

Security Assessment Report

# Raydium CP Swap PR55

September 10, 2025

# Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Raydium CP Swap PR55.

The artifact was the source code of the following programs, excluding tests, in https://github.com/raydium-io/raydium-cp-swap.

The initial audit focused on the following versions and revealed 8 issues or questions.

| # | Task | Type | Commit |
|---|---|---|---|
| P1 | PR55 - Feat: support creator fee | Solana | 7732131 → 0ddfda5 |
| P2 | Beyond PR55 (Out of the scope of this audit) | Solana | 0ddfda5 |

This report provides a detailed description of the findings and their respective resolutions.

# Table of Contents

Result Overview ..... 3
Findings in Detail ..... 4
[P1-L-01] Inconsistent fee calculation in `swap_base_input` and `swap_base_output` ..... 4
[P1-I-01] Inconsistent requirement for `trade_fee_rate + creator_fee_rate` ..... 6
[P1-I-02] Potential division-by-zero in `split_creator_fee` ..... 7
[P1-I-03] Inaccurate fee rate limit ..... 8
[P1-I-04] Redundant bump calculation ..... 10
[P2-L-01] Imprecise `ObservationState` ..... 11
[P2-L-02] Inaccurate `lp_supply` ..... 15
[P2-I-01] Unnecessary signer on burn ..... 17
Appendix: Methodology and Scope of Work ..... 19

# Result Overview

| Issue | Impact | Status |
|---|---|---|
| **PR55 - FEAT: SUPPORT CREATOR FEE** | | |
| [P1-L-01] Inconsistent fee calculation in `swap_base_input` and `swap_base_output` | Low | Resolved |
| [P1-I-01] Inconsistent requirement for `trade_fee_rate + creator_fee_rate` | Info | Resolved |
| [P1-I-02] Potential division-by-zero in `split_creator_fee` | Info | Resolved |
| [P1-I-03] Inaccurate fee rate limit | Info | Resolved |
| [P1-I-04] Redundant bump calculation | Info | Resolved |
| **BEYOND PR55 (OUT OF THE SCOPE OF THIS AUDIT)** | | |
| [P2-L-01] Imprecise `ObservationState` | Low | Resolved |
| [P2-L-02] Inaccurate `lp_supply` | Low | Acknowledged |
| [P2-I-01] Unnecessary signer on burn | Info | Acknowledged |

# Findings in Detail

### PR55 - FEAT: SUPPORT CREATOR FEE

## [P1-L-01] Inconsistent fee calculation in `swap_base_input` and `swap_base_output`

> *Identified in commit* `0ddfda5`.

The fee calculation methods in `swap_base_input` and `swap_base_output` are inconsistent.

The calculation in `swap_base_input` is equivalent to `ceil(amount * trade_fee_rate) + ceil(amount * creator_fee_rate)`. In contrast, `swap_base_output` calculates the fee as `ceil(amount * (trade_fee_rate + creator_fee_rate))`. These two methods do not always produce the same result. The second result is always less than or equal to the first, with a maximum difference of 1.

For example, with an `input_amount` of 100, a `trade_fee_rate` of 1.5%, and a `creator_fee_rate` of 1.5%, the first method calculates a total fee of 4, while the second method results in a fee of 3.

Additionally, the first method creates a risk where the total fee could exceed the `input_amount`. This can cause an underflow and revert the transaction when `input_amount_less_fees` is calculated. This is because the sum of two separate ceiling division results can be larger than the original amount, even if `trade_fee_rate + creator_fee_rate <= FEE_RATE_DENOMINATOR_VALUE` is guaranteed.

```
/* programs/cp-swap/src/curve/calculator.rs */
095 | pub fn swap_base_input(
104 | ) -> Option<SwapResult> {
107 |     let trade_fee = Fees::trading_fee(input_amount, trade_fee_rate)?;
108 |     let input_amount_less_fees = if is_creator_fee_on_input {
109 |         creator_fee = Fees::creator_fee(input_amount, creator_fee_rate)?;
110 |         input_amount
111 |             .checked_sub(trade_fee)?
112 |             .checked_sub(creator_fee)?
115 |     };
142 | }

/* programs/cp-swap/src/curve/calculator.rs */
144 | pub fn swap_base_output(
153 | ) -> Option<SwapResult> {
172 |     let input_amount = if is_creator_fee_on_input {
173 |         let input_amount_with_fee = Fees::calculate_pre_fee_amount(
174 |             input_amount_swapped,
175 |             trade_fee_rate + creator_fee_rate,
176 |         )
177 |         .unwrap();
178 |         let total_fee = input_amount_with_fee - input_amount_swapped;
179 |         creator_fee = Fees::split_creator_fee(total_fee, trade_fee_rate, creator_fee_rate)?;
180 |         trade_fee = total_fee - creator_fee;
```

```
181 |         input_amount_with_fee
187 |     };
200 | }
```

It is recommended to use the second calculation method for both functions to ensure consistency.

### Resolution

Fixed by commit f361e34.

### PR55 - FEAT: SUPPORT CREATOR FEE

## [P1-I-01] Inconsistent requirement for `trade_fee_rate + creator_fee_rate`

*Identified in commit `0ddfda5`.*

The requirements for `trade_fee_rate` and `creator_fee_rate` are inconsistent between the `create_amm_config` and `update_amm_config` functions.

The `create_amm_config` function requires the sum of these two rates to be less than `FEE_RATE_DENOMINATOR_VALUE`. However, the update functions allow their sum to be less than or equal to `FEE_RATE_DENOMINATOR_VALUE`.

```
/* programs/cp-swap/src/lib.rs */
058 | pub fn create_amm_config(
066 | ) -> Result<()> {
067 |     assert!(trade_fee_rate + creator_fee_rate < FEE_RATE_DENOMINATOR_VALUE);
080 | }

/* programs/cp-swap/src/instructions/admin/update_config.rs */
047 | fn update_trade_fee_rate(amm_config: &mut Account<AmmConfig>, trade_fee_rate: u64) {
048 |     assert!(trade_fee_rate + amm_config.creator_fee_rate <= FEE_RATE_DENOMINATOR_VALUE);
049 |     amm_config.trade_fee_rate = trade_fee_rate;
050 | }

058 | fn update_creator_fee_rate(amm_config: &mut Account<AmmConfig>, creator_fee_rate: u64) {
059 |     assert!(creator_fee_rate + amm_config.trade_fee_rate <= FEE_RATE_DENOMINATOR_VALUE);
060 |     amm_config.creator_fee_rate = creator_fee_rate;
061 | }
```

### Resolution

Fixed by commit f361e34.

PR55 - FEAT: SUPPORT CREATOR FEE

## [P1-I-02] Potential division-by-zero in `split_creator_fee`

> *Identified in commit* `0ddfda5`.

In the `split_creator_fee` function, it is recommended to handle the case where `(trade_fee_rate + creator_fee_rate) == 0`, which will help prevent division-by-zero panics.

```
/* programs/cp-swap/src/curve/fees.rs */
062 | pub fn split_creator_fee(
063 |     total_fee: u128,
064 |     trade_fee_rate: u64,
065 |     creator_fee_rate: u64,
066 | ) -> Option<u128> {
067 |     floor_div(
068 |         total_fee,
069 |         u128::from(creator_fee_rate),
070 |         u128::from(trade_fee_rate + creator_fee_rate),
071 |     )
072 | }
```

### Resolution

Fixed by commit f361e34.

### PR55 - FEAT: SUPPORT CREATOR FEE

## [P1-I-03] Inaccurate fee rate limit

*Identified in commit `0ddfda5`.*

In the `update_amm_config` function, the `creator_fee_rate` and `trade_fee_rate` are updated by the admin. The sum of `creator_fee_rate + trade_fee_rate` is currently allowed to be less than or equal to `FEE_RATE_DENOMINATOR_VALUE`.

```
/* programs/cp-swap/src/instructions/admin/update_config.rs */
047 | fn update_trade_fee_rate(amm_config: &mut Account<AmmConfig>, trade_fee_rate: u64) {
048 |     assert!(trade_fee_rate + amm_config.creator_fee_rate <= FEE_RATE_DENOMINATOR_VALUE);
049 |     amm_config.trade_fee_rate = trade_fee_rate;
050 | }

/* programs/cp-swap/src/instructions/admin/update_config.rs */
058 | fn update_creator_fee_rate(amm_config: &mut Account<AmmConfig>, creator_fee_rate: u64) {
059 |     assert!(creator_fee_rate + amm_config.trade_fee_rate <= FEE_RATE_DENOMINATOR_VALUE);
060 |     amm_config.creator_fee_rate = creator_fee_rate;
061 | }
```

However, when the sum is exactly equal to `FEE_RATE_DENOMINATOR_VALUE`, the calculation in the `calculate_pre_fee_amount` function will trigger a division-by-zero panic.

```
/* programs/cp-swap/src/curve/calculator.rs */
172 | let input_amount = if is_creator_fee_on_input {
173 |     let input_amount_with_fee = Fees::calculate_pre_fee_amount(
174 |         input_amount_swapped,
175 |         trade_fee_rate + creator_fee_rate,
176 |     )
177 |     .unwrap();

/* programs/cp-swap/src/curve/fees.rs */
074 | pub fn calculate_pre_fee_amount(post_fee_amount: u128, trade_fee_rate: u64) -> Option<u128> {
075 |     if trade_fee_rate == 0 {
076 |         Some(post_fee_amount)
077 |     } else {
078 |         let numerator = post_fee_amount.checked_mul(u128::from(FEE_RATE_DENOMINATOR_VALUE))?;
079 |         let denominator =
080 |             u128::from(FEE_RATE_DENOMINATOR_VALUE).checked_sub(u128::from(trade_fee_rate))?;
081 |
082 |         numerator
083 |             .checked_add(denominator)?
084 |             .checked_sub(1)?
085 |             .checked_div(denominator)
086 |     }
087 | }
```

It is therefore recommended to enforce that `creator_fee_rate + trade_fee_rate` must be strictly less than `FEE_RATE_DENOMINATOR_VALUE`.

### Resolution

Fixed by commit f361e34.

PR55 - FEAT: SUPPORT CREATOR FEE

## [P1-I-04] Redundant bump calculation

*Identified in commit `0ddfda5`.*

In the `collect_creator_fee` instruction, the bump for the `authority` has already been calculated. So it is not necessary to call `Pubkey::find_program_address` again.

Consider using `pool_state.auth_bump`, similar to the approach in `collect_fund_fee` and `collect_protocol_fee`. Another option is to use `ctx.bumps.authority` provided by Anchor.

```
/* programs/cp-swap/src/instructions/collect_creator_fee.rs */
010 | pub struct CollectCreatorFee<'info> {
016 |     #[account(
017 |         seeds = [
018 |             crate::AUTH_SEED.as_bytes(),
019 |         ],
020 |         bump,
021 |     )]
022 |     pub authority: UncheckedAccount<'info>,
024 |     /// Pool state stores accumulated protocol fee amount
025 |     #[account(mut)]
026 |     pub pool_state: AccountLoader<'info, PoolState>,
086 | }
087 |
088 | pub fn collect_creator_fee(ctx: Context<CollectCreatorFee>) -> Result<()> {
096 |     let (_auth_pda, auth_bump) =
097 |         Pubkey::find_program_address(&[crate::AUTH_SEED.as_bytes()], &crate::id());
098 |     let signer_seeds: &[&[u8]] = &[crate::AUTH_SEED.as_bytes(), &[auth_bump]];
127 | }
```

### Resolution

Fixed by commit f361e34.

BEYOND PR55 (OUT OF THE SCOPE OF THIS AUDIT)

## [P2-L-01] Imprecise `ObservationState`

*Identified in commit `0ddfda5`.*

The current update strategy for `ObservationState` restricts updates to a maximum of one per `OBSERVATION_UPDATE_DURATION_DEFAULT` period. Any price changes occurring within this interval are ignored and do not contribute to `cumulative_token_0_price_x32` or `cumulative_token_1_price_x32`.

This design is fine in general. However, under certain conditions, it could lead to lower overall precision. The oracle will miss sharp price movements that happen inside the `OBSERVATION_UPDATE_DURATION_DEFAULT` window. For example, if the price suddenly increases and then decreases, these fluctuations are not included in the average price, even though they should be.

On the other hand, a predictable price oracle update interval increases the risk of price manipulation. Under the current `OBSERVATION_UPDATE_DURATION_DEFAULT` setting, a malicious user can perform a swap every 15 seconds and ensure their own swap is the last one within each update window. This enables them to manipulate the price recorded in the `ObservationState`. For pools with low trading volume or low liquidity, this type of manipulation is relatively easy to carry out.

```
/* programs/cp-swap/src/states/oracle.rs */
074 | pub fn update(
079 | ) {
087 |     } else {
088 |         let last_observation = self.observations[observation_index as usize];
089 |         let delta_time = block_timestamp.saturating_sub(last_observation.block_timestamp);
090 |         if delta_time < OBSERVATION_UPDATE_DURATION_DEFAULT {
091 |             return;
092 |         }
093 |         let delta_token_0_price_x32 = token_0_price_x32.checked_mul(delta_time.into()).unwrap();
094 |         let delta_token_1_price_x32 = token_1_price_x32.checked_mul(delta_time.into()).unwrap();
095 |         let next_observation_index = if observation_index as usize == OBSERVATION_NUM - 1 {
096 |             0
097 |         } else {
098 |             observation_index + 1
099 |         };
100 |         self.observations[next_observation_index as usize].block_timestamp = block_timestamp;
101 |         // cumulative_token_price_x32 only occupies the first 64 bits, and the remaining 64 bits are used to
↪  store overflow data
102 |         self.observations[next_observation_index as usize].cumulative_token_0_price_x32 =
103 |             last_observation
104 |                 .cumulative_token_0_price_x32
105 |                 .wrapping_add(delta_token_0_price_x32);
106 |         self.observations[next_observation_index as usize].cumulative_token_1_price_x32 =
107 |             last_observation
108 |                 .cumulative_token_1_price_x32
109 |                 .wrapping_add(delta_token_1_price_x32);
110 |         self.observation_index = next_observation_index;
```

```
111 |     }
112 | }
```

Since writing to on-chain accounts on Solana is not expensive, we could consider updating observations on every swap. To implement this, price changes weighted by time could be added to a temporary slot within the `OBSERVATION_UPDATE_DURATION_DEFAULT` period. After the period ends, this accumulated value would be moved into a new observation. This method would make the data in the observations more accurate and secure.

### Resolution

Fixed by commit f361e34.

> **Raydium Team:**
>
> For P2-L-01, I think your suggestion is very good. In fact, we also realized earlier that `OBSERVATION_UPDATE_DURATION_DEFAULT` (15s) updates an observation and some price fluctuations are missed in between, making the oracle's price less accurate. However, if we want to accumulate every price change within the time window, it doesn't seem so easy because the space of ObservationState is not enough. The new structure needs to add three fields:
>
> ```
> pub struct ObservationState {
> ...
> ...
> /// accumulator for current window since last finalized observation
> pub window_accumulated_token_0_price_x32: u128,
> pub window_accumulated_token_1_price_x32: u128,
> /// the last update timestamp
> pub last_update_timestamp: u64,
> }
> ```
>
> The original padding is not sufficient to cover these three fields, or are there other implementation methods? Do you have any other suggestions?
>
> **Sec3 Team:**
>
> One possible approach we came up to keep the values in the records more accurate without extra storage space:
>
> If the time between two records is less than the `OBSERVATION_UPDATE_DURATION_DEFAULT`, we can

cumulate the new price to the existing record in observations. We only create a new record when the time difference is greater than this limit.

First, check the time difference between the current timestamp and the timestamp of the last record (`self.observations[observation_index - 1].block_timestamp`). If the difference is not over the `OBSERVATION_UPDATE_DURATION_DEFAULT`, add the price change to the current record (`self.observations[observation_index]`) and update its timestamp to the current time.

If the time difference is over the limit, then create a new record.

```
// edge case: special handling when curr observation index == 0
let observation_delta_time = curr_timestamp - observations[observation_index - 1].block_timestamp;
if observation_delta_time >= OBSERVATION_UPDATE_DURATION_DEFAULT {
    // insert new observation
    return;
}
let delta_time = curr_timestamp - observations[observation_index].block_timestamp;
if delta_time > 0 {
    // write to the current observation cumulative price
    // update the current observation timestamp
}
```

Basically, this means we are always adding new price information to the last record instead of creating a new one, as long as the updates happen close enough in time.

But it may cause problems if an external program expects the time interval between records to be constant. However, in the original design, the intervals are not fixed anyway, and they could be quite different, especially for trading pools with low activity. So, we think this could be acceptable.

**Raydium Team:**

At first, I implemented a logic similar to yours, but I added a `last_update_timestamp` field because always using `delta_time = curr_timestamp - observations[observation_index].block_timestamp` for price accumulation is problematic.

However, yesterday I reverted this commit because this change made the current observation always in a variable state within a window, which could be confusing for external users. Externally, it is usually assumed that "historical observations are immutable," treating a certain index as an "anchor point." If the anchor point changes within the window, it can lead to inconsistencies in the TWAP/average price for the same interval across multiple queries. Moreover, many clients perform interpolation on observations. If the "previous point" is still changing, the interpolation baseline

> fluctuates, amplifying the error.
>
> Of course, under the current space constraints, it seems there is no better way to record price accumulation every second as much as possible.

BEYOND PR55 (OUT OF THE SCOPE OF THIS AUDIT)

## [P2-L-02] Inaccurate `lp_supply`

*Identified in commit `0ddfda5`.*

In both the `withdraw` and `deposit` instructions, the program calculates the corresponding `token_0` and `token_1` amounts using an internally tracked `lp_supply`.

However, because users can burn their LP tokens at any time, this tracked value can become inaccurate and larger than the actual supply.

```
/* programs/cp-swap/src/instructions/deposit.rs */
087 | pub fn deposit(
092 | ) -> Result<()> {
103 |     let results = CurveCalculator::lp_tokens_to_trading_tokens(
104 |         u128::from(lp_token_amount),
            // @audit: not real supply from mint, might be larger than the real supply from the mint.
105 |         u128::from(pool_state.lp_supply),
106 |         u128::from(total_token_0_amount),
107 |         u128::from(total_token_1_amount),
108 |         RoundDirection::Ceiling,
109 |     )

/* programs/cp-swap/src/instructions/withdraw.rs */
099 | pub fn withdraw(
104 | ) -> Result<()> {
116 |     let results = CurveCalculator::lp_tokens_to_trading_tokens(
117 |         u128::from(lp_token_amount),
            // @audit: not real supply from mint, might be larger than the real supply from the mint.
118 |         u128::from(pool_state.lp_supply),
119 |         u128::from(total_token_0_amount),
120 |         u128::from(total_token_1_amount),
121 |         RoundDirection::Floor,
122 |     )
```

While this issue does not cause direct losses to other users or the protocol, it makes the funds linked to the burned LP tokens impossible to withdraw. A better method is to use the actual `lp_mint.supply`. This would allow the locked funds to be shared among all remaining liquidity providers.

Additionally, switching to `lp_mint.supply` requires changing how `lock_lp_amount` is handled in both the `initialize` and `initialize_with_permission` functions.

Instead of just subtracting this amount from the initial number of LP tokens, the `lock_lp_amount` must be physically minted, and should then be sent to a secure vault or an unusable address to ensure they are never burned. This step is critical because if the locked tokens are not minted or can be burned, the `lp_mint.supply` could become zero, certain calculations would fail because of a division by zero error, leading to a denial of service.

```
/* programs/cp-swap/src/instructions/initialize.rs */
164 | pub fn initialize(
169 | ) -> Result<()> {
281 |     token::token_mint_to(
282 |         ctx.accounts.authority.to_account_info(),
283 |         ctx.accounts.token_program.to_account_info(),
284 |         ctx.accounts.lp_mint.to_account_info(),
285 |         ctx.accounts.creator_lp_token.to_account_info(),
286 |         liquidity
            // @audit: Currently, the lock_lp_amount is not minted, just accounted for in pool_state.lp_supply. To
            ↪  use lp_mint.supply safely, these locked tokens must actually be minted.
287 |             .checked_sub(lock_lp_amount)
288 |             .ok_or(ErrorCode::InitLpAmountTooLess)?,
289 |         &[&[crate::AUTH_SEED.as_bytes(), &[ctx.bumps.authority]]],
290 |     )?;
336 | }

/* programs/cp-swap/src/instructions/initialize_with_permission.rs */
175 | pub fn initialize_with_permission(
181 | ) -> Result<()> {
294 |     token::token_mint_to(
295 |         ctx.accounts.authority.to_account_info(),
296 |         ctx.accounts.token_program.to_account_info(),
297 |         ctx.accounts.lp_mint.to_account_info(),
298 |         ctx.accounts.payer_lp_token.to_account_info(),
299 |         liquidity
300 |             .checked_sub(lock_lp_amount)
301 |             .ok_or(ErrorCode::InitLpAmountTooLess)?,
302 |         &[&[crate::AUTH_SEED.as_bytes(), &[ctx.bumps.authority]]],
303 |     )?;
349 | }
```

### Resolution

The team acknowledged this finding and clarified that:

> 1. In the early design, we used `lp_mint.supply` for calculations. However, we later discovered that some users burned all their LP tokens, causing `lp_mint.supply` to be `0`, making it impossible to deposit assets into the pool. Therefore, we introduced the `pool_state.lp_supply` field in the pool.
> 2. Regarding the lock LP feature, we made the corresponding changes during a previous audit.
>
> Regarding this part, since there is no security risk to the user's funds, and the design does not want to disperse this part of the LP rights to other liquidity providers due to users burning LP, we do not plan to make any further changes.

BEYOND PR55 (OUT OF THE SCOPE OF THIS AUDIT)

## [P2-I-01] Unnecessary signer on burn

*Identified in commit `0ddfda5`.*

In the `withdraw` function, when the program burns LP tokens from a token account owned by the `owner` (who is an external user), it also provides mint authority's seeds.

```
/* programs/cp-swap/src/instructions/withdraw.rs */
014 | pub struct Withdraw<'info> {
016 |     pub owner: Signer<'info>,
019 |     #[account(
020 |         seeds = [
021 |             crate::AUTH_SEED.as_bytes(),
022 |         ],
023 |         bump,
024 |     )]
025 |     pub authority: UncheckedAccount<'info>,
032 |     #[account(
033 |         mut,
034 |         token::authority = owner
035 |     )]
036 |     pub owner_lp_token: Box<InterfaceAccount<'info, TokenAccount>>,
097 | }

099 | pub fn withdraw(
104 | ) -> Result<()> {
179 |     token_burn(
180 |         ctx.accounts.owner.to_account_info(),
181 |         ctx.accounts.token_program.to_account_info(),
182 |         ctx.accounts.lp_mint.to_account_info(),
183 |         ctx.accounts.owner_lp_token.to_account_info(),
184 |         lp_token_amount,
185 |         &[&[crate::AUTH_SEED.as_bytes(), &[pool_state.auth_bump]]],
186 |     )?;
220 | }

/* programs/cp-swap/src/utils/token.rs */
103 | pub fn token_burn<'a>(
104 |     authority: AccountInfo<'a>,
105 |     token_program: AccountInfo<'a>,
106 |     mint: AccountInfo<'a>,
107 |     from: AccountInfo<'a>,
108 |     amount: u64,
109 |     signer_seeds: &[&[&[u8]]],
110 | ) -> Result<()> {
111 |     token_2022::burn(
112 |         CpiContext::new_with_signer(
113 |             token_program.to_account_info(),
114 |             token_2022::Burn {
115 |                 from,
116 |                 authority,
117 |                 mint,
118 |             },
119 |             signer_seeds,
```

```
120 |         ),
121 |         amount,
122 |     )
123 | }
```

However, the `burn` instruction for the token and token-2022 programs only requires the token account's owner or delegate as a signer. It does not require the mint authority or any other party to sign. Since the user (`owner`) is already signing the transaction, the additional program signature is not needed.

```
/* https://github.com/solana-program/token-2022/blob/2e799e3/program/src/instruction.rs#L221-L239 */
221 | /// Burns tokens by removing them from an account.  `Burn` does not support
222 | /// accounts associated with the native mint, use `CloseAccount` instead.
223 | ///
224 | /// Accounts expected by this instruction:
225 | ///
226 | ///   * Single owner/delegate
227 | ///   0. `[writable]` The account to burn from.
228 | ///   1. `[writable]` The token mint.
229 | ///   2. `[signer]` The account's owner/delegate.
230 | ///
231 | ///   * Multisignature owner/delegate
232 | ///   0. `[writable]` The account to burn from.
233 | ///   1. `[writable]` The token mint.
234 | ///   2. `[]` The account's multisignature owner/delegate.
235 | ///   3. ..`3+M` `[signer]` M signer accounts.
236 | Burn {
237 |     /// The amount of tokens to burn.
238 |     amount: u64,
239 | },
```

It is recommended to remove the extra signer.

### Resolution

The team acknowledged this finding.

# Appendix: Methodology and Scope of Work

Assisted by the Sec3 Scanner developed in-house, the manual audit particularly focused on the following work items:

- Check common security issues.
- Check program logic implementation against available design specifications.
- Check poor coding practices and unsafe behavior.
- The soundness of the economics design and algorithm is out of scope of this work

# DISCLAIMER

The instance report ("Report") was prepared pursuant to an agreement between Coderrect Inc. d/b/a Sec3 (the "Company") and Reactor Labs Ltd (the "Client''). This Report solely includes the results of a technical assessment of a specific build and/or version of the Client's code specified in the Report ("Assessed Code") by the Company. The sole purpose of the Report is to provide the Client with the results of the technical assessment of the Assessed Code. The Report does not apply to any other version and/or build of the Assessed Code. Regardless of the contents of the Report, the Report does not (and should not be interpreted to) provide any warranty, representation or covenant that the Assessed Code: (i) is error and/or bug free, (ii) has no security vulnerabilities, and/or (iii) does not infringe any third-party rights. Moreover, the Report is not, and should not be considered, an endorsement by the Company of the Assessed Code and/or of the Client. Finally, the Report should not be considered investment advice or a recommendation to invest in the Assessed Code and/or the Client.

This Report is considered null and void if the Report (or any portion thereof) is altered in any manner.

# ABOUT

The Sec3 audit team comprises a group of computer science professors, researchers, and industry veterans with extensive experience in smart contract security, program analysis, testing, and formal verification.  We are also building automated security tools that incorporate static analysis, penetration testing, and formal verification.

At Sec3, we identify and eliminate security vulnerabilities through the most rigorous process and aided by the most advanced analysis tools.

For more information, check out our website and follow us on twitter.

