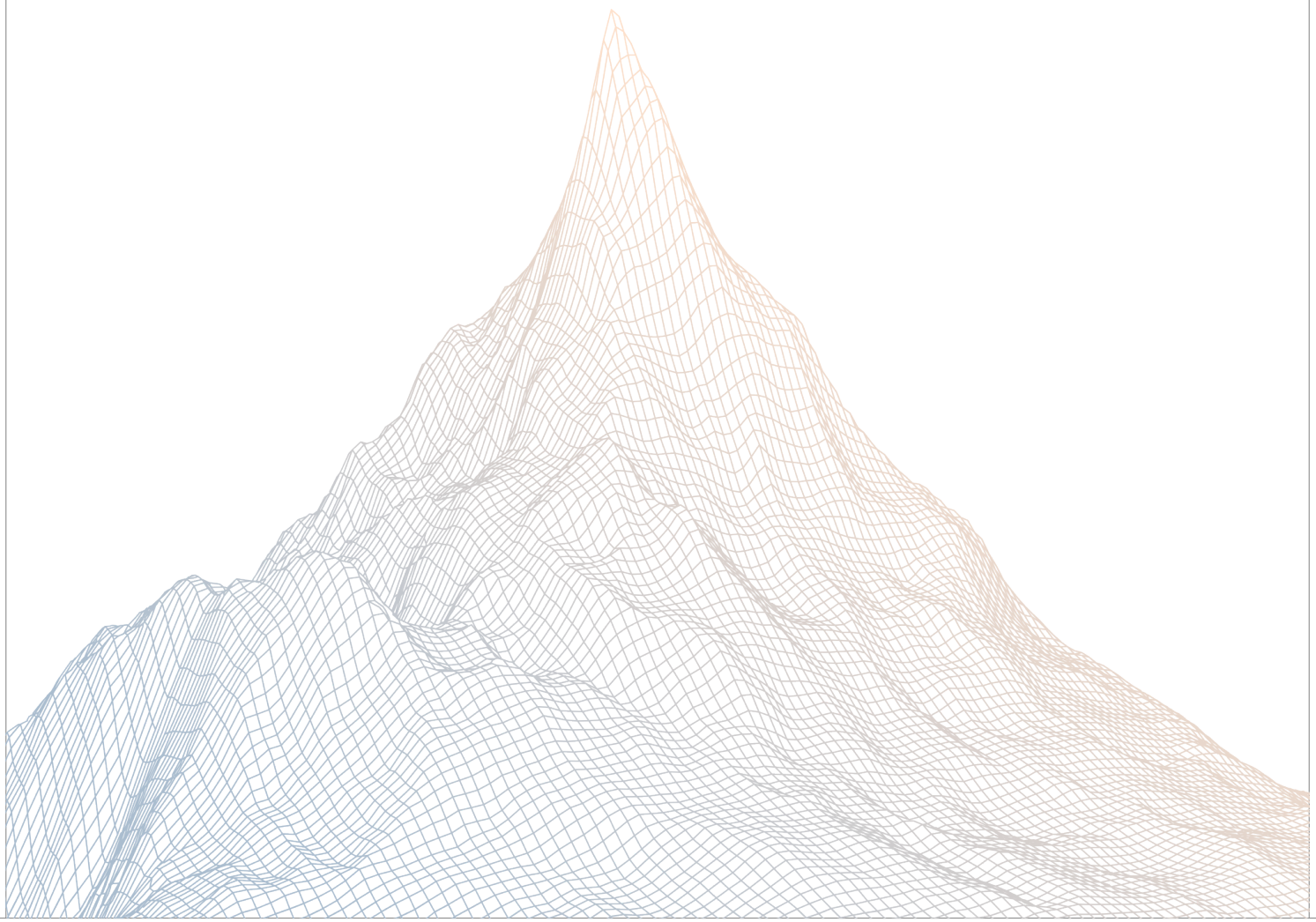
Zenith

ICM.RUN

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: December 22nd to December 24th, 2025

AUDITED BY: DadeKuma
oakcobalt

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About ICM.RUN

Internet Capital Markets (ICM) represent a transformative shift in how businesses, startups, research projects, movements, memes, and video games are launched and funded—as tokens on blockchain networks like Solana. This allows anyone with an internet connection and some SOL to participate in global capital markets, bypassing traditional equity holdings.

2.2 Scope

The engagement involved a review of the following targets:

Target	icm-staking
Repository	https://github.com/merklelabshq/icm-staking
Commit Hash	`dffb8ab0861b1a28f47366ea1996a070848220c2`
Files	`contract/programs/icm-staking/src/**/*.rs`

2.3 Audit Timeline

December 22, 2025	Audit start
December 24, 2025	Audit end
December 24, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	1
Medium Risk	2
Low Risk	2
Informational	2
Total Issues	**7**

3

Findings Summary

ID	Description	Status
H-1	Precision loss can cause reward loss	Resolved
M-1	Mutable staking settings apply retroactively to existing stakes	Resolved
M-2	Reward loss when staked_amount becomes 0	Resolved
L-1	Reward bucket pruning edge case	Resolved
L-2	Reward sandwiching with vulnerable parameter configuration	Resolved
I-1	DEV key can re-initialize admin and other key protocol parameters	Resolved
I-2	Empty reward buckets are unnecessarily added when settings change	Resolved

4

Findings

4.1 High Risk

A total of 1 high risk findings were identified.

[H-1] Precision loss can cause reward loss

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- state.rs#L91-L93

Description:

In `process_smoothing`, the current precision scaling of RPT (10^6) causes loss of rewards in common reward smoothing scenarios due to integer division truncation.

```
let rpt = (newly_unlocked_total as u128)
    .safe_mul(10u128.pow(6))?
    .safe_div(self.staked_amount as u128)?;
```

When `(newly_unlocked_total * 10^6) < staked_amount`, then `rpt = 0`, causing zero reward accrual even though buckets are updated with newly unlocked rewards. This vulnerable state is common due to several factors:

1. the reward token is 6 decimal and staked_amount is in 9 decimal;
2. `process_smoothing` can be invoked frequently to unlock small reward amounts triggered by user actions;
3. Staked amounts can be large relative to reward deposits, especially with low token price.

Suppose below scenario:

- Rewards: $1,000 USD = 1_000_000_000 USDC units,

  Stake amount: 47,809,921 tokens = 47_809_921_000_000_000 units,

  Smoothing period: 30 days,
- $1,000 USDC deposited→ creates reward bucket

- After 1 day, user calls `claim_token` → triggers `process_smoothing`
- Unlocked so far: (1_000_000_000 * 86_400) / 2_592_000 = 33_333_333 ,
- RPT = (33_333_333 * 1_000_000) / 47_809_921_000_000_000 -> 0

Note: Since `process_smoothing` can be user-triggered in smaller time intervals, even when RPT doesn't round down 0, small and incremental precision loss can cause significant loss. e.g. RPT rounds down from 1.04 -> 1, with current 10^6 precision scaling, this represents roughly $1912 USD loss per call (given the same stake amount as above).

Recommendations:

Consider using 10^12 or 10^18 precision scaling for RPT.

icm.run: Resolved with `@977e3850d5 ...`

Zenith: Verified.

4.2 Medium Risk

A total of 2 medium risk findings were identified.

[M-1] Mutable staking settings apply retroactively to existing stakes

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- programs/icm-staking/src/instructions/user/unstake_token.rs
- programs/icm-staking/src/instructions/user/withdraw_token.rs

Description:

Staking configuration parameters are globally shared and apply retroactively to all existing stakes. This affects `unstake` and `withdraw` operations.

The owner can modify these settings after users have already staked, altering the originally agreed-upon terms for ongoing staking positions.

This creates a scenario where users who stake tokens based on current unlock timelines may find their funds unexpectedly locked for extended periods if duration requirements are increased.

Recommendations:

Consider adding these settings to the `stake` account (which is initialized when a user stakes their funds) instead of using a global setting.

icm.run: Resolved with `@fa1f4ed3ef...`

Zenith: Verified.

[M-2] Reward loss when staked_amount becomes 0

SEVERITY: Medium	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Low

Target

- state.rs#L90-L95l

Description:

When `staked_amount` becomes 0, rewards can still be unlocked(e.g. the existing bucket reward distribution) `process_smoothing()` marks rewards as claimed but fails to accumulate them into `cml_reward_per_token`. This causes permanent loss of rewards that should be distributed to future stakers.

`process_smoothing()` calculates newly unlocked rewards and updates bucket `claimed_amount`, but only updates `cml_reward_per_token` if both `newly_unlocked_total > 0` and `staked_amount > 0`.

```
//contract/programs/icm-staking/src/state.rs
pub fn process_smoothing(&mut self, curr_time: i64) -> Result<()> {
...
    let unlocked_so_far = if elapsed >= period {
        total
    } else if elapsed > 0 {
        total.safe_mul(elapsed)?.safe_div(period)?
    } else {
        0
    };

    // compute newly unlocked for this bucket only
    let newly_unlocked = (unlocked_so_far as
    u64).saturating_sub(bucket.claimed_amount);

    if newly_unlocked > 0 {
        bucket.claimed_amount
    = bucket.claimed_amount.safe_add(newly_unlocked)?;
        newly_unlocked_total
    = newly_unlocked_total.safe_add(newly_unlocked)?;
    }
```

```
...
    // update global cumulative RPT
▷   if newly_unlocked_total > 0 && self.staked_amount > 0 {
        let rpt = (newly_unlocked_total as u128)
            .safe_mul(10u128.pow(6))?
            .safe_div(self.staked_amount as u128)?;

        self.cml_reward_per_token
    = self.cml_reward_per_token.safe_add(rpt)?;
    }
```

Suppose below scenario:

1. The bot deposits rewards into buckets daily.
2. Users unstake (e.g. liquidity or market reasons) → `global.staked_amount = 0`.
3. Some rewards unlocked based on `smoothing_period`
4. New user calls `stake_token()`.
5. Bucket `claimed_amount` updated , `cml_reward_per_token` not updated. Unlocked rewards are not claimable.

Note: The impact can also occur at protocol launch when the bot is scheduled to deposit rewards daily but no users have staked yet (`staked_amount == 0`).

Impacts: High Loss of funds. Unlocked rewards cannot be claimed or distributed later.

Likelihood: Medium - Low Users requiring liquidity or other market reasons may cause mass unstaking. Or at the early protocol launch, where no users have staked.

Recommendations:

1. Consider tracking missed rewards separately and applying them when `staked_amount` becomes non-zero.
2. Or consider checking `staked_amount>0` upfront before `newly_unlocked` calculations.

icm.run: Resolved with `@e3858c50c1 ...`

Zenith: Verified.

4.3 Low Risk

A total of 2 low risk findings were identified.

[L-1] Reward bucket pruning edge case

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: High

Target

- state.rs#L76

Description:

Pruning is skipped when there is exactly 1 reward bucket, even if it's fully claimed. This prevents automatic replacement with a new empty bucket.

The pruning logic in process_smoothing only runs when there are multiple buckets:

```
// prune buckets that are fully distributed
if self.reward_buckets.len() > 1 {
    self.reward_buckets
        .retain(|b| b.claimed_amount < b.total_amount);
}

if self.reward_buckets.is_empty() {
    self.reward_buckets.push(RewardBucket {
        start_time: curr_time,
        total_amount: 0,
        claimed_amount: 0,
    });
}
```

When len() == 1 and the bucket is fully claimed (claimed_amount == total_amount), pruning is skipped, the fully-claimed bucket remains, and no new empty bucket is created.

When `add_reward_bucket` is called: If within `new_reward_bucket_duration`, adds to the existing fully-claimed bucket, which allow new rewards to skip the smoothing period; If beyond new_reward_bucket_ducket_duration: creates a new bucket, leaving 2 buckets

(old fully-claimed + new). Pruning will work on the next process_smoothing call, which is inefficient.

Recommendations:

Change the condition to `≥ 1` to allow pruning and new bucket creation even with a single bucket.

icm.run: Resolved with `@c6bd30a23d ...`

Zenith: Verified.

[L-2] Reward sandwiching with vulnerable parameter configuration

SEVERITY: Low	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Low

Target

- state.rs#L61

Description:

The reward smoothing mechanism has a side effect where reward distribution accelerates as the current timestamp approaches the `smoothing_period` boundary. This creates reward sandwiching opportunities. For example, with `smoothing_period = 30 days`: Day 1 deposit 100 usd: Unlocks $3.33 immediately; Day 29 deposit 100 usd : Unlocks $96.67 immediately.

Reward Sandwiching The `deposit_token` call immediately updates the bucket's `total_amount`, and a subsequent user call that invokes `process_smoothing()` will accumulate the immediately unlocked rewards to Rpt. This allows users to:

1. Time or front-run `deposit_token` by staking before the deposit
2. Capture the RPT increase with the immediate rewards unlocked.
3. Claim rewards they didn't earn over time.

In normal conditions, the reward sandwiching can be mitigated by Lockup and cooldown periods. However, when `new_reward_bucket_duration` is close to or greater than `smoothing_period`, reward sandwiching becomes more profitable as shown above. Especially when`new_reward_bucket_duration > smoothing_period`, smoothing is effectively bypassed once the bucket has elapsed past `smoothing_period`.

Recommendations:

1. Consider adding check in `initialize` to ensure `new_reward_bucket_duration`, `lockup_duration` and `cooldown_duration` are in a safe range.
2. Or consider implementing a per-deposit time-weighted reward accounting where deposit is released at a constant pace.

icm.run: Resolved with `@304368048c ...`

Zenith: Verified.

4.4 Informational

A total of 2 informational findings were identified.

[I-1] DEV key can re-initialize admin and other key protocol parameters

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- state.rs#L31-L36

Description:

The `initialize` function can be called multiple times by the DEV key to re-initialize the Global account, overwriting the admin and all protocol parameters.

The `initialize` handler uses `init_if_needed`, which allows the function to execute even when the Global account already exists. It then unconditionally calls `global.initialize(args)`, which overwrites all fields, including `admin` and protocol parameters

```
#[account(
    init_if_needed,
    seeds = [seeds::GLOBAL_SEED],
    payer = dev,
    space = 8 + Global::INIT_SPACE,
    bump,
)]
pub global: Box<Account<'info, Global>>,
...
```

In `state.rs`, the `initialize` method overwrites all fields without checking if already initialized:

```
    pub fn initialize(&mut self, args: InitializeArgs) -> Result<()> {
        self.admin = args.admin;
```

```
        self.lockup_duration = args.lockup_duration;
        self.cooldown_duration = args.cooldown_duration;
        self.smoothing_period = args.smoothing_period;
        self.new_reward_bucket_duration = args.new_reward_bucket_duration;
...
```

This allows DEV to modify critical protocol parameters and admin, which can be considered unsafe. If DEV key is compromised, admin can be updated to a malicious address.

Recommendations:

1. If this is unintended, consider adding checks to prevent re-initialization. For example,

```
require!(
    global.admin == Pubkey::default(),
    IcmStakingError::AlreadyInitialized
);
```

2. If DEV key is allowed to update parameters, consider separating the initialization method from a dedicated `setParams()`.

icm.run: Resolved with `@fa1f4ed3ef ...`

Zenith: Verified.

[I-2] Empty reward buckets are unnecessarily added when settings change

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- programs/icm-staking/src/state.rs

Description:

The `Initialize` instruction can be called multiple times (for example to change settings such as the admin, or stake related settings).

The issue is that every time it's called a new bucket is unnecessarily added, even if the list is not empty.

All empty buckets will be pruned on the next `process_smoothing` operation.

Recommendations:

Consider adding a new bucket only when the bucket list is empty.

icm.run: Resolved with `@fa1f4ed3ef ...`

Zenith: Verified.