Pontem Network - Liquidswap Harvest

Move Smart Contract Security Audit

Prepared by: **Halborn**

Date of Engagement: **November 21st, 2022 - December 16th, 2022**

Visit: **Halborn.com**

DOCUMENT REVISION HISTORY

VERSION	MODIFICATION	DATE	AUTHOR
0.1	Document Creation	11/21/2022	Elena Maranon
0.2	Document Update	11/29/2022	Elena Maranon
0.3	Draft Version	12/01/2022	Elena Maranon
0.4	Draft Review	12/01/2022	Gabi Urrutia
0.5	Draft Update	12/16/2022	Elena Maranon
0.6	Draft Review	12/20/2022	Gabi Urrutia
1.0	Remediation Plan	01/09/2023	Elena Maranon
1.1	Remediation Plan Review	01/11/2023	Luis Quispe Gonzales
1.2	Remediation Plan Review	01/11/2023	Gabi Urrutia

CONTACTS

CONTACT	COMPANY	EMAIL
Rob Behnke	Halborn	Rob.Behnke@halborn.com
Steven Walbroehl	Halborn	Steven.Walbroehl@halborn.com
Gabi Urrutia	Halborn	Gabi.Urrutia@halborn.com
Luis Quispe Gonzales	Halborn	Luis.QuispeGonzales@halborn.com
Elena Maranon	Halborn	Elena.Maranon@halborn.com

EXECUTIVE OVERVIEW

1.1 INTRODUCTION

Pontem Network engaged Halborn to conduct a security audit on their smart contracts beginning on November 21st, 2022 and ending on December 16th, 2022. The security assessment was scoped to the smart contracts provided in the GitHub repository Harvest, commit hashes and further details can be found in the Scope section of this report.

The audit has been focused on the staking rewards system of the Liquidswap project, as well as the new NFT Boost feature recently added.

1.2 AUDIT SUMMARY

The team at Halborn assigned a full-time security engineer to audit the security of the smart contract. The security engineer is a blockchain and smart-contract security expert with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this audit is to:

- Ensure that smart contract functions operate as intended
- Identify potential security issues with the smart contracts

In summary, Halborn identified some improvements to reduce the likelihood and impact of risks, which were partially addressed by Pontem Network. The main ones are the following:

- Add more validations before mathematical operations in order to maintain a minimum level of rewards or avoid them to be blocked.
- Allow rewards depositors to withdraw their deposits in case of emergency state.
- Avoid edge situations where users may not receive any reward after staking time.
- Add a two steps procedure to transfer privilege addresses in order to avoid unwilling errors that could lead to security problems.

- Add the ability to revoke an emergency state in case of mistake or false alarm.
- In case of emergency, add a last harvest operation into emergency_unstake in order to recover the last earned rewards until that moment.
- Add the ability to remove an empty Stake Pool in case the creator wants to create a new one again.
- Check if Stake Pool duration has ended before adding a NFT boost.

1.3 TEST APPROACH & METHODOLOGY

RISK METHODOLOGY:

Vulnerabilities or issues observed by Halborn are ranked based on the risk assessment methodology by measuring the **LIKELIHOOD** of a security incident and the **IMPACT** should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL

10 - CRITICAL
9 - 8 - HIGH

7 - 6 - MEDIUM

5 - 4 - LOW

3 - 1 - VERY LOW AND INFORMATIONAL

1.4 SCOPE

First round of testing (Nov 21st - Dec 1st)

1. Repository: harvest
2. Commit ID: 1bc74e2
3. Modules in scope:
   - dgen
   - stake
   - stake_config
   - scripts

Second round of testing (Dec 12th - Dec 16th): NFT Boost added

1. Repository: harvest
2. Commit ID: 37d976a
3. Modules in scope:
   - dgen
   - stake
   - stake_config
   - scripts

Out-of-scope: External libraries and financial related attacks.

2. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
0	0	1	6	1

LIKELIHOOD

IMPACT

IMPACT	🟨 (HAL-01)	🟨	🟥	🟥	🟪
	🟩 (HAL-04) (HAL-05)	🟨	🟨	🟥	🟥
	🟩 (HAL-06)	🟩 (HAL-02) (HAL-03)	🟨	🟨	🟥
	⬜	🟩 (HAL-07)	🟩	🟨	🟨
	⬜ (HAL-08)	⬜	🟩	🟩	🟨

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
(HAL-01) ALL REWARDS CAN BE WITHDRAWN BY MARKET TREASURY	Medium	RISK ACCEPTED
(HAL-02) REWARDS CAN BE BLOCKED	Low	RISK ACCEPTED
(HAL-03) SOME USERS MIGHT NOT OBTAIN ANY REWARD	Low	RISK ACCEPTED
(HAL-04) PRIVILEGED ADDRESS TRANSFERRED WITHOUT CONFIRMATION	Low	RISK ACCEPTED
(HAL-05) EMERGENCY STATE CANNOT BE REVOKED	Low	RISK ACCEPTED
(HAL-06) HARVESTED REWARDS ARE LOST ON EMERGENCY	Low	RISK ACCEPTED
(HAL-07) NO OPTION TO DESTROY AN EMPTY STAKE POOL	Low	RISK ACCEPTED
(HAL-08) NFT BOOST CAN BE PERFORMED AFTER THE STAKE POOL ENDS	Informational	ACKNOWLEDGED

FINDINGS & TECH DETAILS

3.1 (HAL-01) ALL REWARDS CAN BE WITHDRAWN BY MARKET TREASURY - MEDIUM

Description:

The withdraw_to_treasury function from **stake** module allows the treasury account to withdraw all the remaining rewards after 3 months from the end of the Stake Pool duration and also to withdraw all the non-harvested rewards of the pool in case of emergency state.

The treasury account is configured during the initialization of the **stake_config** module by the @stake_emergency_admin account, which is the same account responsible for enabling the global emergency state.

A potential malicious stake_emergency_admin account could configure a treasury account under its control, enable the global emergency state and withdraw the rewards from all the Stake Pools in the market.

It should be noted that the stake_emergency_admin account is managed by a multisig wallet (3 participants, 2 signatures needed) which decreases the likelihood of the finding.

Code Location:

The withdraw_to_treasury function from **stake** module:

Listing 1: sources/stake.move (Lines 377-380,382)

```
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)
}
```

Risk Level:

Likelihood - 1
Impact - 5

Recommendation:

It is recommended to only allow the depositors of the rewards to withdraw them in case of emergency and leave to the treasury only those user rewards unclaimed for a period longer than 3 months.

Remediation plan:

RISK ACCEPTED: The Pontem Network team accepted the risk of this finding.

3.2 (HAL-02) REWARDS CAN BE BLOCKED - LOW

Description:

The update_accum_reward and accum_rewards_since_last_updated functions from **stake** module are responsible for the calculation of the accumulated rewards factor, which multiplied by the amount staked will result in the earned reward per user.

That factor is accumulative in time, adding the new value to the oldest one each time it is updated. On the other hand, the unobtainable_reward parameter from UserStake resource keeps the track of the rewards that the user cannot request.

The main points for rewards calculation could be summarized in the next mathematical operations:

$$\text{AccumulatedDelta} = \frac{\text{RewardsPerSecond} \times \text{SecondsElapsed}}{(\text{TotalStaked} + \text{TotalBoosted})}$$

$$\text{AccumulatedNew} = \text{AccumulatedDelta} + \text{AccumulatedOld}$$

$$\text{EarnedReward} = (\text{AccumulatedNew} \times (\text{AmountStaked} + \text{AmountBoosted})) - \text{Unobtainable}$$

There are some edge cases where the rewards could be totally blocked during the complete pool duration to all the users or some of them.

Analyzing the formula of AccumulatedDelta, if (TotalStaked+TotalBoosted) > RewardsPerSecond * SecondsElapsed the result is truncated and equal to 0 because of working with integers instead of floating numbers, meanwhile the pool.last_updated timestamp is still updating.

This situation can either be triggered unwillingly when there are many deposits in a short period of time (so that the numerator, based on the elapsed time since the last one, is not big enough), or either on purpose by a malicious actor depositing 1 token regularly, updating the last_updated timestamp while truncating the rewards for that period of time, therefore depriving the users from their rewards.

Also exists another way of blocking the rewards independently of time, which is being the first user of the staking pool and depositing a high amount of stake at once, higher than the product of RewardsPerSecond * PoolDuration.

In all these scenarios, the blocked rewards will be stuck in the pool until they could be recovered by the treasury account 3 months after the pool duration ends.

The following Proofs of Concept show an example of two of the above-mentioned scenarios.

Code Location:

The update_accum_reward function from **stake** module to update the accumulated rewards' parameter:

Listing 2: sources/stake.move (Lines 516,517,519)

```
515 fun update_accum_reward<S, R>(pool: &mut StakePool<S, R>) {
516     let current_time = get_time_for_last_update(pool);
517     let new_accum_rewards = accum_rewards_since_last_updated(pool,
  ↳ current_time);
518
519     pool.last_updated = current_time;
520
521     if (new_accum_rewards != 0) {
522         pool.accum_reward = pool.accum_reward + new_accum_rewards;
523     };
524 }
```

The accum_rewards_since_last_update function from **stake** module to update the accumulated rewards' parameter:

Listing 3: sources/stake.move (Lines 531,532,537,538)

```
530 fun accum_rewards_since_last_updated<S, R>(pool: &StakePool<S, R>, current_time: u64): u128 {
531     let seconds_passed = current_time - pool.last_updated;
532     if (seconds_passed == 0) return 0;
533
534     let total_boosted_stake = pool_total_staked_with_boosted(pool);
535     if (total_boosted_stake == 0) return 0;
536
537     let total_rewards = to_u128(pool.reward_per_sec) * to_u128(seconds_passed) * to_u128(pool.stake_scale);
538     total_rewards / to_u128(total_stake)
539 }
```

Proof of Concept 1 - Non-intentional blocking:

The scenario used for this proof of concept consists on: 100 reward coins, 100 seconds pool duration (1 reward coin per second) and 3 users. No NFT Boost applied.

After 15 seconds, the first user makes an initial deposit of 20 APT coins, 15 seconds later the second user makes a deposit of 30APT coins and 25 seconds later the third user makes a deposit of 10APT coins.

This makes a total stake of 60APT coins and 0 accumulated rewards because of the time constraints. Because of the remaining time is 45 seconds until the end of the pool duration, there will be no more rewards for any of the users because the update of accumulated rewards parameter will return zero.

Listing 4: PoC_script

```
amount='100'
duration='100'
pool_address=$POOL_ADDRESS
#Stake Pool initiated
aptos move run --profile dgen_coin --function-id 'harvest::scripts::register_pool' --args 'u64:'$amount --args 'u64:'$duration --type-args $typeS --type-args $typeR --assume-yes
```

```
#===============
echo 'Wait 15 seconds since the beginning'
sleep 15

echo 'User1 stakes 20 APT'
amount='2000000000'
aptos move run --profile user --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

echo '15 seconds passed for a total stake of 20APT'
sleep 15

echo 'User2 stakes 30 APT'
amount='3000000000'
aptos move run --profile user2 --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

echo '25 seconds passed for a total stake of 50APT'
sleep 25

echo 'User3 stakes 10 APT'
amount='1000000000'
aptos move run --profile user3 --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

#===== Harvest time
echo '55 seconds has passed for a total stake of 60APT, only 45
↳ seconds remains of pool duration'

echo 'User try to harvest rewards --> NOTHING TO HARVEST ERROR'
aptos move run --profile user --function-id 'harvest::scripts::
↳ harvest' --args 'address:'$pool_address --type-args $typeS --type-
↳ args $typeR --assume-yes

echo 'User2 try to harvest rewards --> NOTHING TO HARVEST ERROR'
aptos move run --profile user2 --function-id 'harvest::scripts::
↳ harvest' --args 'address:'$pool_address --type-args $typeS --type-
↳ args $typeR --assume-yes

echo 'User3 try to harvest rewards --> NOTHING TO HARVEST ERROR'
aptos move run --profile user3 --function-id 'harvest::scripts::
```

```
↳ harvest' --args 'address:'$pool_address --type-args $typeS --type-
↳ args $typeR --assume-yes
```

Proof of Concept 2 - Intentional blocking:

The scenario used for this proof of concept consists on: 100 reward coins, 100 seconds pool duration (1 reward coin per second) and 2 users. No NFT Boost applied.

The first user makes an initial deposit of 10 APT coins and waits until the second user makes another deposit of 10APT coins. Since the time passed between both deposits is 20 seconds, the accumulated rewards factor will be 2.

From that moment, the first user makes small deposits of 1 APT coin in order to freeze the accumulated rewards factor until the end of pool duration. After that, the first user will harvest a reward of 20 APT coins, while the second user will harvest nothing. The reason is that the accumulated rewards factor has been frozen since the moment the second user joined to the market, so the rewards accumulated are annulled by the unobtainable parameter.

Listing 5: PoC_script

```
amount='100'
duration='100'
pool_address=$POOL_ADDRESS
#Init Stake Pool
aptos move run --profile dgen_coin --function-id 'harvest::scripts
↳ ::register_pool' --args 'u64:'$amount --args 'u64:'$duration --
↳ type-args $typeS --type-args $typeR --assume-yes
#==========================
echo 'User1 stakes 10 APT at the beggining'
amount='1000000000'
aptos move run --profile user --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

echo '20 seconds passed'
sleep 20
```

```

echo 'User2 stake 20 APT coins'
amount='2000000000'
aptos move run --profile user2 --function-id 'harvest::scripts::stake' --args 'address:'$pool_address --args 'u64:'$amount --type-args $typeS --type-args $typeR --assume-yes

echo '25 seconds passed for a total stake of 30'
sleep 25

echo 'User1 stake 1 APT coin to update the pool.last_updated value'
pool_address=$dgen
amount='100000000'
aptos move run --profile user --function-id 'harvest::scripts::stake' --args 'address:'$pool_address --args 'u64:'$amount --type-args $typeS --type-args $typeR --assume-yes

echo '25 seconds passed for a total stake of 31'
sleep 25

echo 'User1 stake 1 APT coin to update the pool.last_updated value'
pool_address=$dgen
amount='100000000'
aptos move run --profile user --function-id 'harvest::scripts::stake' --args 'address:'$pool_address --args 'u64:'$amount --type-args $typeS --type-args $typeR --assume-yes

echo '25 seconds passed for a total stake of 32'
sleep 25

echo 'User1 try to harvest rewards --> HARVEST 20 reward coins'
aptos move run --profile user --function-id 'harvest::scripts::harvest' --args 'address:'$pool_address --type-args $typeS --type-args $typeR --assume-yes

sleep 10

echo 'User2 try to harvest rewards -> NOTHING TO HARVEST ERROR'
aptos move run --profile user --function-id 'harvest::scripts::harvest' --args 'address:'$pool_address --type-args $typeS --type-args $typeR --assume-yes
```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended to add more validations for edge cases to avoid this kind of scenarios. In addition, the fact that accumulated rewards are only updated by operations performed by external users could lead to manipulations, as it was shown in Proof of Concept 2.

The total staked contained in the pool and the new stakes should be controlled in order to not reduce the rewards to zero. A staking pool with zero rewards would have a negative impact for the users, losing the opportunity to have their funds staked in more profitable markets and, by extension, for the company image.

Remediation plan:

RISK ACCEPTED: The Pontem Network team has implemented a fix that partially solves the situation, decreasing the risk from **Medium** to **Low**.

The code was modified to include a scaling factor to add more precision to the accumulated_rewards calculation, avoiding some situations where the value would freeze due to truncated integers. This scaling factor modifies the condition needed to enter truncated values:

$$\text{TotalStake} + \text{TotalBoosted} > \text{RewardsPerSecond} \times \text{SecondsElapsed} \times \text{RewardScale}$$

However, there are still some edge cases where the proofs of concept shown above are still successful. For example, considering that the reward_scale value is just 100 because the reward token has 10 decimals (which might not be the common thing, but it is allowed by the code), both PoCs will still work by simply multiplying the deposited amounts by 100, which is not an unrealistic amount. For an average 8 decimals token, the multiplying factor would be 10.000.

Of course, this measure reduces the likelihood, but does not eliminate the possibility of occurrence, so the risk has been reduced to **Low**, but not completely eliminated.

3.3 (HAL-03) SOME USERS MIGHT NOT OBTAIN ANY REWARD - LOW

Description:

The update_accum_rewards function from **stake** module allows to update the accum_rewards factor in order to calculate the rewards earned by users. This function is called on the main operations of the stake module (stake, unstake and harvest) and it is responsible to update the parameter pool .last_updated from Stake Pool resource, apart from other operations.

In relation with the first finding, there is a point in time from which the users who deposit the stake into the pool will not receive any reward. The seconds left to the end of the pool will be smaller than the division of (TotalStake+TotalBoosted)/ RewardsPerSec, then, the accumulated rewards will not be increased anymore and the obtained rewards will be annulled by the unobtainable parameter of each UserStake. As it was mentioned in the first finding, this is caused by the maximum limit in the pool.last_update parameter.

Code Location:

The update_accum_reward function from **stake** module:

Listing 6: sources/stake.move (Line 516)

```
515 fun update_accum_reward<S, R>(pool: &mut StakePool<S, R>) {
516     let current_time = get_time_for_last_update(pool);
517     let new_accum_rewards = accum_rewards_since_last_updated(pool,
    ↳ current_time);
518
519     pool.last_updated = current_time;
520
521     if (new_accum_rewards != 0) {
522         pool.accum_reward = pool.accum_reward + new_accum_rewards;
523     };
524 }
```

The get_time_for_last_update function from **stake** module in charge of calculate the pool.last_update value:

Listing 7: sources/stake.move (Line 580)

```
579 fun get_time_for_last_update<S, R>(pool: &StakePool<S, R>): u64 {
580     math64::min(pool.end_timestamp, timestamp::now_seconds())
581 }
```

The accum_rewards_since_last_update from **stake** module, responsible of updating the accumulated rewards parameter:

Listing 8: sources/stake.move (Lines 531,532)

```
530 fun accum_rewards_since_last_updated<S, R>(pool: &StakePool<S, R>,
 ↳  current_time: u64): u128 {
531     let seconds_passed = current_time - pool.last_updated;
532     if (seconds_passed == 0) return 0;
533
534     let total_boosted_stake = pool_total_staked_with_boosted(pool)
 ↳ ;
535     if (total_boosted_stake == 0) return 0;
536
537     let total_rewards = to_u128(pool.reward_per_sec) * to_u128(
 ↳ seconds_passed) * to_u128(pool.stake_scale);
538     total_rewards / to_u128(total_stake)
539 }
```

Proof of Concept:

The scenario used for this proof of concept consists on: 100 reward coins, 100 seconds pool duration (1 reward coin per second) and 3 users. No NFT Boost applied.

After 10 seconds from the beginning, the first user deposits 5APT coins, 15 seconds later the second user deposits 30APT coins and 45 seconds later the third user deposits 20APT coins.

From this moment, the accumulated rewards is 4, but this will only apply to first and second users because the unobtainable amount for third user overrides the reward. The reason is that the first and second users made the deposit with space enough to accumulate some rewards, however, when the third user made the deposit, the remaining time to the end of the pool duration is smaller than the division of (TotalStake+TotalBoosted)/RewardsPerSecond, therefore, the last user will not receive any reward.

Listing 9: PoC_script

```
amount='100'
duration='100'
pool_address=$POOL_ADDRESS
#Init Stake Pool
aptos move run --profile dgen_coin --function-id 'harvest::scripts
↳ ::register_pool' --args 'u64:'$amount --args 'u64:'$duration --
↳ type-args $typeS --type-args $typeR --assume-yes
#=========================
echo 'Wait 10 second before the first stake'
sleep 10

echo 'User1 stakes 5 APT'
amount='500000000'
aptos move run --profile user --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

echo '15 seconds pass for a total stake of 5 APT'
sleep 15

echo 'User2 stakes 30 APT'
amount='3000000000'
aptos move run --profile user2 --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

echo '45 seconds passed for a total stake of 35APT'
sleep 45

echo 'User3 stakes 20 APT'
amount='2000000000'
aptos move run --profile user3 --function-id 'harvest::scripts::
↳ stake' --args 'address:'$pool_address --args 'u64:'$amount --type-
↳ args $typeS --type-args $typeR --assume-yes

#====== Harvest time
```

```
echo 'Wait some seconds'
sleep 20

echo 'User1 try to harvest rewards --> HARVEST 20 reward coins'
aptos move run --profile user --function-id 'harvest::scripts::
↳ harvest' --args 'address:'$pool_address --type-args $typeS --type-
↳ args $typeR --assume-yes

sleep 5

echo 'User2 try to harvest rewards --> HARVEST 30 reward coins'
aptos move run --profile user2 --function-id 'harvest::scripts::
↳ harvest' --args 'address:'$pool_address --type-args $typeS --type-
↳ args $typeR --assume-yes

sleep 5

echo 'User3 try to harvest rewards --> NOTHING TO HARVEST'
aptos move run --profile user3 --function-id 'harvest::scripts::
↳ harvest' --args 'address:'$pool_address --type-args $typeS --type-
↳ args $typeR --assume-yes

```

Risk Level:

Likelihood - 2
Impact - 3

Recommendation:

It is recommended to check the status of the duration of the pool and the total stake before allowing more stakes, canceling those that will not receive any reward.

Remediation plan:

RISK ACCEPTED: The Pontem Network team has implemented a fix that partially solves the situation, decreasing the risk from **Medium** to **Low**.

The code was modified to include a scaling factor to add more precision to the accumulated_rewards calculation, avoiding some situations where the value would freeze due to truncated integers.
However, there are still some edge cases where the issue persists.

In this case, to get to a scenario where the user does not receive any rewards, the TotalStake saved in the pool should be greater than RewardsPerSec * SecondsLeftToEndPool * RewardScale. If the minimum value of RewardsPerSec = 1 is considered, the minimum value of RewardScale = 100 and a logical scenario of 1 day remaining to the end of the pool (86400 seconds), that implies a TotalStake greater than 8 millions.

It looks like a huge amount, but depending on the token it might not be an impossible amount to reach, for that reason, the risk has been reduced to **Low**, but not completely eliminated.

3.4 (HAL-04) PRIVILEGED ADDRESS TRANSFERRED WITHOUT CONFIRMATION - LOW

Description:

The initialization of the stake_config module allows declaring two main addresses for the staking configuration: emergency_admin_address responsible to enable the global emergency status for the complete pools market, and the treasury_admin, in charge of collect the non-requested rewards after 3 months or withdraw all the rewards in case of emergency.

These addresses could be changed at any time by the current emergency_admin_address and treasury_admin accounts using the set_emergency_admin_address and set_treasury_admin_address functions. However, an incorrect use of them could set the addresses to an invalid one, unwillingly losing control of the staking configuration, which cannot be undone in any way.

Losing the ability to enable global emergency status or being unable to change the treasury address in case of compromise are some of the consequences of an incorrect address transfer.

It should be noted that the stake_emergency_admin account is managed by a multisig wallet (3 participants, 2 signatures needed) which decreases the likelihood of the finding.

Code Location:

The set_emergency_admin_address function from **stake_config** module:

Listing 10: sources/stake_config.move (Lines 53,56)

```
public entry fun set_emergency_admin_address(emergency_admin: &
↳ signer, new_address: address) acquires GlobalConfig {
    assert!(exists<GlobalConfig>(@stake_emergency_admin),
↳ ERR_NOT_INITIALIZED);
```

```
    let global_config = borrow_global_mut<GlobalConfig>(
↳ @stake_emergency_admin);
    assert!(
        signer::address_of(emergency_admin) == global_config.
↳ emergency_admin_address,
        ERR_NO_PERMISSIONS
    );
    global_config.emergency_admin_address = new_address;
}
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

The set_emergency_admin_address function should follow a two steps process, being split into set_emergency_admin_address and accept_emergency_admin_address functions. The latter one requiring the transfer to be completed by the recipient, effectively protecting the contract against potential typing errors compared to single-step address transfer mechanisms. The same process should be applied to set_treasury_admin_address function.

Remediation plan:

RISK ACCEPTED: The Pontem Network team accepted the risk of this finding.

3.5 (HAL-05) EMERGENCY STATE CANNOT BE REVOKED - LOW

Description:

The enable_global_emergency and enable_emergency functions, from **stake_-config** and **stake** modules respectively, allow enabling the emergency mode at global level (all the pools) or just in a specific pool address. This emergency state disables all the operations on the StakePool instances, except for the emergency_unstake function from **stake** module.

The emergency state could not be deactivated once it is enabled, this means that any unwilling error or false alarm would have important impact on the availability of the harvesting service.

Code Location:

enable_global_emergency function from **stake_config** module:

Listing 11: sources/stake_config.move (Line 100)

```
 92 public entry fun enable_global_emergency(emergency_admin: &signer)
  ↳  acquires GlobalConfig {
 93     assert!(exists<GlobalConfig>(@stake_emergency_admin),
  ↳ ERR_NOT_INITIALIZED);
 94     let global_config = borrow_global_mut<GlobalConfig>(
  ↳ @stake_emergency_admin);
 95     assert!(
 96         signer::address_of(emergency_admin) == global_config.
  ↳ emergency_admin_address,
 97         ERR_NO_PERMISSIONS
 98     );
 99     assert!(!global_config.global_emergency_locked,
  ↳ ERR_GLOBAL_EMERGENCY);
100     global_config.global_emergency_locked = true;
101 }
```

enable_emergency function from stake module:

Listing 12: sources/stake.move (Line 344)

```
334 public fun enable_emergency<S, R>(admin: &signer, pool_addr:
  ↳ address) acquires StakePool {
335     assert!(exists<StakePool<S, R>>(pool_addr), ERR_NO_POOL);
336     assert!(
337         signer::address_of(admin) == stake_config::
  ↳ get_emergency_admin_address(),
338         ERR_NOT_ENOUGH_PERMISSIONS_FOR_EMERGENCY
339     );
340
341     let pool = borrow_global_mut<StakePool<S, R>>(pool_addr);
342     assert!(!is_emergency_inner(pool), ERR_EMERGENCY);
343
344     pool.emergency_locked = true;
345 }
```

Risk Level:

Likelihood - 1
Impact - 4

Recommendation:

It is recommended to have an alternative to disable the emergency state in case of an unwilling error or false alarm, at least for the global emergency, due to the high-impact on the availability of the service.

Remediation plan:

RISK ACCEPTED: The Pontem Network team accepted the risk of this finding.

3.6 (HAL-06) HARVESTED REWARDS ARE LOST ON EMERGENCY - LOW

Description:

The emergency_unstake function from **stake** module allows any user to withdraw their stake deposited into an emergency-stopped Stake Pool. The function only could be executed once because all the amount deposited will be withdrawn.

In case of emergency, emergency-unstake is the only operation allowed for users, and it does not include any reward harvesting; therefore, the users will lose any rewards accumulated since the last call to harvest function before the emergency state began.

The harvested rewards cannot be withdrawn by each user, and only the stake could be recovered. However, these rewards could be harvested by the treasury account, even when it was not the account which deposited them into the pool.

Code Location:

The emergency_unstake function from **stake** module:

Listing 13: sources/stake.move (Lines 355,363)

```
351 public fun emergency_unstake<S, R>(user: &signer, pool_addr:
  ↳ address): Coin<S> acquires StakePool {
352     assert!(exists<StakePool<S, R>>(pool_addr), ERR_NO_POOL);
353
354     let pool = borrow_global_mut<StakePool<S, R>>(pool_addr);
355     assert!(is_emergency_inner(pool), ERR_NO_EMERGENCY);
356
357     let user_addr = signer::address_of(user);
358     assert!(table::contains(&pool.stakes, user_addr), ERR_NO_STAKE
  ↳ );
359
360     let user_stake = table::remove(&mut pool.stakes, user_addr);
```

```
    let UserStake { amount, unobtainable_reward: _, earned_reward:
↳ _, unlock_time: _ } = user_stake;

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

Risk Level:

Likelihood - 1
Impact - 3

Recommendation:

The emergency state is expected to be activated when the mathematical operations stop working due to any unexpected behaviour, therefore, since it is not possible to execute any last harvest operation to withdraw the rewards pending to be claimed, it is recommended to modify the emergency_unstake function by adding to the stake amount the harvested rewards reflected into the earned_reward record of the UserStake parameter, even if it was not recently updated.

Remediation plan:

RISK ACCEPTED: The Pontem Network team accepted the risk of this finding.

3.7 (HAL-07) NO OPTION TO DESTROY AN EMPTY STAKE POOL - LOW

Description:

The Stake module allows any user to create a StakePool resource, however, once all the pool rewards have been shared, the duration time has ended and the users have unstake their funds, the resource does not have any other utility because it cannot stake more coins neither add more rewards.

If the same user wants to create again a StakePool with same pair of coins in order to add more rewards, or to add an NFT Collection for Boost capabilities, it won't be possible because of the checking of Stake Pool resource existence.

Code Location:

Fragment of register_pool function from **stake** module, where the previous existence of the Stake Pool is validated:

Listing 14: sources/stake.move (Line 125)

```
124 public fun register_pool<S, R>(owner: &signer, reward_coins: Coin<R>, duration: u64) {
125     assert!(!exists<StakePool<S, R>>(signer::address_of(owner)), ERR_POOL_ALREADY_EXISTS);
126     assert!(coin::is_coin_initialized<S>() && coin::is_coin_initialized<R>(), ERR_IS_NOT_COIN);
127     assert!(!stake_config::is_global_emergency(), ERR_EMERGENCY);
128     assert!(duration > 0, ERR_DURATION_CANNOT_BE_ZERO);
129
130     let reward_per_sec = coin::value(&reward_coins) / duration;
131     assert!(reward_per_sec > 0, ERR_REWARD_CANNOT_BE_ZERO);
132
133     let current_time = timestamp::now_seconds();
134     let end_timestamp = current_time + duration;
135
136 }
```

Risk Level:

Likelihood - 2
Impact - 2

Recommendation:

It is recommended to add a mechanism to destroy the Stake Pool resource once all the creation conditions has been accomplished: all the funds unstaked, all the rewards shared or withdrawn to treasury, harvest time finished and so on.

Remediation plan:

RISK ACCEPTED: The Pontem Network team accepted the risk of this finding.

3.8 (HAL-08) NFT BOOST CAN BE PERFORMED AFTER THE STAKE POOL ENDS - INFORMATIONAL

Description:

The boost function from **stake** module allows any user to deposit an NFT into their UserStake resource in order to increase (or boost) the amount of rewards earned. This action could be done at any time, even when the Stake Pool duration has ended.

In that case, the boost will have no effect on the reward harvested, being useless at that point, and the user will need to spend another amount of gas to remove the boost and recover the NFT.

Code Location:

Fragment of boost function from **stake** module where should be added an extra check for Pool duration:

Listing 15: sources/stake.move

```
    public fun boost<S, R>(user: &signer, pool_addr: address, nft:
  ↳ Token) 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);
        assert!(option::is_some(&pool.nft_boost_config),
  ↳ ERR_NON_BOOST_POOL);

        let user_address = signer::address_of(user);
        assert!(table::contains(&pool.stakes, user_address),
  ↳ ERR_NO_STAKE);

        let token_amount = token::get_token_amount(&nft);
        assert!(token_amount == 1, ERR_NFT_AMOUNT_MORE_THAN_ONE);

        let token_id = token::get_token_id(&nft);
        let (token_collection_owner, token_collection_name, _, _)
 ↳ = token::get_token_id_fields(&token_id);

        let params = option::borrow(&pool.nft_boost_config);
        let boost_percent = params.boost_percent;
        let collection_owner = params.collection_owner;
        let collection_name = params.collection_name;

        // check nft is from correct collection
        assert!(token_collection_owner == collection_owner,
 ↳ ERR_WRONG_TOKEN_COLLECTION);
        assert!(token_collection_name == collection_name,
 ↳ ERR_WRONG_TOKEN_COLLECTION);

        // recalculate pool
        update_accum_reward(pool);
```

Risk Level:

Likelihood - 1
Impact - 1

Recommendation:

It is recommended to add an extra check for Stake Pool duration in the boost function, similar to the stake one, in order to avoid useless operation and the consequent waste of gas.

Remediation plan:

ACKNOWLEDGED: The Pontem Network team acknowledged this finding.

THANK YOU FOR CHOOSING

// HALBORN