



SMART CONTRACT SECURITY AUDIT

Luxxcoin

HeroRewards Program

CLIENT

Luxxcoin

PLATFORM

Solana

LANGUAGE

Rust (Anchor)

REVIEW

November 2025

Document information

Project	Luxxcoin, HeroRewards Program
Client	Luxxcoin
Platform	Solana
Language	Rust (Anchor)
Repository	Luxxcoin-team/Luxx-heroReward-program: programs/hero-reward
Commit	Not recorded in the original report
Methods	Comprehensive manual code review, static analysis, threat-modeling, and best-practice compliance verification
Timeframe	November 2025

Document history

VERSION	DATE	CHANGE
1.0	November 2025	Original report issued View original report
1.1	October 2026	Reformatted into the Safe Edges standard template. Findings unchanged

This report was reformatted into the Safe Edges standard template in October 2026. Findings, severities, recommendations and code references are unchanged from the original report, which is preserved unchanged in the [Safe Edges Audit Portfolio on GitHub](#).

CONTENTS

Contents

01	Executive summary	3
02	Scope	4
03	Methodology	5
04	Severity and status	6
05	Findings summary	7
06	Detailed findings	8
	C-01 Incorrect Round End-Time Calculation Allows Immediate Round Closure	8
	H-01 Permanent Token Lockout Due to Missing Withdrawal Mechanism	9
	M-01 Missing Deadline Enforcement Allows Late Reward Distribution	10
	M-02 Governance Can Modify Round Cadence, Breaking Immutable Schedule Guarantee	12
	M-03 Missing Round Start Time Validation Allows Premature Round Initialization and Schedule Bypass	14
	M-04 Missing Prefund Validation Allows Premature Round Initialization and Distribution	15
	L-01 Missing Duplicate Address Check in distribute_batch Allows Double Payouts	17
	L-02 Missing Automatic Round Finalization Allows Inconsistent Round State	18
	Disclaimer	20

Executive summary

About Luxxcoin

Luxxcoin (\$LUX) is a Solana-based MemeFi project that turns meme culture into a rules-based, on-chain competition. Its HeroRewards program pays out \$LUX to top holders in recurring reward rounds.

Summary of the review

Safe Edges reviewed Luxxcoin's HeroRewards program on Solana, written in Rust with Anchor, in November 2025. Eight issues were identified: one Critical, one High, four Medium and two Low. The Critical issue computed each round's end time as its start time, so any round could be closed the moment it opened and no rewards distributed. Luxxcoin fixed seven issues and acknowledged one, the missing deadline check on reward distribution.

Findings by severity



Findings by status

STATUS	ISSUES
Resolved	7
Acknowledged	1

Scope

Repository	Luxxcoin-team/Luxx-heroReward-program: programs/hero-reward
Commit	Not recorded in the original report
Platform	Solana
Language	Rust (Anchor)

IN-SCOPE FILES

```
programs/hero-reward
```

Only the code at the commit above was reviewed. Changes made after this commit are not covered by this report.

Methodology

Methods recorded for this engagement: Comprehensive manual code review, static analysis, threat-modeling, and best-practice compliance verification.

Manual review

Auditors read the in-scope code line by line, tracing control flow, state changes and trust boundaries against the intended behaviour described by the client.

Threat modelling

We map privileged roles, external dependencies and value flows, then test the assumptions the code makes about each of them.

Functional and automated testing

Where tooling exists for the language and chain, we run static analysis and targeted tests to confirm suspected issues and check edge cases.

Fix review

After the client addresses the findings, we review the updated code and record the final status of each issue.

Severity and status

Severity levels

SEVERITY	DEFINITION
Critical	Direct loss or permanent lock of funds, or full compromise of the protocol, with few or no preconditions. Must be fixed before deployment.
High	Significant loss of funds, broken core functionality or manipulation of critical state under realistic conditions. Should be fixed before deployment.
Medium	Limited loss or degraded functionality, or issues that need specific conditions or privileged access to exploit. Should be fixed.
Low	Minor impact, low likelihood, or deviations from best practice. Fix when practical.
Informational	Code quality, documentation, gas and style observations with no direct security impact.

Status definitions

STATUS	DEFINITION
Resolved	The issue was fixed and the fix was verified.
Partially resolved	The risk was reduced but not fully removed.
Acknowledged	The client accepted the issue without changing the code.
Open	The issue was not addressed at the time of the report.
Not recorded	The original report did not record a resolution status for this issue.

Findings summary

ID	TITLE	SEVERITY	STATUS
C-01	Incorrect Round End-Time Calculation Allows Immediate Round Closure	Critical	● Resolved
H-01	Permanent Token Lockout Due to Missing Withdrawal Mechanism	High	● Resolved
M-01	Missing Deadline Enforcement Allows Late Reward Distribution	Medium	● Acknowledged
M-02	Governance Can Modify Round Cadence, Breaking Immutable Schedule Guarantee	Medium	● Resolved
M-03	Missing Round Start Time Validation Allows Premature Round Initialization and Schedule Bypass	Medium	● Resolved
M-04	Missing Prefund Validation Allows Premature Round Initialization and Distribution	Medium	● Resolved
L-01	Missing Duplicate Address Check in distribute_batch Allows Double Payouts	Low	● Resolved
L-02	Missing Automatic Round Finalization Allows Inconsistent Round State	Low	● Resolved

Detailed findings

C-01 Incorrect Round End-Time Calculation Allows Immediate Round Closure

Severity **Critical** Status ● Resolved

SUMMARY

The `close_round` function uses an incorrect formula to compute a rounds `end_time`. Instead of calculating (start of round + round duration), it computes just the rounds start time. This causes every round to appear ended the moment it begins. As a result, a round can be closed immediately after initialization—even within the same second—completely breaking the intended reward distribution timelines.

VULNERABILITY DETAILS

The `close_round` function computes a rounds end time using the following formula:

```
round_end_time = tge_ts + (round - 1) * round_len_secs
```

This formula does not calculate the end time of the current round. Instead, it computes the start time of the current round, because:

What this calculates:

- Round 1: $tge_ts + (1-1) * round_len = tge_ts + 0$ End time = Start of Round 1
- Round 2: $tge_ts + (2-1) * round_len = tge_ts + 60 \text{ days}$ End time = Start of Round 2
- Round 3: $tge_ts + (3-1) * round_len = tge_ts + 120 \text{ days}$ End time = Start of Round 3

As a result, every rounds end time equals its start time, meaning the round is considered over the moment it begins.

Thus, the condition:

```
let deadline_passed = current_time > round_end_time;
```

is immediately true as soon as the round is initialized—even one second after. This enables anyone to call `close_round` immediately after `initialize_round`, completely bypassing the intended full duration of the round.

IMPACT

Each round appears to end the moment it starts, allowing anyone to close it instantly.

Instant closure blocks `distribute_batch`, so no rewards are ever distributed for that round. All rounds can be force-closed back-to-back, collapsing the entire 60-day schedule.

This destroys reward fairness, as participants across all rounds receive nothing.

The also creates a DoS vector, permanently disabling reward distribution for any prematurely closed round

RECOMMENDATION

Modify the round_end_time calculation with correct formula .

```
pub fn close_round(ctx: Context<CloseRound>, round: u8) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    let round_state = &mut ctx.accounts.round_state;
    require!(round_state.distribution_started, HeroRewardsError::DistributionNotStarted);
    require!(!round_state.distribution_finished, HeroRewardsError::DistributionFinished);
    // Check if cap reached or deadline passed
    let current_time = Clock::get()?.unix_timestamp;
    -- let round_end_time = ctx.accounts.state.tge_ts + (round as i64 - 1) *
    ctx.accounts.state.round_len_secs;
    ++ let round_end_time = ctx.accounts.state.tge_ts + (round as i64) *
    ctx.accounts.state.round_len_secs;
    let deadline_passed = current_time > round_end_time;
    let cap_reached = round_state.claimed_amount >= round_state.total;
    require!(cap_reached || deadline_passed, HeroRewardsError::RoundNotClosable);
    round_state.distribution_finished = true;
    emit!(RoundClosed {
        round,
        total_claimed: round_state.claimed_amount,
        cap_reached,
        deadline_passed,
    });
    Ok(())
}
```

H-01 Permanent Token Lockout Due to Missing Withdrawal Mechanism

Severity High Status ● Resolved

SUMMARY

The HeroRewards program receives 45B LUX tokens via the prefund() step, storing them permanently in a PDA-owned vault. However, the program does not include any function that allows governance to recover or sweep undistributed tokens.

If any round is never initialized or distribution for a round is never executed, all tokens allocated to that round become permanently stuck, as only the publisher can trigger distribution and no fallback exists. This creates a situation where undistributed rewards can never be reclaimed, even by governance .

VULNERABILITY DETAILS

The vault ATA is owned by a PDA, and only the program can authorize transfers.

The program exposes no function to:

- Withdraw tokens back to governance,
- Sweep unused or expired round funds,

As a result:

If the publisher never initializes or distributes a round, its tokens are locked forever. Any unused funds from skipped or failed rounds cannot be reclaimed.

IMPACT

This vulnerability can lead to a permanent loss of treasury tokens, with up to 45B LUX becoming completely unrecoverable. Since governance has no mechanism to withdraw or reclaim stranded tokens from the PDA vault, any missed round, failed distribution, or publisher inactivity results in those funds being permanently locked.

RECOMMENDATION

Add a governance-only withdrawal / sweep function that allows the PDA vault to transfer unused or undistributed tokens back to the treasury or a designated authority .

M-01 Missing Deadline Enforcement Allows Late Reward Distribution

Severity **Medium** Status ● Acknowledged

SUMMARY

The `distribute_batch()` function does not enforce any deadline or round-expiry check. As a result, a publisher can continue distributing rewards for a round even after the rounds designated deadline has passed. This makes the round deadline meaningless and allows distributions to occur indefinitely until the round is manually closed. Even users may be forced to wait an indefinite amount of time for their rewards .

VULNERABILITY DETAILS

The protocol defines rounds with deadlines, implying that distribution should only occur within a specific valid time window. However, `distribute_batch()` only checks:

`distribution_started` `distribution_finished` `claimed_amount <= round_state.total`

It never checks whether the round deadline has passed.

```
pub fn distribute_batch(
    ctx: Context<DistributeBatch>,
    round: u8,
    addresses: Vec<Pubkey>,
    amounts: Vec<u64>,
) -> Result<> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    require!(addresses.len() <= 20, HeroRewardsError::TooManyWinners); // Limit to 20 per batch
    for safety
    require!(addresses.len() == amounts.len(), HeroRewardsError::InvalidAmount);
    require!(!addresses.is_empty(), HeroRewardsError::InvalidAmount);
    let round_state = &mut ctx.accounts.round_state;
    require!(round_state.distribution_started, HeroRewardsError::DistributionNotStarted);
    require!(!round_state.distribution_finished, HeroRewardsError::DistributionFinished);
    // @audit-issue : Missing check for round deadline should not passed
```

```
// Calculate total amount to distribute
let total_amount: u64 = amounts.iter()
    .try_fold(0u64, |acc, &amount| acc.checked_add(amount))
    .ok_or(HeroRewardsError::Overflow)?;
// Check round cap
let current_claimed = round_state.claimed_amount
    .checked_add(total_amount)
    .ok_or(HeroRewardsError::Overflow)?;
require!(current_claimed <= round_state.total, HeroRewardsError::ExceedsRoundCap);
    .....
}
```

This means:

- Even when the rounds end timestamp is exceeded,
- And even when previous rounds have correctly closed,

A publisher can still call `distribute_batch()` for any round where `distribution_finished` is not manually set to true.

IMPACT

Publisher can delay rewards indefinitely

Since the function having no deadline enforcement, a malicious publisher can intentionally delay distributions. Rounds remain open far beyond the intended timeline.

Users may wait an indefinite amount of time for their rewards

Because the round never auto-finalizes, users could be forced to wait for long periods—minutes, hours, or even days—depending on publishers .

RECOMMENDATION

Add a strict **deadline check inside `distribute_batch()`** .

```
pub fn distribute_batch(
    ctx: Context<DistributeBatch>,
    round: u8,
    addresses: Vec<Pubkey>,
    amounts: Vec<u64>,
) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    require!(addresses.len() <= 20, HeroRewardsError::TooManyWinners); // Limit to 20 per batch
    for safety
        require!(addresses.len() == amounts.len(), HeroRewardsError::InvalidAmount);
        require!(!addresses.is_empty(), HeroRewardsError::InvalidAmount);
    ++ let round_end_time = ctx.accounts.state.tge_ts + (round as i64) *
    ctx.accounts.state.round_len_secs;
    ++ require!(
    ++     Clock::get()?.unix_timestamp <= round_state.deadline,
    ++     HeroRewardsError::RoundExpired
    ++ );
    let round_state = &mut ctx.accounts.round_state;
    require!(round_state.distribution_started, HeroRewardsError::DistributionNotStarted);
    require!(!round_state.distribution_finished, HeroRewardsError::DistributionFinished);
    // Calculate total amount to distribute
    let total_amount: u64 = amounts.iter()
        .try_fold(0u64, |acc, &amount| acc.checked_add(amount))
        .ok_or(HeroRewardsError::Overflow)?;
    ....
}
```

M-02 Governance Can Modify Round Cadence, Breaking Immutable Schedule Guarantee

Severity **Medium** Status ● Resolved

SUMMARY

The protocol documentation explicitly states an "Immutable Schedule" with a "60-day cadence" for reward rounds. However, the `initialize` function accepts `round_len_secs` as a parameter that can be set by governance, allowing arbitrary modification of the round duration and breaking the promised immutable schedule.

VULNERABILITY DETAILS

The code constants define `ROUND_LEN_DAYS`: `u64 = 60`, suggesting a fixed 60-day round duration. However, in the `initialize` function:

```
pub fn initialize(
    ctx: Context<Initialize>,
    tge_ts: i64,
    round_len_secs: i64,
) -> Result<()> {
    ....
    state.governance = ctx.accounts.governance.key();
    state.mint_lux = ctx.accounts.mint_lux.key();
}
```

```

state.vault_lux = ctx.accounts.vault_lux.key();
state.tge_ts = tge_ts;
@-> state.round_len_secs = round_len_secs;
state.total_rounds = TOTAL_ROUNDS;
state.total_units = total_units;
state.per_round_cap = per_round_cap;
state.publisher = ctx.accounts.publisher.key();
state.paused = false;
state.prefunded = false;
emit!(Initialized {
    tge_ts,
    round_len_secs,
    total_rounds: TOTAL_ROUNDS,
    total_units,
    publisher: ctx.accounts.publisher.key(),
});
Ok(())
}

```

The `round_len_secs` parameter is stored directly without validation against the documented 60-day cadence. This allows governance to:

- Set rounds to any duration (shorter or longer than 60 days)
- Accelerate or delay token distribution arbitrarily
- Break the promised emission schedule

IMPACT

This allows governance to modify the round duration during initialization, breaking the documented 60-day immutable schedule guarantee .

RECOMMENDATION

Remove the `round_len_secs` parameter and use the constant value directly:

```

pub fn initialize(
    ctx: Context<Initialize>,
    tge_ts: i64,
    -- round_len_secs: i64,
) -> Result<()> {
    ...
    ++ let round_len_secs = (ROUND_LEN_DAYS * 24 * 60 * 60) as i64;
    state.governance = ctx.accounts.governance.key();
    state.mint_lux = ctx.accounts.mint_lux.key();
    state.vault_lux = ctx.accounts.vault_lux.key();
    state.tge_ts = tge_ts;
    ++ state.round_len_secs = round_len_secs;
    state.total_rounds = TOTAL_ROUNDS;
    state.total_units = total_units;
    state.per_round_cap = per_round_cap;
    state.publisher = ctx.accounts.publisher.key();
    state.paused = false;
    state.prefunded = false;
    emit!(Initialized {
        tge_ts,

```

```
round_len_secs,  
total_rounds: TOTAL_ROUNDS,  
total_units,  
publisher: ctx.accounts.publisher.key(),  
});  
Ok::<(),  
}
```

M-03 Missing Round Start Time Validation Allows Premature Round Initialization and Schedule Bypass

Severity **Medium** Status ● Resolved

SUMMARY

The `initialize_round()` function does not enforce that a round can only be initialized after its scheduled start time. While the protocol is designed to follow a strict time-based emission schedule derived from $tge_ts + (round-1) * round_len_secs$, the implementation having missing check for when rounds should actually start. This means the publisher can start any round whenever they want - even years early. This breaks the promised schedule and makes the timing rules useless.

VULNERABILITY DETAILS

The expected start time for any round is:

```
round_start_time = tge_ts + (round - 1) * round_len_secs
```

However, `initialize_round()` never checks whether:

`current_time >= round_start_time`

This allows the publisher to initialize any round at any time, with no regard for the chronological order or intended timing.

IMPACT

Malicious publisher can initialize all 45 rounds immediately after program initialization, then distribute tokens at any time without waiting for the scheduled round start times.

When rounds are initialized prematurely, higher-numbered rounds get disproportionately longer active duration than intended:

(a). Round 1: If initialized immediately, remains active for ~60 days until its natural deadline (b). Round 45: If initialized immediately, remains active for ~7.4 years until its natural deadline. Loss of Fairness Across Rounds

Because higher rounds get massively extended windows:

- Users in later rounds may receive rewards much earlier than expected
- This results in unequal and unpredictable reward timing, even when users follow all rules.

RECOMMENDATION

Add time validation to `initialize_round`:

```
pub fn initialize_round(
    ctx: Context<InitializeRound>,
    round: u8,
    total: u64,
) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(total as u128 == ctx.accounts.state.per_round_cap, HeroRewardsError::InvalidTotal);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    require!(ctx.accounts.state.prefunded, HeroRewardsError::NotPrefunded); // Also add prefund
    check
    // ADD TIME VALIDATION
    ++ let current_time = Clock::get()?.unix_timestamp;
    ++ let round_start_time = ctx.accounts.state.tge_ts + (round as i64 - 1) *
    ctx.accounts.state.round_len_secs;
    ++ require!(current_time >= round_start_time, HeroRewardsError::RoundNotStarted);
    let round_state = &mut ctx.accounts.round_state;
    require!(!round_state.distribution_started, HeroRewardsError::DistributionAlreadyStarted);
    round_state.round = round;
    round_state.total = total;
    round_state.distribution_started = true;
    round_state.distribution_finished = false;
    emit!(RoundInitialized {
        round,
        total,
        publisher: ctx.accounts.publisher.key(),
    });
    Ok(())
}
```

M-04 Missing Prefund Validation Allows Premature Round Initialization and Distribution

Severity Medium Status ● Resolved

SUMMARY

`initialize_round` and `distribute_batch` are missing checks to ensure that `state.prefunded` is true. Without this validation, the publisher can call these functions before the rewire multisig has funded the vault. As a result, rounds can be initialized before any tokens exist, and the system may attempt to distribute rewards from an empty vault.

VULNERABILITY DETAILS

Missing Prefund Checks in Critical Functions

```

**1.initialize_round**
pub fn initialize_round(...) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(total as u128 == ctx.accounts.state.per_round_cap, HeroRewardsError::InvalidTotal);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    // audit-issue : Missing require!(ctx.accounts.state.prefunded,
    HeroRewardsError::NotPrefunded);
    // ... proceeds to initialize round
}

```

This function allows new rounds to be initialized even when no tokens exist in the vault.

```

**2.distribute_batch**
pub fn distribute_batch(...) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    // @audit-issue : Missing require!(ctx.accounts.state.prefunded,
    HeroRewardsError::NotPrefunded);
    // ... attempts token transfers from empty vault
}

```

This results in batch distributions attempting SPL token transfers that are guaranteed to fail. State Tracking Exists But Unused

```

pub struct HeroRewardsState {
    pub prefunded: bool, // Tracked but never validated in round operations
}

```

The program correctly records whether the vault was prefunded, but this value is never used to enforce correct protocol behavior.

IMPACT

- Distribution attempts from empty vault will fail at the token program level
- Publishers and winners pay transaction fees for operations that cannot succeed
- Winners see distributions initiated but never receive tokens due to insufficient vault balance

RECOMMENDATION

Add the following check in both `initialize_round` and `distribute_batch`:

```

pub fn initialize_round(...) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(total as u128 == ctx.accounts.state.per_round_cap, HeroRewardsError::InvalidTotal);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    ++ require!(ctx.accounts.state.prefunded, HeroRewardsError::NotPrefunded);
    let round_state = &mut ctx.accounts.round_state;
    require!(!round_state.distribution_started, HeroRewardsError::DistributionAlreadyStarted);
    .....
}
pub fn distribute_batch(
    ctx: Context<DistributeBatch>,

```

```

round: u8,
addresses: Vec<Pubkey>,
amounts: Vec<u64>,
) -> Result<()> {
    require!(round > 0 && round <= TOTAL_ROUNDS, HeroRewardsError::InvalidRound);
    require!(!ctx.accounts.state.paused, HeroRewardsError::Paused);
    require!(addresses.len() <= 20, HeroRewardsError::TooManyWinners); // Limit to 20 per batch
    for safety
    require!(addresses.len() == amounts.len(), HeroRewardsError::InvalidAmount);
    require!(!addresses.is_empty(), HeroRewardsError::InvalidAmount);
    ++ require!(ctx.accounts.state.prefunded, HeroRewardsError::NotPrefunded);
    let round_state = &mut ctx.accounts.round_state;
    require!(round_state.distribution_started, HeroRewardsError::DistributionNotStarted);
    require!(!round_state.distribution_finished, HeroRewardsError::DistributionFinished);
    .....
}

```

L-01 Missing Duplicate Address Check in distribute_batch Allows Double Payouts

Severity Low Status ● Resolved

SUMMARY

The distribute_batch function does not verify that the list of recipient addresses contains unique entries. This allows a malicious publisher or misconfigured off-chain system to include the same winner multiple times in a single batch, causing that address to receive multiple payouts within one distribution call.

VULNERABILITY DETAILS

Inside distribute_batch, the program iterates through:

```

pub fn distribute_batch(
    ctx: Context<DistributeBatch>,
    round: u8,
    addresses: Vec<Pubkey>,
    amounts: Vec<u64>,
) -> Result<()> {
    .....
    for (idx, (winner_address, amount)) in addresses.iter().zip(amounts.iter()).enumerate() {
        ...
    }
    .....
}

```

However, it never checks whether the addresses vector contains duplicates. If the same address is inserted more than once, the logic will:

- Treat them as separate winners
- Perform multiple token transfers
- Inflate the total distribution for that batch (within cap)

IMPACT

This issue can cause double or multiple payouts to the same address within the same batch.

RECOMMENDATION

Add a uniqueness check before processing distribution .

L-02 Missing Automatic Round Finalization Allows Inconsistent Round State

Severity Low Status Resolved

SUMMARY

The `distribute_batch()` function never sets `round_state.distribution_finished = true`, even when the rounds distribution is fully completed. This allows a publisher to keep calling `distribute_batch()` indefinitely. While the round cap prevents over-distribution, the round is never formally closed, breaking expected round-lifecycle logic.

VULNERABILITY DETAILS

Each round has two flags:

```
distribution_started: bool
distribution_finished: bool
```

`initialize_round()` sets:

```
pub fn initialize_round(
    ctx: Context<InitializeRound>,
    round: u8,
    total: u64,
) -> Result<()> {
    .....
    distribution_started = true;
    distribution_finished = false;
    .....
}
```

But no where in `distribute_batch()` is `distribution_finished` set to true, even when:

```
pub fn distribute_batch(
    ctx: Context<DistributeBatch>,
    round: u8,
    addresses: Vec<Pubkey>,
    amounts: Vec<u64>,
) -> Result<()> {
```

..... `claimed_amount == total`

```
.....  
}
```

meaning the round is effectively complete.

Only `close_round()` sets:

```
pub fn close_round(ctx: Context<CloseRound>, round: u8) -> Result<()> {  
    .....  
    distribution_finished = true;  
    .....  
}
```

- If the publisher does not call `close_round()`, the round stays open forever.
- The round remains in active distribution mode even after its cap is reached.

IMPACT

The round state becomes misleading:

Rounds that are fully distributed still appear active.

RECOMMENDATION

Update distribution status when the cap is reached in the `distribute_batch()` function .

Disclaimer

This report is based on the code at the commit listed in the Scope section and on information the client provided during the review. It reflects Safe Edges' assessment at the time of the review and is not a guarantee that the code is free of vulnerabilities.

Changes made after the reviewed commit, deployment configuration, off-chain components and third-party dependencies are outside the scope of this report unless stated otherwise.

Security is ongoing. We recommend continued testing, monitoring and bug bounty coverage alongside any audit.

This report is not investment advice and is not an endorsement of the project, its team or any token.



Safe Edges is a Web3 security firm and official security partner for the Fuel, ENI and XI EcoChain ecosystems. We work across smart contract audits, Rust and WASM reviews, ZK circuit review, AI and MCP security, and penetration testing.

200+

PROJECTS AUDITED

\$4B+

TVL SECURED

0

POST-AUDIT HACKS