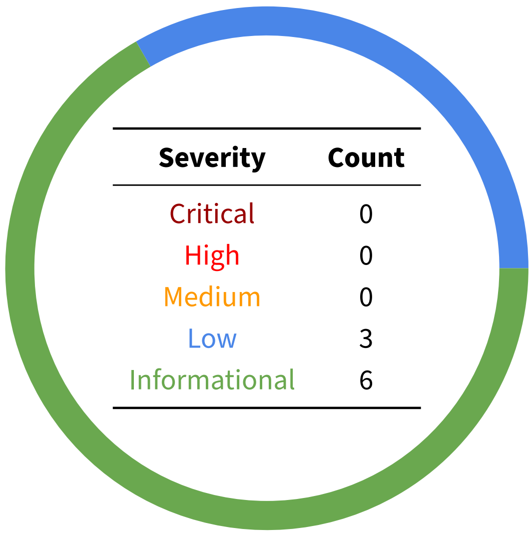
A brief description of the programs is as follows:

Name	Description
liquid staking protocol	A non-custodial liquid staking protocol, developed on Sui, enabling individuals to stake their SUI tokens to contribute to the governance and decentralization of the Sui blockchain.

03 | Findings

Overall, we reported 9 findings.

We split the findings into **vulnerabilities** and **general findings**. Vulnerabilities have an immediate impact and should be remediated as soon as possible. General findings do not have an immediate impact but will aid in mitigating future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	3
Informational	6

04 | Vulnerabilities

Here, we present a technical analysis of the vulnerabilities we identified during our audit. These vulnerabilities have *immediate* security implications, and we recommend remediation as soon as possible.

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-HDL-ADV-00	Low	Resolved	Capability to withdraw the entire staked amount with a minimal value of `need_amount`.
OS-HDL-ADV-01	Low	Resolved	Increasing `withdraw_time_limit` beyond the current epoch timestamp may break the assumption that epoch claims are sorted
OS-HDL-ADV-02	Low	Resolved	Ability to claim rewards via `claim_coin` before approval of the corresponding `EpochClaim` object for the current epoch.

OS-HDL-ADV-00 [low] | Excessive Withdrawal Of Staked Amount

Description

`get_split_amount` in `staking`, is designed to determine the amount of staked SUI (`StakedSui`) that can be withdrawn, considering the need for a specific amount.

sources/staking.move RUST

```
fun get_split_amount(wrapper: &mut SuiSystemState, staked_sui_ref:&StakedSui,
    ↪ need_amount: u64, current_epoch: u64): (u64, u64) {
    [...]
    // withdraw entire or partial of the statked_sui
    let (left_principal, withdraw_principal) = if (staked_sui_with_rewards<=
        ↪ need_amount) { // withdraw entire StakedSui
        (0, staked_sui_with_rewards)
    } else { // split StakedSui and only withdraw partial
        let withdraw_principal = util::mul_div(need_amount,
            ↪ principal,staked_sui_with_rewards) + 1;
        let left_principal = principal - withdraw_principal;
        if (withdraw_principal >= MIN_STAKING_THRESHOLD && left_principal>=
            ↪ MIN_STAKING_THRESHOLD) { // can be split
            (left_principal, withdraw_principal)
        } else { // can't be split
            (0, staked_sui_with_rewards)
        }
   };
   (left_principal, withdraw_principal)
}
```

The issue stems from the existing implementation of `get_split_amount`, where a modest `need_amount` (e.g., 1 MIST) has the potential to trigger the withdrawal of the entire `StakedSui`, even if the total staked SUI amount is considerably high, instead of withdrawing only a portion of the staked amount as intended, and keeping the rest staked.

Remediation

Ensure to modify `get_split_amount` to better handle small values of `need_amount`.

Patch

Fixed in 56a221a

OS-HDL-ADV-01 [low] | Non-Ascending Epoch Claims

Description

The issue is with regards to `set_withdraw_time_limit`, which grants administrators the ability to define the time limit for withdrawing staked tokens. To elaborate, should the administrator elevate the `withdraw_time_limit` to surpass the present timestamp calculated from the inception of the epoch, it may result in a scenario where a user generates two `UnstakeTicket` objects within the same epoch.

sources/manage.move RUST

```rust
public entry fun set_withdraw_time_limit(_: &AdminCap, staking: &mutStaking,
    ↪  withdraw_time_limit: u64) {
    staking::assert_version(staking);

    config::set_withdraw_time_limit(
    staking::get_config_mut(staking),
    withdraw_time_limit);
}
```

In particular, if the timestamp (X) of the first ticket exceeds `epoch_timestamp_ms + old_limit`, and the timestamp (Y) of the second ticket falls below `epoch_timestamp_ms + new_limit`, the resultant `EpochClaim` for these two tickets will correspond to epochs `E+2` and `E+1`, respectively. This creates a scenario where the epoch claims are in a non-ascending order, which is problematic as epoch claims should be in ascending order. This non-ascending order may disrupt the normal flow of epoch-based operations.

Remediation

Ensure that changes to `withdraw_time_limit` are carefully managed, considering the potential impact on the timestamp-based order of epoch claims.

Patch

Fixed in 2d543ae.

OS-HDL-ADV-02 [low] | Possible Race Condition

Description

`claim_coin` in `Staking` handles the claiming process of SUI tokens after a user initiates an unstaking operation. It calls `claim_epoch_record` internally, which updates the staking contract's records related to claimed amounts for specific epochs.

sources/staking.move RUST

```rust
/// claim the SUI back, and return SUI coin
public fun claim_coin(
    staking: &mut Staking,
    ticket: UnstakeTicket,
    ctx:&mut TxContext): Coin<SUI> {
        [...]
        claim_epoch_record(staking, claim_epoch, sui_amount);
        let bal = vault::withdraw(&mut staking.claim_sui_vault, sui_amount);
        let sender = tx_context::sender(ctx);
        staking.unclaimed_sui_amount = staking.unclaimed_sui_amount - sui_amount;
        [...]
    }
```

The vulnerability originates from a potential race condition during the transition between epochs, which arises when a user executes `claim_coin` before the program approves the corresponding `EpochClaim` object for the current epoch. Thus, if the claiming occurs before the approval, `claim_epoch_record` may not decrease the value of `ue.amount` as expected because `ue.approved` will be set to false.

sources/staking.move RUST

```rust
fun claim_epoch_record(staking: &mut Staking, epoch: u64, sui_amount:u64) {
    [...]
    while (i < length) { // unstake_epochs is ordered by the epoch
        let ue = vector::borrow_mut(&mut staking.unstake_epochs, i);
        if (ue.epoch == epoch && ue.approved) {
            ue.amount = ue.amount - sui_amount;
            [...]
        };
        i = i + 1;
    };
    [...]
}
```

Consequently, this may result in a successful reward claim, where the program does not properly deduct the claimed amount from `ue.amount` due to the unapproved state of the `EpochClaim` object. Note that while this is a vulnerability in itself, there might be additional concerns associated with not locking the protocol until epoch-update-connected functions have been completely executed.

Proof of Concept

Illustrating the above issue with an example:

1. The current epoch is N, and the staking contract is transitioning to epoch $N + 1$.
2. During this transition, the new `EpochClaim` object for epoch $N + 1$ is created, and the approved flag is initially set to false.
3. A user decides to claim their staking rewards by now calling `claim_coin`.
4. Inside `claim_coin`, `claim_epoch_record` is called to update the staking records.
5. However, since the epoch transition is not complete, the `EpochClaim` object for epoch $N + 1$ is still in an unapproved state (`ue.approved = false`).
6. Due to the race condition, `claim_epoch_record` may not decrease `ue.amount` since it checks for the approval(`ue.approved`), and this flag is still false during the epoch transition.

Remediation

Ensure to lock the protocol within the function that is initially called at the end of the epoch, and subsequently, unlock the protocol within the final function called at the beginning of the new epoch.

Patch

Fixed in 2d543ae.

05 | General Findings

Here, we present a discussion of general findings during our audit. While these findings do not present an immediate security impact, they represent anti-patterns and may result in security issues in the future.

ID	Description
OS-HDL-SUG-00	Missing validation to verify if the exchange rate for `current_epoch` exists or not.
OS-HDL-SUG-01	Missing pause functionality in `claim_coin_v2`.
OS-HDL-SUG-02	The absence of safeguards in `set_validator_count` may allow the setting of `validator_count` to zero.
OS-HDL-SUG-03	Suggestions regarding the utilization of certain functions within the code base to improve code readability.
OS-HDL-SUG-04	Incorporate an absent check.
OS-HDL-SUG-05	Potential denial of service through exploiting the staking system.

OS-HDL-SUG-00 | Exchange Rate Check

Description

`calculate_rewards` derives the rewards a user would receive when withdrawing staked funds from a pool, considering exchange rates at the time of stake activation and the current epoch.

sources/util.move RUST

```
/// these codes are copied and changed from sui_system::staking_pool.move
public fun calculate_rewards(wrapper: &mut SuiSystemState,
    pool_id: ID,staked_amount: u64,
    stake_activation_epoch: u64,
    current_epoch: u64):u64{
        [...]
        let exchange_rates = sui_system::pool_exchange_rates(wrapper,&pool_id);
        [...]
    }
```

It currently lacks a check to validate if the exchange rate for `current_epoch` exists. Thus, if the exchange rate for the `current_epoch` is unavailable, it indicates that the system has insufficient information to perform this conversion, resulting in inaccurate reward calculation.

Remediation

Return zero when the exchange rate for the `current_epoch` is not available, providing a safe fallback value indicating the absence of the exchange rate.

OS-HDL-SUG-01 | Lack Of Pause Functionality

Description

`claim_coin_v2` handles the process of claiming SUI coins, ensuring that the claim is valid based on epoch and timestamp conditions.

However, it does not utilize any pause flag to enable it to stop the claiming process. The absence of such a pausing mechanism may prevent the protocol from mitigating the impact of a vulnerability or temporarily restricting users from claiming, especially during critical periods such as protocol upgrades or network instability.

Remediation

Add the `pause_claim` flag, which adds an extra layer of control and security to the protocol.

Patch

Fixed in 62f3a77

OS-HDL-SUG-02 | Division By Zero Error

Description

The issue is associated with the lack of protective measures in `set_validator_count` to prohibit the configuration of `validator_count` to zero.

sources/staking.move RUST

```rust
/// call this at the end of every epoch, and it can be called severaltimes in one
    ↪  epoch
public(friend) fun do_stake(
    staking: &mut Staking,
    wrapper: &mut SuiSystemState,
    validators: vector<address>,
    ctx: &mut TxContext,
) {
    [...]
    // calculate the amount to stake on each validator
    let validator_count = vector::length(&validators);

    // select maxmum config_validator_count validators
    let config_validator_count =config::get_validator_count(&staking.config);
    if (config_validator_count < validator_count) {
        validator_count = config_validator_count;
    };
    [...]
    let avg_amount = need_stake_amount / validator_count;
    [...]
}
```

This may result in a division by zero error in `do_stake` in the `staking` when attempting to calculate the average amount to stake on each validator, based on `need_stake_amount` and the selected validator count as shown above.

Remediation

Include a check in `set_validator_count` to verify `validator_count` is greater than zero.

Patch

Fixed in 5fae915

OS-HDL-SUG-03 | Code Maturity

Description

1. Within `table_queue`, it is advisable for `is_empty`, `borrow_front`, and `borrow_front_mut` to utilize `table::is_empty` for obtaining the result, rather than checking the length of the `TableQueue`.

sources/table_queue.move RUST

```rust
/// Return if the TableQueue is empty or not.
public fun is_empty<Element: store>(t: &TableQueue<Element>): bool {
    length(t) == 0
}
```

2. In `util`, utilize `mul_div` instead of duplicating the same code for such operations.

sources/util.move RUST

```rust
public fun get_sui_amount(exchange_rate: &PoolTokenExchangeRate,token_amount:
    ↪ u64): u64 {
    [...]
    let res = (sui_amount as u128)
            * (token_amount as u128)
            / (pool_token_amount as u128);
}
```

3. Within `staking`, it is more appropriate for `is_active_validator` to employ `vector::contains` instead of iterating through the vector to check the elements.

sources/util.move RUST

```rust
fun is_active_validator(validator: address,
    ↪ active_validators:&vector<address>): bool {
    [...]
    while (i < length) {
        if (validator == *vector::borrow(active_validators, i)) {
            return true
        };
        i = i + 1;
    };
    return false
}
```

Remediation

Utilize the functions mentioned above to enhance overall readability and mitigate code duplication.

Patch

Fixed in 5bcf233 and 56a221a

OS-HDL-SUG-04 | Missing Check

Description

Add a check in `do_validator_unstake` to ensure that `total_rewards` is greater than or equal to `left_rewards + withdraw_rewards` to verify that the program does not reduce the total rewards below the sum of rewards left in the pool and the rewards under withdrawal.

Remediation

Ensure the above check is implemented.

OS-HDL-SUG-05 | Inconsistent Storage Of Staked Coins

Description

The storing of staked SUI in two different locations by the `request_stake_coin` method may result in a potential issue. This method enables such storage based on whether the user has specified a validator or not, which may lead to a situation where a malicious user can prevent other users from withdrawing their funds immediately.

sources/staking.move RUST

```
/// stake and return haSUI
public fun request_stake_coin(wrapper: &mut SuiSystemState, staking:&mut Staking,
    ↪ input: Coin<SUI>, validator: address, ctx: &mutTxContext): Coin<HASUI> {
    [...]
    // user does not select a validator, or selects a invalid validator,just let
        ↪ protocol to select a validator
    if (validator == @0x0 || !is_active_validator(validator,&active_validators)) {
        // deposit sui to vault, and wait for staking later
        vault::deposit(&mut staking.sui_vault,coin::into_balance(input));
    } else { // user selects a validator
        save_user_selected_staking(staking, input, validator);
    };
    [...]
}
```

This occurs as `request_stake_coin` stores the staked SUI in `sui_vault` in the case a user did not specify any validator, chose an inactive validator, or if the chosen validator is 0x0. However, when an active validator is selected, the staked SUI is stored in `user_selected_validator_bals` as shown below.

sources/staking.move RUST

```
fun save_user_selected_staking(staking: &mut Staking, input: Coin<SUI>,validator:
    ↪ address) {
    [...]
    let validator_balance =
        ↪ vec_map::get_mut(&mutstaking.user_selected_validator_bals, &validator);
    balance::join(validator_balance, coin::into_balance(input));
}
```

Thus, this allows a user to choose an active validator while staking coins, which results in storing their staked coins in `user_selected_validator_bals`. Consequently, the user may unstake their SUI via `request_unstake_instant`, which withdraws the staked SUI stored in `sui_vault` instead of deducting from `request_unstake_instant`, where the user's stake is stored. This may drain staked SUI from `sui_vault`, thereby preventing users with staked coins in `sui_vault` from withdrawing due to lack of funds.

Remediation

Deduct staked coins from the correct storage location based on where the user's staked SUI is stored.

A | Vulnerability Rating Scale

We rated our findings according to the following scale. Vulnerabilities have immediate security implications. Informational findings may be found in the General Findings section.

Critical	Vulnerabilities that immediately result in a loss of user funds with minimal preconditions. Examples: • Misconfigured authority or access control validation. • Improperly designed economic incentives leading to loss of funds.
High	Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit. Examples: • Loss of funds requiring specific victim interactions. • Exploitation involving high capital requirement with respect to payout.
Medium	Vulnerabilities that may result in denial of service scenarios or degraded usability. Examples: • Computational limit exhaustion through malicious input. • Forced exceptions in the normal user flow.
Low	Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk. Examples: • Oracle manipulation with large capital requirements and multiple transactions.
Informational	Best practices to mitigate future security risks. These are classified as general findings. Examples: • Explicit assertion of critical internal invariants. • Improved input validation.

B | Procedure

As part of our standard auditing procedure, we split our analysis into two main sections: design and implementation.

When auditing the design of a program, we aim to ensure that the overall economic architecture is sound in the context of an on-chain program. In other words, there is no way to steal funds or deny service, ignoring any chain-specific quirks. This usually requires a deep understanding of the program's internal interactions, potential game theory implications, and general on-chain execution primitives.

One example of a design vulnerability would be an on-chain oracle that could be manipulated by flash loans or large deposits. Such a design would generally be unsound regardless of which chain the oracle is deployed on.

On the other hand, auditing the program's implementation requires a deep understanding of the chain's execution model. While this varies from chain to chain, some common implementation vulnerabilities include reentrancy, account ownership issues, arithmetic overflows, and rounding bugs.

As a general rule of thumb, implementation vulnerabilities tend to be more "checklist" style. In contrast, design vulnerabilities require a strong understanding of the underlying system and the various interactions: both with the user and cross-program.

As we approach any new target, we strive to comprehensively understand the program first. In our audits, we always approach targets with a team of auditors. This allows us to share thoughts and collaborate, picking up on details that the other missed.

While sometimes the line between design and implementation can be blurry, we hope this gives some insight into our auditing procedure and thought process.