Hubble Farms

Audit

Presented by:

OtterSec contact@osec.io

Akash Gurugunti sud0u53r.ak@osec.io

Ajay Kunapareddy d1r3wolf@osec.io

Robert Chen r@osec.io

Contents

Appendices

01 | Executive Summary

Overview

Hubble Protocol engaged OtterSec to perform an assessment of the `farms` program. This assessment was conducted between October 4th and October 13th, 2023. For more information on our auditing methodology, see Appendix B.

Key Findings

Throughout this audit engagement, we produced 10 findings in total.

In particular, we identified multiple high-risk vulnerabilities, including one concerning the potential modification of the time unit in the farm configuration midway, resulting in incorrect values for accrued rewards and reward types (OS-FRM-ADV-00).

We discovered another issue pertaining to the alteration of reward parameters without properly updating global rewards, resulting in inaccurate reward distributions (OS-FRM-ADV-01). We also advised against permitting withdrawals that reduce either the total active amount or total pending amount to zero, as this may be exploitable and disrupt functions relying on these values (OS-FRM-ADV-03).

We also recommended implementing a two-step process when updating the owner for specific structures to minimize the risk of inadvertent change in ownership (OS-FRM-SUG-00) and proposed the removal of unused code (OS-FRM-SUG-01). Additionally, we suggested the inclusion of missing checks (OS-FRM-SUG-04) along with code adjustments and optimizations to improve readability and efficiency (OS-FRM-SUG-03, OS-FRM-SUG-05).

02 | Scope

The source code was delivered to us in a git repository at github.com/hubbleprotocol/farms. This audit was performed against commit 5fcec0d.

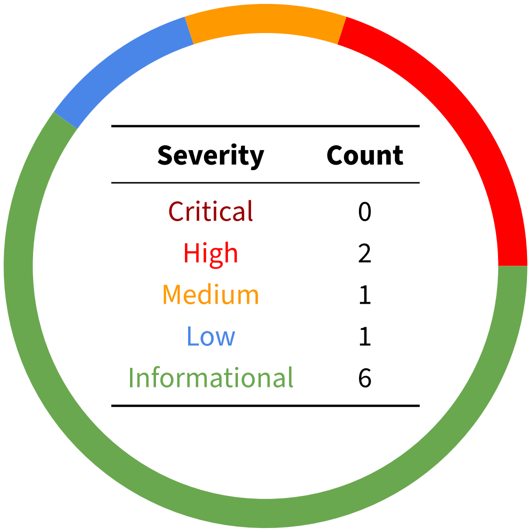
A brief description of the programs is as follows:

Name	Description
farms	A versatile farm/staking pool featuring permissionless farms, each with a distinct global administrator. Users have the flexibility to stake or unstake at any time, and they may harvest their accrued rewards individually.

03 | Findings

Overall, we reported 10 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	2
Medium	1
Low	1
Informational	6

04 | Vulnerabilities

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

Rating criteria can be found in Appendix A.

ID	Severity	Status	Description
OS-FRM-ADV-00	High	Resolved	Updating `time_unit` in `UpdateFarmConfig` results in incorrect accrued rewards and reward type values.
OS-FRM-ADV-01	High	Resolved	Modifying reward parameters without properly updating global rewards results in inaccurate reward distributions.
OS-FRM-ADV-02	Medium	Resolved	`update_user_rewards_tally_on_stake_increase` is invoked for pending stakes, forcing early inclusion of these stakes in the rewards tally.
OS-FRM-ADV-03	Low	Resolved	`convert_stake_to_amount` incorrectly adds a user's stake to the total amount, miscalculating the conversion of stake shares to token amounts.

OS-FRM-ADV-00 [high]| Modification Of Time Unit

Description

update_farm_config updates the configuration of a farming pool, allowing pool administrators to modify various parameters and settings within the farm to adapt to changing market conditions. Particularly, UpdateFarmConfig instruction modifies time_unit, which represents the time interval over which the distribution of rewards occurs and how frequently users can claim their rewards.

farm_operations.rs RUST

```rust
pub fn update_farm_config(
    farm_state: &mut FarmState,
    mode: FarmConfigOption,
    data: &[u8; 32],
) -> Result<()> {
    match mode {
        [...]
        FarmConfigOption::UpdateFarmTimeUnit => {
            let value: u8 = BorshDeserialize::deserialize(&mut &data[..])?;
            [...]
            farm_state.time_unit = value;
        }
        [...]
    };
    Ok(())
}
```

The issue arises as time_unit correlates with multiple crucial time stamps such as last_issuance_ts in reward_infos, representing the timestamp of the last issuance of rewards to users, and last_claim_ts in user_state, responsible for recording the time when a user last claimed their rewards.

Therefore, updating time_unit will modify these time stamps. Users who have previously staked their assets and claimed rewards will have done so using the earlier time_unit, and a sudden change in this value may result in inconsistencies, as the historical data regarding when users staked and when rewards were issued is no longer valid.

Moreover, this change will inconvenience users who have become used to claiming rewards based on the previous time duration. It may result in generating inaccurate reward calculations, discrepancies in when users may claim rewards, and confusion regarding their expected rewards and claim timing.

Proof of Concept

1. The contract uses `time_unit` of seconds, with users earning rewards based on this unit.
2. An administrator decides to change `time_unit` to minutes, and the contract updates accordingly.
3. The change in `time_unit` renders `last_issuance_ts` in `reward_infos` incompatible with the new `time_unit`.
4. Users' `last_claim_ts` in `user_state` is tied to the old `time_unit`, potentially resulting in timing and calculation issues when claiming rewards.

The changes described above may cause user reward discrepancies and inconsistent calculations.

Remediation

Ensure the `time_unit` is not modified mid-course.

Patch

Fixed in 6ac662e by removing option to change time unit.

OS-FRM-ADV-01 [high] | Reward Parameter Modifications

Description

`update_reward_config` allows for the direct modification of parameters like `reward_per_share`, `reward_type`, and `rewards_per_second_decimals` within the `RewardInfo` structure. These parameters are essential for calculating and distributing rewards to users.

farm_operations.rs RUST

```rust
pub(crate) fn update_reward_config(
    reward_info: &mut RewardInfo,
    mode: FarmConfigOption,
    value: u64,
    ts: u64,
) {
    [...]
    match mode {
        FarmConfigOption::UpdateRewardRps => {
            reward_info.reward_per_second = value;
            reward_info.last_issuance_ts = ts;
        }
        FarmConfigOption::RewardType => {
            let value: u8 = value.try_into().unwrap();
            xmsg!(
                "farm_operations::update_farm_config reward_type={value}
                    ↪  type={:?}",
                RewardType::try_from_primitive(value).unwrap()
            );
            reward_info.reward_type = value;
        }
        FarmConfigOption::RpsDecimals => {
            let value: u8 = value.try_into().unwrap();
            xmsg!("farm_operations::update_farm_config rps_decimals={value}",);
            reward_info.rewards_per_second_decimals = value;
        }
        [...]
    }
}
```

The vulnerability stems from the function not refreshing or updating the global reward state when these parameters are modified. Consequently, these new values are applied on already elapsed time, leading to inconsistent values for accrued rewards and reward type, resulting in inaccurate reward distribution.

Specifically, when `reward_per_share` is updated, and the `last_issuance_ts` is set to the current timestamp, the system may neglect the rewards accrued from the previous configuration change to the current timestamp.

Proof of Concept

1. Initially, the reward parameters are as follows:
   - `reward_per_share`: 0.1 tokens.
   - `last_issuance_ts`: 1000 (in seconds).
2. An update occurs utilizing `update_reward_config`:
   - `mode`: `UpdateRewardRps`.
   - `value`: 0.15 tokens per share.
   - `ts`: Current timestamp (1500 seconds).
3. In this update, `reward_per_share` is modified from 0.1 to 0.15 tokens per share, and `last_issuance_ts` is updated to 1500 seconds.
4. The issue arises because the function does not consider the time between the previous update (at timestamp 1000) and the current update (at timestamp 1500). Users earned rewards based on the old `reward_per_share` during this period.
5. The new `reward_per_share` setting is applied to all users immediately, and any rewards earned between timestamps 1000 and 1500 were calculated based on the old rate.
6. As a result, users' reward balances are not accurately updated to reflect what they should have earned with the new parameters. This may result in inconsistent and incorrect reward distributions.

Remediation

Ensure these parameters are not directly altered without refreshing the global rewards state.

Patch

Fixed in 0064975 by refreshing global rewards before changing reward parameters on reward config.

OS-FRM-ADV-02 [med]| Incorrect Addition Of Staked Amounts

Description

`stake` is essential for users to enter the farming protocol and begin staking tokens. If there is no pending deposit period (i.e., `deposit_warmup_period` is zero), the function directly adds the staked tokens to the user's active stake. However, if there is a pending deposit period, users experience a waiting period before their staked tokens become active and start earning rewards.
`update_user_rewards_tally_on_stake_increase` in `stake` adds a user's earned rewards to their rewards tally for each reward token.

farm_operations.rs RUST

```rust
pub fn stake(
    farm_state: &mut FarmState,
    user_state: &mut UserState,
    amount: u64,
    current_ts: u64,
) -> Result<StakeEffects> {
    [...]
    let stake_gained = if farm_state.deposit_warmup_period > 0 {
        // If there is a pending stake period, we add the stake to the pending
            ↪  stake
        user_state.pending_deposit_stake_ts = current_ts
            .checked_add(farm_state.deposit_warmup_period.into())
            .ok_or_else(|| dbg_msg!(FarmError::IntegerOverflow))?;
        let stake_gained = stake_ops::add_pending_deposit_stake(user_state,
            ↪  farm_state, amount)?;
        [...]
        stake_gained
    }
    [...]
    update_user_rewards_tally_on_stake_increase(farm_state, user_state,
        ↪  stake_gained)?;
}
```

The issue arises because `update_user_rewards_tally_on_stake_increase` is called whenever there is an increase in the user's stake, whether it is an active stake or a pending stake that is still in the warm-up period. Thus, `stake` prematurely adds pending stakes to the rewards tally by calling `update_user_rewards_tally_on_stake_increase` for increases in both active and pending stakes.

This results in pending stakes being added to the rewards tally before they have officially become active and start earning rewards, which is problematic as `update_user_rewards_tally_on_stake_increase` is called again when the pending stakes turn active, effectively doubling the users rewards tally for the same token.

Remediation

Limit the utilization of `update_user_rewards_tally_on_stake_increase` solely to actively staked amounts.

Patch

Fixed in 9949c09 by using `update_user_rewards_tally_on_stake_increase` on only actively staked amounts.

OS-FRM-ADV-03 [low] | Incorrect Removal Of Pending Deposit Stake

Description

In stake_operations, convert_stake_to_amount converts a stake (represented as a decimal) into an equivalent amount of tokens (represented as a u64) based on the total stake and total amount in a farm. It ensures that token distribution is proportional and follows the specified rounding rules.

stake_operations.rs RUST

```rust
pub fn convert_stake_to_amount(
    stake: Decimal,
    total_stake: Decimal,
    total_amount: u64,
    round_up: bool,
) -> u64 {
    [...]
    let amount_dec = if total_stake != Decimal::zero() {
        stake * total_amount / total_stake
    } else {
        stake + total_amount.into()
    };
    [...]
}

pub fn remove_pending_withdrawal_stake(
    user_stake: &mut impl UserStakeAccessor,
    farm: &mut impl FarmStakeAccessor,
) -> Result<u64, FarmError> {
    [...]
    // Round down to favor the farm.
    let pending_amount_removed: u64 = convert_stake_to_amount(
        user_stake.pending_withdrawal_unstake,
        farm.total_pending_stake,
        farm.total_pending_amount,
        false,
    );

    farm.total_pending_amount -= pending_amount_removed;
    farm.total_pending_stake = farm.total_pending_stake -
        ↪ user_stake.pending_withdrawal_unstake;
    [...]
}
```

remove_pending_deposit_stake calls convert_stake_to_amount internally which returns pending_amount_removed. The program removes this value from the pending deposit stake of a user. The problem arises in the else branch of convert_stake_to_amount, where it attempts to calculate the amount of tokens based on the sum of stake and total_amount, instead of total_amount alone.

When this incorrect value is deducted in `remove_pending_deposit_stake`, it may result in the deduction of more tokens than the user's pending deposit stake actually represents. In this instance, the calculation may generate a negative balance of `total_pending_amount`, triggering an error.

Remediation

Modify `convert_stake_to_amount` to calculate the amount of tokens based on `total_amount alone`, excluding the `stake`.

Patch

Fixed in d7411aa.

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-FRM-SUG-00	Utilize a two-step verification process to confirm a change in ownership.
OS-FRM-SUG-01	Remove unutilized functions, structure fields, accounts, and instructions.
OS-FRM-SUG-02	Lack of proper error handling in the `RefreshUserState` instruction and `farm_operations::update_farm_config`.
OS-FRM-SUG-03	Suggestions regarding best practices and improved code readability.
OS-FRM-SUG-04	Missing checks affect code readability and may result in security issues.
OS-FRM-SUG-05	Recommendations regarding optimizing the code to increase efficiency.

OS-FRM-SUG-00 | Double Verification For Owner Change

Description

The `GlobalConfig` structure represents the global configuration settings, and the `UserState` structure represents the state of a user's interaction, containing information about the user's stake, rewards, etc. Currently, the owner change process for both `GlobalConfig` and `UserState` is a single-step process; there is no confirmation step. Once the transaction is submitted, the owner change is irreversible. This may result in a denial of service when the current owner accidentally sends an unintended input as a parameter while executing an owner change.

Remediation

Utilize a two-step process to change the owner of the lending market.

OS-FRM-SUG-01 | Unused Code

Description

The following functions, structure fields, accounts, or instructions remain unused and should be removed:

1. `gen_signer_seeds` in `macros`.
2. `program_id` in `transfer_from_vault`.
3. `reward_fee_rate_bps` in the `RewardInfo` structure.
4. The `system_program` account in the `UpdateGlobalConfig` instruction.

Remediation

Remove the above-listed items, upholding best coding practices and improving readability and maintainability.

OS-FRM-SUG-02 | Error Handling

Description

1. The RefreshUserState instruction refreshes a user's state within the farm. Part of this refresh operation involves calling user_refresh_stake, responsible for activating a user's pending stake. However, this operation may not be suitable for delegated farms, which have different mechanisms for handling stakes and rewards. Utilizing the same refresh mechanism for delegated farms may result in issues or unintended consequences.
2. In farm_operations in update_farm_config, the program should throw an error if it is a delegated farm and the farm admin is trying to update the deposit warmup period or withdraw cooldown period and withdraw_authority, since these values assert to be zero in set_stake.

Remediation

1. The RefreshUserState instruction should throw an error if the farm is delegated.
2. update_farm_config should throw an error in the above-described scenario.

OS-FRM-SUG-03 | Code Maturity

Description

1. Renaming `reward_per_second` to `reward_per_unit_time` in the `RewardInfo` structure may render the code more accurate in terms of naming and documentation, especially when this field is utilized for both seconds and slots.

   state.rs RUST

   ```rust
   #[zero_copy]
   #[derive(AnchorSerialize, AnchorDeserialize, Debug, Default, PartialEq, Eq)]
   pub struct RewardInfo {
       pub token: TokenInfo,
       [...]
       pub reward_per_second: u64,
       [...]
   }
   ```

2. The seed strings for `BASE_SEED_FARM_VAULTS_AUTHORITY` and `BASE_SEED_TREASURY_VAULTS_AUTHORITY` are identical. It is recommended to modify one of them for the purpose of distinction.

3. In `ten_pow`, the error message erroneously mentions the upper range as ten instead of 19.

   math.rs RUST

   ```rust
   pub fn ten_pow(x: usize) -> u64 {
       [...]
       // Validate that x is in the range [0, 19]
       if x > 19 {
           panic!("The exponent must be between 0 and 10.");
       }
       [...]
   }
   ```

4. `refresh_global_rewards` utilizes `total_active_stake_scaled` instead of `total_staked_amount` for the early return.

Remediation

Implement the modifications mentioned above into the code base.

OS-FRM-SUG-04 | Missing Checks

Description

1. Add a check to restrict the maximum value to ensure that the `treasury_fee_bps` value does not exceed a maximum allowed value (`MAX_BPS`) when updating the global configuration.
2. Modify `update_farm_config` to utilize `reward_info.is_initialised()` instead of checking if `reward_info.token.mint` is not equal to `Pubkey::default()`. This makes the code more readable and potentially reduces the risk of errors.
3. Adding an authority check over the `depositor_ata` token account is essential to ensure the depositor authorizes the deposit operation. This check should verify that the depositor owns the `depositor_ata` account.

Remediation

1. Add the following check: `value <= MAX_BPS` for `treasury_fee_bps` updation.
2. Utilize `reward_info.is_initialised()` instead of `reward_info.token.mint != Pubkey::default()`
3. Ensure to check the authority of the `depositor_ata` token account.

OS-FRM-SUG-05 | Code Optimizations

Description

1. In `farm_operations` in `user_refresh_all_rewards`, implement an early return if `user_state.active_stake_scaled` equals zero, which prevents unnecessary code execution.
2. For the `GlobalConfig`, `FarmConfig`, `UserState` structures, utilizing u64 for bump is excessive as u8 is sufficient for bump fields, especially when the expected range of values is small. Similarly, in `Tokeninfo`, `decimals` may be of u8 type.

state.rs RUST

```rust
pub struct TokenInfo {
    [...]
    pub decimals: u64,
}
```

3. In `Stake` and the `DepositToFarmVault` instruction accounts constraints, `load_mut` is used, which is inappropriate in this context as it modifies the account state. However, in this instance, no writing occurs; it should be replaced with `load`, which loads the account's state for reading.

handler_deposit_to_farm_vault.rs RUST

```rust
pub struct DepositToFarmVault<'info> {
    [...]
    [account(mut,
        [...]
        constraint = farm_vault.mint == farm_state.load_mut()?.token.mint @
            ↪  FarmError::TokenFarmTokenMintMissmatch,
    )]
    [...]
}
```

4. In `farm_operations` in `initialize_user`, `farm_state_key` may be derived from `farm_state`, eliminating the requirement of passing it as a separate function argument.

farm_operations.rs RUST

```rust
pub fn initialize_user(
    farm_state: &mut FarmState,
    user_state: &mut UserState,
    owner_key: &Pubkey,
    farm_state_key: &Pubkey,
    ts: u64,
) -> Result<()> {}
```

Remediation

Implement the optimizations listed above.

A | Vulnerability Rating Scale

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

Critical

Vulnerabilities that immediately result in a loss of user funds with minimal preconditions.

Examples:

- Misconfigured authority or access control validation.
- Improperly designed economic incentives leading to loss of funds.

High

Vulnerabilities that may result in a loss of user funds but are potentially difficult to exploit.

Examples:

- Loss of funds requiring specific victim interactions.
- Exploitation involving high capital requirement with respect to payout.

Medium

Vulnerabilities that may result in denial of service scenarios or degraded usability.

Examples:

- Computational limit exhaustion through malicious input.
- Forced exceptions in the normal user flow.

Low

Low probability vulnerabilities, which are still exploitable but require extenuating circumstances or undue risk.

Examples:

- Oracle manipulation with large capital requirements and multiple transactions.

Informational

Best practices to mitigate future security risks. These are classified as general findings.

Examples:

- Explicit assertion of critical internal invariants.
- Improved input validation.

B | Procedure

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

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

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

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

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

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

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