Turbos Finance

Security Assessment

May 24th, 2024 — Prepared by OtterSec

Michał Bochnak embe221ed@osec.io

Bartłomiej Wierzbiński dark@osec.io

Robert Chen notdeghost@osec.io

Table of Contents

01 — Executive Summary

Overview

Turbos Finance engaged OtterSec to assess the `liquidity-vault` program. This assessment was conducted between May 1st and May 22nd, 2024. For more information on our auditing methodology, refer to Appendix B.

Key Findings

We produced 9 findings throughout this audit engagement.

In particular, we identified multiple vulnerabilities concerning the reward distribution process, including one where the reward calculation logic takes into consideration periods with zero reward emission when calculating accumulated reward, resulting in excessive rewards being released (OS-TBF-ADV-00), and another issue related to the double-counting of rewards due to the order of function calls, resulting in inaccurate tracking of released rewards (OS-TBF-ADV-02). Additionally, we highlighted a possible overflow risk due to a lack of validation of user-defined tick steps during the vault creation process (OS-TBF-ADV-01).

We also made recommendations around refactoring the code to mitigate possible security issues (OS-TBF-SUG-02) and emphasized the lack of proper validations in multiple areas within the code base (OS-TBF-SUG-01). We further suggested removing redundant code in adherence to best coding practices (OS-TBF-SUG-05).

02 — Scope

The source code was delivered to us in a Git repository at https://github.com/turbos-finance/liquidity-vault. This audit was performed against commit 5ed7096.

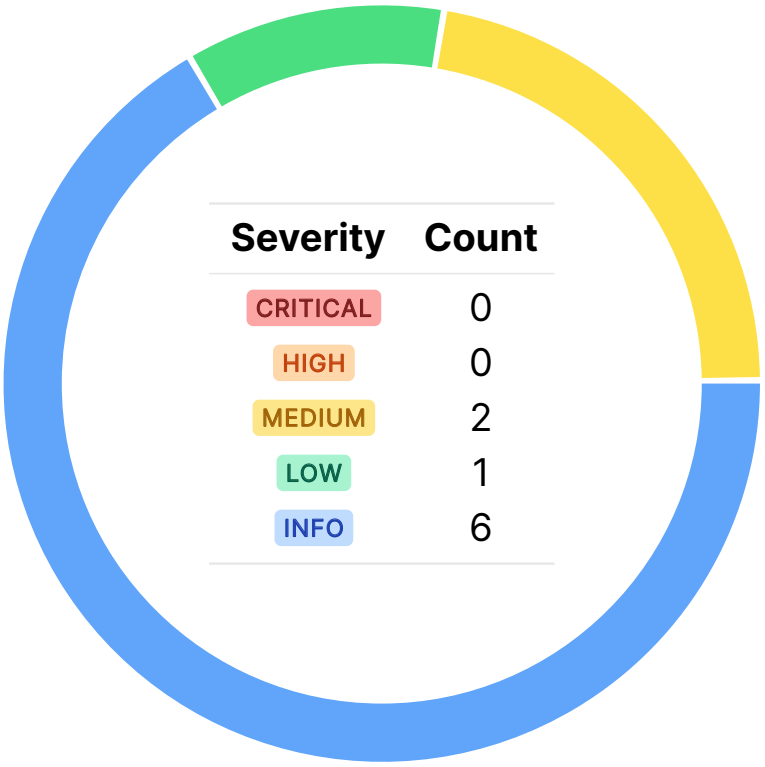
A brief description of the programs is as follows:

Name	Description
liquidity-vault	A framework for managing liquidity pools, strategies, rewards, and user interactions within the Turbos protocol. It allows users to participate in providing liquidity, earning rewards, and managing assets, facilitating operations such as opening, depositing, withdrawing, and closing vaults, as well as managing fees and rewards.

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	2
LOW	1
INFO	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-TBF-ADV-00	MEDIUM	RESOLVED	The reward calculation logic would include a zero reward emission period after updating the `emission_per_second` when calculating rewards. This overcalculates the total reward amount.
OS-TBF-ADV-01	MEDIUM	RESOLVED	Unvalidated user-defined steps in `vault` creation may overflow when multiplied by `strategy` values.
OS-TBF-ADV-02	LOW	RESOLVED	The current code in the `rewarder` double-counts rewards due to the order of function calls. This may result in inaccurate reward distribution and faster depletion of the reward pool.

Incorrectly Calculated Reward Period MEDIUM OS-TBF-ADV-00

Description

`accumulate_strategys_reward` and `accumulate_strategy_reward` update the `last_reward_time` if the `emission_per_second` is greater than zero. Thus, if the emission rate is set to zero for a period of time and then set to a positive value, the reward calculations will include the time elapsed during the zero-emission period.

>_ *rewarder.move* rust

```
public(friend) fun accumulate_strategys_reward<T>(manager: &mut RewarderManager, current_time:
    ↪  u64) {
    // Obtain a mutable reference to the specific rewarder by its type.
    let type_name = type_name::get<T>();
    let rewarder = linked_table::borrow_mut<TypeName, Rewarder>(&mut manager.rewarders,
        ↪  type_name);

    if (rewarder.emission_per_second == 0) {
        return;
    };
    [...]
}
```

Since `last_reward_time` is used to calculate the elapsed time for reward accumulation, including a period with zero emissions will result in an overcalculation of the rewards earned by strategies during that time.

Remediation

Update `last_reward_time` in both functions regardless of the current `emission_per_second` value. This ensures that the reward calculations always consider the total elapsed time, even during periods with zero emission.

Patch

Resolved in 971cfd9.

Missing Tick Step Validation MEDIUM OS-TBF-ADV-01

Description

In `open_vault` within `vault` module, the `base_tick_step` and `limit_tick_step` values are user-provided inputs and are not validated. This means a malicious user may enter substantially high values for these steps. Inside `check_rebalance`, these user-provided steps are multiplied by `tick_spacing`. This multiplication results in an integer overflow if the product (`limit_tick_step * tick_spacing` or `base_tick_step * tick_spacing`) is large. Consequently, the function may abort if an overflow occurs, effectively disrupting `check_rebalance_loop` and influencing the re-balancing logic.

>_ vault.move rust

```
public fun check_rebalance<CoinTypeA, CoinTypeB, FeeType>(
    global_config: &GlobalConfig,
    strategy: &mut Strategy,
    vault_id: ID,
    clmm_pool: &mut Pool<CoinTypeA, CoinTypeB, FeeType>,
    ctx: &mut TxContext
): CheckRebalance {
    [...]
    let base_tick_step = vault_info.base_tick_step;
    let limit_tick_step = vault_info.limit_tick_step;
    [...]
}
```

Remediation

Either validate the values of `base_tick_step` and `limit_tick_step` in `open_vault` or utilize the `strategy` values instead.

Patch

Resolved in 9333ce4

Double Counting Rewards LOW OS-TBF-ADV-02

Description

`rewarder::accumulate_rewarder_released` and `accumulate_strategy_reward` update the `total_reward_released` for the entire `rewarder` based on the emission rate and elapsed time. However, `accumulate_strategy_reward` does not update the `last_reward_time` for the `rewarder` unlike `accumulate_rewarder_released`.

>_ *rewarder.move* rust

```
public(friend)  fun accumulate_rewarder_released(rewarder: &mut Rewarder, current_time: u64) {
    if (rewarder.last_reward_time <= current_time) {
        let elapsed_time = current_time - rewarder.last_reward_time;
        if (rewarder.emission_per_second > 0) {
            let additional_rewards = rewarder.emission_per_second * elapsed_time;
                rewarder.total_reward_released = rewarder.total_reward_released +
                    ↪  (additional_rewards as u128);
        };
        rewarder.last_reward_time = current_time;
    }
}
```

Thus, when `strategy_rewards_settle` calls `accumulate_rewarder_released` for each `rewarder`, it updates the `total_reward_released` for the entire `rewarder` based on the current time and elapsed time since the `last_reward_time` of the `rewarder`. The subsequent call to `accumulate_strategy_reward` also calculates the elapsed time considering the last reward time of the `strategy`, which may be different from the last reward time of the `rewarder`.

>_ *rewarder.move* rust

```
public(friend)  fun accumulate_strategy_reward(
    rewarder: &mut Rewarder,
    strategy_id: ID,
    strategy_share: u128,
    current_time: u64
): u128 {
    [...]
    if (current_time > strategy_info.last_reward_time && rewarder.emission_per_second > 0) {
        [...]
        rewarder.total_reward_released = rewarder.total_reward_released + (new_rewards_total as
        u128);
    };
    [...]
}
```

Since `accumulate_strategy_reward` does not update the `last_reward_time` of the `rewarder`, the elapsed time it calculates may include a period already accounted for in the previous call to `accumulate_rewarder_released`. As a result, the `total_reward_released` for the `rewarder` increments twice for the same elapsed time, double-counting the `total_reward_released` value.

Remediation

Split the rewards calculations logic into two steps: strategy's rewards calculation and global rewarder's reward calculations.

Patch

Resolved in 971cfd9.

05 — General Findings

Here, we present a discussion of the general findings identified during our audit. While these findings do not pose an immediate security impact, they represent anti-patterns and could potentially lead to security issues in the future.

ID	Description
OS-TBF-SUG-00	A potential race condition within `vault::check_rebalance_loop` enables anyone to close an existing `vault`.
OS-TBF-SUG-01	Several functionalities must be refactored to include missing validations.
OS-TBF-SUG-02	Recommendations to mitigate possible security issues and implement proper checks.
OS-TBF-SUG-03	The code may be optimized for increased efficiency in multiple areas within the code base.
OS-TBF-SUG-04	Multiple functionalities lack a check to ensure a certain value is greater than zero.
OS-TBF-SUG-05	Suggestions regarding the removal of redundancy and ensuring adherence to best coding practices.

Race Condition

OS-TBF-SUG-00

Description

There is a potential race condition in `vault::check_rebalance_loop`, as it iterates through vaults with `vault_id_option`, which points to vaults within the `strategy`. Thus, another user may try to close the same `vault` (pointed to by `vault_id_option`). Since there is no lock on the `strategy`, the `vault` will be closed while `check_rebalance_loop` is still iterating through the loop.

Remediation

Lock the `strategy` before iterating through the vaults. This will ensure exclusive access to the `strategy` data while `check_rebalance_loop` is executing.

Missing Validations

OS-TBF-SUG-01

Description

1. `config::add_operator` and `config::set_tier` are missing a version validation check, resulting in possible version mismatch errors. The functions should call `checked_package_version(global_config)` to ensure the current package version in the `GlobalConfig` matches the constant defined in the module (`VERSION`).
2. `set_package_version` fails to explicitly check whether the new version is an upgrade (whether the new version is greater than the old version). If someone accidentally attempts to set a version lower than the current version (downgrade), the function will still update the configuration without raising any warnings. The function should check that the `new_version` is greater than `global_config.package_version` .
3. When the `vault` , `clmm_pool` , and `strategy` objects are passed as parameters to functions, explicitly validate their relationships to ensure data consistency and prevent possible manipulation of values.
4. `vault::collect_clmm_reward_direct_return` should validate the `recipient` address before using it, as the `recipient` argument may be manipulated to spoof this value in the emitted event.

>_ vault.move rust

```
public fun collect_clmm_reward_direct_return<CoinTypeA, CoinTypeB, FeeType,
    ↪  RewardCoinType>(
    global_config: &GlobalConfig,
    strategy: &mut Strategy,
    vault: &Vault,
    clmm_pool: &mut Pool<CoinTypeA, CoinTypeB, FeeType>,
    clmm_positions: &mut Positions,
    clmm_reward_vault: &mut PoolRewardVault<RewardCoinType>,
    clmm_reward_index: u64,
    recipient: address,
    clock: &Clock,
    clmm_versioned: &Versioned,
    ctx: &mut TxContext
): Coin<RewardCoinType> {
    config::checked_package_version(global_config);
    assert!(strategy.status == 0, EStrategyLocked);
    assert!(strategy.clmm_pool_id == object::id(clmm_pool), ENotTheCorrespondingStrategy);
    [...]
}
```

Remediation

Implement the above-mentioned modifications to the code base.

Patch

Resolved in 971cfd9.

Code Refactoring

OS-TBF-SUG-02

Description

1. `vault::check_rebalance` and `vault::check_rebalance_loop` should avoid utilizing mutable references (`&mut`) for its input parameters, to ensure there will be no changes to these objects in these functions.
2. Threshold setters, in the context of rebalancing strategies, should validate the new threshold value before applying it so that the system becomes more resistant to manipulation attempts and errors due to accidental input invalid values.
3. `vault::close_vault` does not explicitly remove the entry for the closed `vault` from the `strategy.accounts` linked table, wasting storage.

>_ *vault.move* rust

```rust
public fun close_vault<CoinTypeA, CoinTypeB>(
    global_config: &GlobalConfig,
    strategy: &mut Strategy,
    vault: Vault,
    ctx: &mut TxContext
) {
    [...]
    assert!(get_vault_balance<CoinTypeA>(strategy, vault_id) == 0, ECoinABalanceGtZero);
    assert!(get_vault_balance<CoinTypeB>(strategy, vault_id) == 0, ECoinBBalanceGtZero);
    remove_vault_info(strategy, vault_id);
    [...]
}
```

Remediation

1. Ensure `vault::check_rebalance` and `vault::check_rebalance_loop` do not take mutable parameters.
2. Validate the threshold setter functions within `vault`.
3. Remove the `vault` entry from `strategy.accounts` on closing a `vault`.

Code Optimizations

OS-TBF-SUG-03

Description

1. Within `rewarder`, `add_reward_black_list` may be improved by checking if an address is already present in the blacklist before adding it to avoid duplicate entries. Furthermore, in `vault`, the reward information should be updated immediately after adding or removing a vault to or from the blacklist to ensure data consistency.

   >_ rewarder.move rust

   ```
   public fun add_reward_black_list(
       operator_cap: &OperatorCap,
       config: &GlobalConfig,
       manager: &mut RewarderManager,
       addreses: vector<address>,
       ctx: &mut TxContext
   ) {
       config::checked_package_version(config);
       let operator_address = object::id_address<OperatorCap>(operator_cap);
       config::check_reward_manager_role(config, operator_address);
       vector::append(&mut manager.black_list, addreses);
   }
   ```

2. Ensure that functions do not emit an event if the action has not been performed, as this will result in the logging of false information, especially affecting off-chain applications monitoring event logs.
3. The assert statement in `rewarder::create_rewarder` checks for an existing `rewarder` (`linked_table::contains`). This may be placed earlier to improve efficiency and prevent unnecessary operations in case of a revert.
4. `vault::rebalance` performs several operations before checking the value of `rebalance`, which will result in multiple unnecessary operations if the `rebalance` check were to fail. Move this check to the beginning of the function before any significant operations are executed.

Remediation

Apply the above optimizations to the code base.

Patch

Resolved in 971cfd9.

Zero Value Checks

OS-TBF-SUG-04

Description

1. `rewarder::deposit_rewarder` relies on `coin::value<T>(&coin)` to obtain the deposit amount. However, there is no check on the validity of the coin object itself, enabling the creation of a coin object with a fake type (`T`) and a value of zero using `coin::zero<u64>()` for example.

>_ rewarder.move rust

```
public fun deposit_rewarder<T>(
    config: &GlobalConfig,
    manager: &mut RewarderManager,
    coin: Coin<T>,
    ctx: &mut TxContext
) {
    config::checked_package_version(config);
    let type_name = type_name::get<T>();
    if (!bag::contains<TypeName>(&manager.vault, type_name)) {
            let zero_balance = balance::zero<T>();
            bag::add<TypeName, Balance<T>>(&mut manager.vault, type_name, zero_balance);
    };

    let before_amount = coin::value<T>(&coin);
    let after_amount = balance::join<T>(bag::borrow_mut<TypeName, Balance<T>>(&mut
        ↪  manager.vault, type_name), coin::into_balance(coin));
    [...]
}
```

2. `rewarder::accumulate_strategy_strategy_reward` will fail due to division by zero if `strategy_share` is equal to zero.
3. `vault::check_rebalance_loop` utilizes a `limit` parameter to control the maximum number of vaults processed in a single call. if `limit == 0` the loop condition `rebalance_vaults_lenght < limit` will always be false, and the loop will never be executed, therefore the offchain logic that executes `check_rebalance_loop` will run infinitely because the returned `vault_id_option` will be the same. Therefore `check_rebalance_loop` should return early to avoid unnecessary operations.

Remediation

1. Include a check to ensure the deposited coin has a value greater than zero.
2. Add a check to verify `strategy_share` is greater than zero before the division.

3. Include a check for `limit > 0` at the beginning of the function to ensure the loop only runs if a positive `limit` is provided.

Patch

Resolved in 971cfd9.

Code Maturity

OS-TBF-SUG-05

Description

1. In `rewarder`, `strategy_rewards_settle` may be enhanced by utilizing the existing `is_strategy_registered` instead of conducting a manual check with `linked_table::contains` to prevent redundancy. Since `is_strategy_registered` already includes the logic for verifying if a strategy exists in `manager.strategy_shares`, this approach would streamline the process.

>_ rewarder.move rust

```
public(friend) fun strategy_rewards_settle(
    manager: &mut RewarderManager,
    rewarder_types: vector<TypeName>,
    strategy_id: ID,
    clock: &Clock
): VecMap<TypeName, u128> {
    assert!(linked_table::contains<ID, u128>(&manager.strategy_shares, strategy_id),
        ↪  EStrategyNotRegistered);

    [...]
}
```

2. There are currently several instances of duplicate data (`sqrt_price`, `liquidity`, `base_clmm_position_id`, and `limit_clmm_position_id`). Updates to these values may not be consistently reflected, resulting in unnecessary complexities when storing the same data in multiple locations.
3. `collect_reward` is not called when closing a `vault` in the current implementation. As a result, any unclaimed rewards associated with the `vault` will be left behind, negatively impacting the users.

Remediation

1. Utilize `is_strategy_registered` instead of `linked_table::contains`.
2. Use `turbos_clmm` to retrieve the above-mentioned values ensuring they are calculated and retrieved from a single source.
3. Call `collect_reward` within `close_vault`.

Patch

Issue #1 and #2 resolved in 971cfd9

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.

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.

INFO

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 others may have 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.