Pontem Harvest

Audit

Presented by:

OtterSec contact@osec.io

Fineas Silaghi fedex@osec.io

Robert Chen notdeghost@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Pontem engaged OtterSec to perform an assessment of the `harvest` program. This assessment was conducted between November 14th and November 18th, 2022.

Critical vulnerabilities were communicated to the team prior to the delivery of the report to speed up remediation. After delivering our audit report, we worked closely with the team to streamline patches and confirm remediation. We delivered final confirmation of the patches January 2nd, 2023.

Key Findings

Over the course of this audit engagement, we produced 5 findings total.

In particular, we identified a couple of ways in which unintended rewards behaviour could occur, such as permanently locked rewards (OS-PHV-SUG-01) and the possibility of not being able to harvest rewards (OS-PHV-SUG-03).

We also made recommendations around extra functionality that could be included (OS-PHV-SUG-02).

Overall, we commend the Pontem team for being responsive and knowledgeable throughout the audit.

02 | Scope

The source code was delivered to us in a git repository at github.com/pontem-network/pontem-network-harvest. This audit was performed against commit `d940288`.

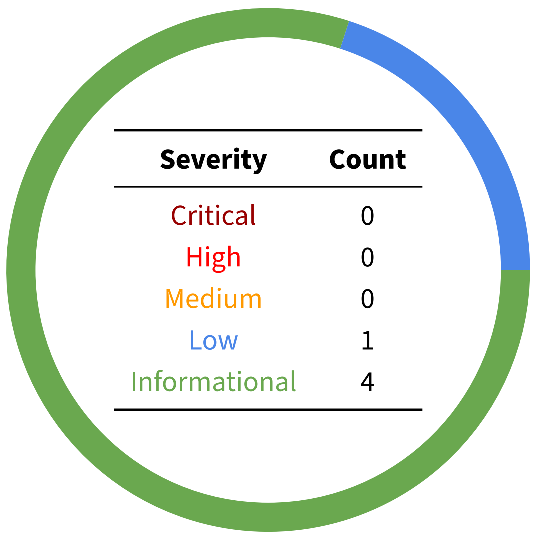
A brief description of the programs is as follows.

Name	Description
harvest	Staking protocol where delegators can earn rewards.

03 | Findings

Overall, we report 5 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 don't have an immediate impact but will help mitigate future vulnerabilities.



Severity	Count
Critical	0
High	0
Medium	0
Low	1
Informational	4

04 | Vulnerabilities

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

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-PHV-ADV-00	Low	Resolved	The calculation of `boosted_amount` is vulnerable to integer overflow.

OS-PHV-ADV-00 [low] [resolved] | Integer Overflow In Boosted_Amount

Description

Boosting is a staking feature that allows stakers under specifically configured pools to stake a token from a specific collection and increase the total staked amount by a percentage.

If a token has been staked, the total boosted amount will be recalculated upon staking, unstaking, or boosting.

stake.move RUST

```rust
// update user stake and pool after stake boost
user_stake.boosted_amount = (user_stake.amount * boost_percent) / 100;
pool.total_boosted = pool.total_boosted + user_stake.boosted_amount;
```

This calculation can potentially overflow if the user tries to stake a large amount of coins. This is because the boost percent times user amount could potentially exceed the maximum bounds of a u64, resulting in a forced abort.

Remediation

It is recommended to convert `boosted_amount` to the `u128` data type.

Patch

Resolved in #37.

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 antipatterns and could lead to security issues in the future.

ID	Description
OS-PHV-SUG-00	In case of emergency, users are not able to harvest their rewards.
OS-PHV-SUG-01	Potential lack of precision in pool rewards
OS-PHV-SUG-02	Users who unstaked all of their coins cannot leave the pool.
OS-PHV-SUG-03	Permissionless rewarding could lead to insufficient amounts of rewards.

OS-PHV-SUG-00 | Missing Harvest Functionality On Emergency

Description

The Harvest protocol allows users to stake coins in exchange for rewards. The accumulated rewards can be harvested by the stakers at any given time, if enough rewards are available.

sources/stake.move RUST

```rust
public fun harvest<S, R>(user: &signer, pool_addr: address): Coin<R>
    ↪ acquires StakePool {
        assert!(exists<StakePool<S, R>>(pool_addr), ERR_NO_POOL);

    [...]

    event::emit_event<HarvestEvent>(
        &mut pool.harvest_events,
        HarvestEvent { user_address: user_addr, amount:
    ↪   earned_to_withdraw },
    );

    coin::extract(&mut pool.reward_coins, earned_to_withdraw)
}
```

However, pools can enter a local or global emergency state. During the emergency state, stakers are allowed to unstake their coins, despite the lockup period not being finished, while new stake cannot be added anymore. On the other hand, depositing and harvesting rewards become restricted actions, which could be harmful if there are rewards in the pool, as this would lock them forever.

sources/stake.move RUST

```rust
/// This field set to `true` only in case of emergency:
/// * only `emergency_unstake()` operation is available in the state of
    ↪ emergency
emergency_locked: bool,

[...]

public fun harvest<S, R>(user: &signer, pool_addr: address): Coin<R>
    ↪ acquires StakePool {
        assert!(exists<StakePool<S, R>>(pool_addr), ERR_NO_POOL);

        let pool = borrow_global_mut<StakePool<S, R>>(pool_addr);
        assert!(!is_emergency_inner(pool), ERR_EMERGENCY);
```

Remediation

The unwanted scenario could be avoided by taking one of the following two approaches:

- Allow stakers to harvest during an emergency state by removing the `is_emergency_inner` check.
- Allow the protocol to collect the rewards in a protocol treasury.

Patch

The issue was fixed by creating a treasury admin account, which is allowed to harvest the rewards in case of emergency or if 3 months have passed since the duration of the pool has expired.

sources/stake.move RUST

```rust
public fun withdraw_to_treasury<S, R>(treasury: &signer, pool_addr:
    ↪  address, amount: u64): Coin<R> acquires StakePool {
    assert!(exists<StakePool<S, R>>(pool_addr), ERR_NO_POOL);
    assert!(signer::address_of(treasury) ==
    ↪  stake_config::get_treasury_admin_address(), ERR_NOT_TREASURY);

    let pool = borrow_global_mut<StakePool<S, R>>(pool_addr);

    if (!is_emergency_inner(pool)) {
        let now = timestamp::now_seconds();
        assert!(now >= (pool.end_timestamp +
    ↪  WITHDRAW_REWARD_PERIOD_IN_SECONDS), ERR_NOT_WITHDRAW_PERIOD);
    };

    coin::extract(&mut pool.reward_coins, amount)
}
```

OS-PHV-SUG-01 | Potential Reward Precision Issue

Description

When updating rewards, accumulated rewards are calculated via the following formula.

```rust
let total_rewards = (pool.reward_per_sec as u128) * (seconds_passed
↪  as u128) * pool.stake_scale;
total_rewards / total_boosted_stake
```

`stake_scale` is hardcoded to

```rust
stake_scale: math128::pow(10, (coin::decimals<S>() as u128)),
```

When distributing a small amount of rewards across a large amount of stake, this division could potentially round towards zero, meaning that users would not receive their rewards.

Remediation

Allow `stake_scale` to be specified as an external parameter to enable arbitrary stake precision.

Patch

Resolved in #40.

OS-PHV-SUG-02 | Inactive Users In The Pool

Description

After the lockup period has passed (one week), users are allowed to unstake all of their coins or just part of them. In the case that the user unstakes all their coins, there are no possibilities to leave the `StakePool`, unless an emergency occurs. In case of an emergency, a user can perform an `emergency_unstake`, which will automatically remove them from the pool

dgen/sources/dgen.move RUST

```rust
let user_stake = table::remove(&mut pool.stakes, user_addr);
let UserStake {
    amount,
    unobtainable_reward: _,
    earned_reward: _,
    unlock_time: _,
    nft,
    boosted_amount: _
} = user_stake;
```

But if no emergency occurs, after withdrawing all the stake, the user won't be able to leave the pool. They will be part of `StakePool.stakers` forever.

Remediation

Adding the possibility to leave the pool, via an extra parameter or a separate function, when all of the stake and rewards have been collected could help avoid this scenario.

OS-PHV-SUG-03 | Missing Staker Reward Ensurance

Description

Users who have staked coins to the pool earn rewards at a custom `reward_per_sec` rate.

sources/stake.move RUST

```rust
let total_rewards = to_u128(pool.reward_per_sec) *
    ↪  to_u128(seconds_passed) * to_u128(pool.stake_scale);
```

The rewards are deposited in a permissionless manner in the pool.

sources/stake.move RUST

```rust
/// Depositing reward coins to specific pool.
/// * `pool_addr` - address under which pool are stored.
/// * `coins` - R coins which are used in distribution as reward.
public fun deposit_reward_coins<S, R>(pool_addr: address, coins: Coin<R>)
    ↪  acquires StakePool {
    assert!(exists<StakePool<S, R>>(pool_addr), ERR_NO_POOL);

    [...]

    coin::merge(&mut pool.reward_coins, coins);

    event::emit_event<DepositRewardEvent>(
        &mut pool.deposit_events,
        DepositRewardEvent { amount },
    );
}
```

Since there are no enforcements at the smart contract level, an arbitrary number of rewards can be deposited. Rewards redemption works by the principle of First In First Served. This aspect could become an issue when not enough rewards are being deposited, and thus, not all users get to harvest rewards.

Remediation

Display a warning on the UI if the pool doesn't have enough rewards to make sure that the pool owners are aware of the situation and deposit more rewards.

A | Vulnerability Rating Scale

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

Critical	Vulnerabilities that immediately lead to 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 could lead to 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 could lead to denial of service scenarios or degraded usability. Examples: • Malicious input that causes computational limit exhaustion • Forced exceptions in normal user flow
Low	Low probability vulnerabilities which could still be 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