Pashov Audit Group

Meteora Security Review

September 1st 2025 - September 3rd 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource and expertise bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs and on-chain monitoring are strongly recommended.

3. Risk Classification

Severity	Impact: High	Impact: Medium	Impact: Low
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Meteora

Meteora is a system for managing the distribution of position NFTs. Vaults are initialized with parameters like creation fees and capacity limits, allowing users to register by paying a fee, while operators are authorized to manage operations.

5. Executive Summary

A time-boxed security review of the **MeteoraAg/position-nft-distribution** repository was done by Pashov Audit Group, during which **Newspace**, **LordAlive**, **FrankCastle** engaged to review **Meteora**. A total of **6** issues were uncovered.

Protocol Summary	
Project Name	Meteora
Protocol Type	Position NFTs Distribution
Timeline	September 1st 2025 - September 3rd 2025

Review commit hash:

- 58a3b7e8267f34ab8c26e22e732b2c2d819ed039 (MeteoraAg/position-nft-distribution)

Fixes review commit hash:

- 4192ee14c085124eed539b602939f21716761ce6 (MeteoraAg/position-nft-distribution)

Scope

`macros.rs` `lib.rs` `errors.rs` `constants.rs` `auth.rs` `operator.rs` `mod.rs` `record_tx.rs` `register.rs` `vault.rs` `safe_math.rs` `ix_initialize_operator.rs` `ix_initialize_vault.rs` `ix_record_tx.rs` `ix_register.rs` `ix_remove_operator.rs` `ix_remove_registered_user.rs` `ix_update_vault.rs` `ix_withdraw_from_vault.rs`

6. Findings

Findings count

Severity	Amount
Medium	1
Low	5
Total findings	**6**

Summary of findings

ID	Title	Severity	Status
[M-01]	Multiple vaults share the same authority, breaking withdrawal timeline invariants	Medium	**Resolved**
[L-01]	Hardcoded admin addresses	Low	Acknowledged
[L-02]	Missing validation when updating vault parameters allows inconsistent state	Low	**Resolved**
[L-03]	Single operator can register users across all vaults, risking centralization	Low	Acknowledged
[L-04]	`handle_register` can exceed `max_cap`	Low	Acknowledged
[L-05]	User removal does not refund fees, causing user fund loss	Low	Acknowledged

Medium findings

[M-01] Multiple vaults share the same authority, breaking withdrawal timeline invariants

Severity

Impact: Medium

Likelihood: Medium

Description

Currently, multiple vaults are created, but all of them send their lamports to the **same** `vault_authority` **account**. When withdrawing, the program asserts that a timeline (end time) has passed. However, since all vault balances are aggregated into the same `vault_authority`, withdrawals transfer **all lamports** (including those belonging to incomplete vaults).

This behavior **breaks the timeline invariant**, as it allows lamports from vaults that have not yet reached their end time to be withdrawn prematurely.

Example:

- Vault A and Vault B exist, both sending funds to the same `vault_authority`.
- Vault A's end time is reached, and a withdrawal is triggered.
- Since funds are shared under the same authority, **Vault B's lamports** (not yet eligible for withdrawal) are also transferred, violating time-based restrictions.

Relevant code

Vault initialization:

```
#[account(
    init,
    payer = payer,
    space = 8 + Vault::INIT_SPACE
)]
pub vault: AccountLoader<'info, Vault>,
```

Shared vault authority:

```
#[account(
    mut,
    seeds = [
        VAULT_AUTHORITY_PREFIX.as_ref(),
    ],
```

```
    bump,
)]
pub vault_authority: UncheckedAccount<'info>,
```

Funds transferred to shared authority:

```
transfer(
    CpiContext::new(
        ctx.accounts.system_program.to_account_info(),
        Transfer {
            from: ctx.accounts.payer.to_account_info(),
            to: ctx.accounts.vault_authority.to_account_info(),
        },
    ),
    vault.creation_fee,
)?;
Ok(())
```

Recommendations

There are two possible solutions:

Derive a unique vault authority per vault.

- Each vault account should have its own PDA authority.
- Validate during withdrawal that the correct authority is used.
- This ensures withdrawals only affect the intended vault.

Modify the withdrawal logic.

- Instead of transferring all lamports from the shared authority, only transfer the balance **associated with the specific vault.**
- This ensures incomplete vaults remain unaffected until their timelines are reached.

Low findings

[L-01] Hardcoded admin addresses

Admin addresses are hardcoded as constants, and they can't be updated after deployment. When the admin key is compromised, it can't be updated.

Implement an upgradeable admin configuration or multi-signature system to allow admin rotation and improve operational flexibility.

[L-02] Missing validation when updating vault parameters allows inconsistent state

The `handle_update_vault` function allows updates to critical vault parameters such as `end_time`, `max_cap`, and `max_user`. However, the current implementation does not perform sufficient validation, which can result in inconsistent or invalid vault states.

Specifically:

End time validation is missing. * The vault's `end_time` can be set to a past timestamp, which breaks the expected timeline invariant.

Max cap validation is missing. * `max_cap` can be set lower than the current vault balance, potentially making the vault state impossible to satisfy.

Max user validation is missing. * `max_user` can be set lower than the current number of users, creating an inconsistency that blocks new user registrations while existing ones exceed the limit.

Relevant code

```
pub fn handle_update_vault(
    ctx: Context<UpdateVaultCtx>,
    params: &UpdateVaultParameters,
) -> Result<()> {
    // validate params
    params.validate()?;

    let mut vault = ctx.accounts.vault.load_mut()?;
    if let Some(creation_fee) = params.creation_fee {
        vault.creation_fee = creation_fee;
    }
    if let Some(end_time) = params.end_time {
        vault.end_time = end_time;
    }

    if let Some(max_cap) = params.max_cap {
        vault.max_cap = max_cap;
    }
```

```
    if let Some(max_user) = params.max_user {
        vault.max_user = max_user;
    }

    Ok(())
}
```

Recommendations

Add validation checks to enforce consistent and safe updates:

1. **End time must be in the future.**

   `rust require!(end_time > Clock::get()?.unix_timestamp, ErrorCode::InvalidEndTime);`

1. **Max cap must be ≥ current vault balance.**
2. **Max user must be ≥ current number of users.**

By enforcing these rules, the vault will remain in a valid state, preventing edge cases where withdrawals or user registrations become inconsistent with the vault's configuration.

[L-03] Single operator can register users across all vaults, risking centralization

The current implementation allows a **single operator** to register users on **all vaults** created by the program. This design introduces a **centralization risk**, as one compromised or malicious operator could control registration across the entire system.

The issue arises because the operator account is derived only from the `OPERATOR_PREFIX` and the operator's public key. It does not take the **vault address** into account, meaning the same operator PDA applies globally to all vaults.

Relevant code

Operator initialization:

```
#[account(
    init,
    payer = admin,
    seeds = [
        OPERATOR_PREFIX.as_ref(),
        operator.key().as_ref(),
    ],
    bump,
    space = 8 + RegisterOperator::INIT_SPACE
)]
pub register_operator: AccountLoader<'info, RegisterOperator>,
```

Handler logic:

```
pub fn handle_initialize_register_operator(
    ctx: Context<InitializeRegisterOperatorCtx>,
) -> Result<()> {
    let mut register_operator = ctx.accounts.register_operator.load_init()?;
    register_operator.initialize(ctx.accounts.operator.key())?;
    Ok(())
}
```

Because the vault address is not included in the PDA seeds, one operator can operate system-wide.

Recommendations

Derive operator accounts **per vault** to reduce centralization risks:

- Include the **vault address** in the PDA derivation for `register_operator`.
- This ensures that operators are scoped to specific vaults, rather than controlling all vaults globally.

Example seed modification:

```
seeds = [
    OPERATOR_PREFIX.as_ref(),
    vault.key().as_ref(),
    operator.key().as_ref(),
],
```

This way, each vault has its own operator namespace, preventing a single operator from monopolizing registrations across all vaults.

[L-04] `handle_register` can exceed `max_cap`

`ix_register.rs`. `handle_register` checks `vault.total_registered_amount < vault.max_cap` before `vault.add_register_user(amount)?;`. `total_registered_amount` is value before update so it can be greater than `max_cap` after register.

Attacker can cause exceeding `max_cap` easily.

Recommendations

Update `handle_register`

```
    require!(
-       vault.total_registered_amount < vault.max_cap,
+       vault.total_registered_amount + amount < vault.max_cap,
        DistributionError::ReachedMaxCap
    );
```

[L-05] User removal does not refund fees, causing user fund loss

When a user registers, they pay a **creation fee** in addition to covering account rent. In the current implementation of `handle_remove_registered_user`, if a user is removed, the program reduces the registered user count but does **not refund the creation fee**.

This results in a **loss of funds for users**, as rent is already transferred to the designated rent receiver, but the creation fee remains locked. Over time, repeated user removals can lead to significant unrecoverable losses for participants.

Relevant code

```
pub fn handle_remove_registered_user(ctx: Context<RemoveRegisteredUserCtx>) -> Result<()> {
    let mut vault = ctx.accounts.vault.load_mut()?;
    let register = ctx.accounts.register.load()?;
    vault.remove_registered_user(register.amount)?;
    Ok(())
}
```

The function only updates vault state and does not reimburse the removed user.

Recommendations

Update the removal logic to refund the creation fee back to the user when they are deregistered:

Track the creation fee paid by each user.

- Store the fee amount in the `register` account or related state.

Refund the fee on removal.

- Transfer the creation fee back to the user during `handle_remove_registered_user`.