Solar DEX Audit Report

Prepared by KupiaSec

Version 1.0

Auditors

KupiaSec

Dec 6, 2024

Contents

1 About KupiaSec

KupiaSec is a team of Web3 security experts that operates with transparency and a meritocratic spirit.

KupiaSec executes the modified MPA model for the Private Audits, a.k.a. **Solo Audit by a Lead + Internal Competition + Mitigation Review**.

- Solo Audit by a Lead

  KupiaSec assigns a senior auditor as a Lead Auditor based on the protocol category. The Lead Auditor is responsible for the first phase and will be the main point of contact for the client. The Lead Auditor shares the analysis and findings with the team.

- Internal Competition

  KupiaSec assigns 5~7 assist auditors to conduct the second phase. The auditors compete to find the most issues and the best solutions. This phase ensures the protocol goes through a rigorous review process by "many eyes" in a competitive environment.

- Mitigation Review

  After the protocol team has fixed the issues, KupiaSec conducts a final review to ensure all the issues are resolved.

2 Disclaimer

The KupiaSec team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Solar DEX is an automated market maker (AMM) deployed on the Eclipse blockchain which enables lightning-fast trades and permissionless pool creation.

Solar DEX's CLMM is forked from Raydium's AMM.

5 Audit Scope

Diff updates from Raydium's CLMM programs.

Summary of Commits

Project Name	Solar DEX
Repository	solar-clmm
Initial Commit	8614cddc9321...
Mitigation Commit	52741c4dcdeb...

6 Executive Summary

KupiaSec executed a modified Multi-Phase Audit model, a.k.a. **Solo Audit by a Lead + Internal Competition + Mitigation Review**.

@auditsea conducted the audit as the Lead Auditor and 2 auditors competed in the second phase.

Execution Timeline

Phase-1: Audit by a Lead	Dec 2 - Dec 3
Phase-2: Internal Competition	Dec 3 - Dec 5
Initial Report Delivery	Dec 6, 2024
Phase-3: Mitigation Review	Dec 6
Final Report Delivery	Dec 6, 2024

Issues Found

Critical Risk	0
High Risk	1
Medium Risk	1
Low Risk	0
Informational	2
Gas Optimizations	0
Total Issues	4

Summary of Findings

[H-1] Incorrect tick passed to observation data that results in invalid oracle price entry	Open
[M-1] Current structure of observation entries opens up a vulnerability of TWAP manipulation	Open
[I-1] Tick limits differ between `tick_math.rs` and `error.rs`.	Open
[I-2] Incorrect comment in `create_pool.rs`.	Open

7 Findings

7.1 High Risk

7.1.1 Incorrect tick passed to observation data that results in invalid oracle price entry

Description: The CLMM protocol contains a logic to manage TWAP price by implementing observation entries that keep tracking of cumulative ticks. These entries are modified during swap actions that modifies price of tokens(ticks).

```
// update tick
if state.tick != pool_state.tick_current {
    pool_state.tick_current = state.tick;
}
// update the previous price to the observation
observation_state.update(block_timestamp, pool_state.tick_current);
```

Here's the code snippet of `swap_internal` function of `swap.rs`, which adds a new entry to observation array.

However, it includes a severe vulnerability where it passes new tick as a parameter, because the passed parameter means the tick that has been kept so far.

As a result, the observation data mistakenly puts incorrect price data into its entry that affects TWAP.

Impact: Incorrect price data in oracle observation entry.

Proof Of Concept:

Recommended Mitigation: The old tick should be passed instead of the updated tick. Also, it's recommended to update observation data only when the tick is modified so that unnecessary entries don't get into oracle data.

NOTE: This has been update in Raydium codebase, but not in Solar.

```
// update tick
if state.tick != pool_state.tick_current {
+   // update the previous price to the observation
+   observation_state.update(block_timestamp, pool_state.tick_current);
    pool_state.tick_current = state.tick;


}
- // update the previous price to the observation
- observation_state.update(block_timestamp, pool_state.tick_current);
```

Client: Fixed in commit.

KupiaSec: Verified.

7.2 Medium Risk

7.2.1 Current structure of observation entries opens up a vulnerability of TWAP manipulation

Description: Current design of TWAP oracle is implemented by using maximum of 100 entries of cumulative tick entires. The reason I assume it only uses 100 entries for accumulative ticks(while Uniswap uses 65535 entries) would be because of storage cost to be payed for account's rent exempt.

And for this reason, the TWAP oracle only inputs entries when it has passed more than 15 seconds since last entry of price oracle data, as implemented in `oracle.rs`:

```
pub fn update(&mut self, block_timestamp: u32, tick: i32) {
    let observation_index = self.observation_index;
    if !self.initialized {
        self.initialized = true;
        self.observations[observation_index as usize].block_timestamp = block_timestamp;
        self.observations[observation_index as usize].tick_cumulative = 0;
    } else {
        let last_observation = self.observations[observation_index as usize];
        let delta_time = block_timestamp.saturating_sub(last_observation.block_timestamp);
>       if delta_time < OBSERVATION_UPDATE_DURATION_DEFAULT {
            return;
        }

        let delta_tick_cumulative = i64::from(tick).checked_mul(delta_time.into()).unwrap();
        let next_observation_index = if observation_index as usize == OBSERVATION_NUM - 1 {
            0
        } else {
            observation_index + 1
        };
        self.observations[next_observation_index as usize].block_timestamp = block_timestamp;
        self.observations[next_observation_index as usize].tick_cumulative = last_observation
            .tick_cumulative
            .wrapping_add(delta_tick_cumulative);
        self.observation_index = next_observation_index;
    }
}
```

While this makes sense in terms of using storage effective, but this causes a vulnerability where a malicious user can manipulate TWAP by putting any price data into oracle entry.

Here's a scenario how it would work:

- At time of T, a new observation data is written with tick TICK.
- Some swaps happen during [T, T+10], which modifies the tick but oracle entry is not added because of duration validation (15s).
- Alice, a malicious user, moves the tick back to TICK between [T+13, T+15), and calls another swap with dust amount, which will eventually put TICK as new observation data.
- Alice can repeats this process to manipulate oracle price entry that affects TWAP.

This kind of behavior from Alice doesn't usually have incentive since Alice will be paying gas and swap fees but this becomes serious when other protocols rely on Solar's TWAP as price oracle. In this case, Alice is able to manipulate the price which will give her advantage eventually.

Impact: TWAP can be manipulated.

Proof Of Concept:

Recommended Mitigation: It's recommended to increase number of entries and remove observation duration.

Client: Acknowledged.

KupiaSec: Acknowledged.

7.3 Informational

7.3.1 Tick limits differ between `tick_math.rs` and `error.rs`.

Description: In `error.rs`, the error messages for `TickLowerOverflow` and `TickUpperOverflow` indicate that the tick limit is `221818`:

```
    #[msg("The tick must be greater, or equal to the minimum tick(-221818)")]
    TickLowerOverflow,
    #[msg("The tick must be lesser than, or equal to the maximum tick(221818)")]
    TickUpperOverflow,
```

However, in `tick_math.rs`, the tick limit is set to `443636`:

```
    /// The minimum tick
    pub const MIN_TICK: i32 = -443636;
    /// The minimum tick
    pub const MAX_TICK: i32 = -MIN_TICK;
```

This discrepancy appears in several places, including `tick_array.rs`.

```
pub fn check_tick_array_start_index(
    tick_array_start_index: i32,
    tick_index: i32,
    tick_spacing: u16,
) -> Result<()> {
    require!(
@>      tick_index >= tick_math::MIN_TICK,
        ErrorCode::TickLowerOverflow
    );
    require!(
@>      tick_index <= tick_math::MAX_TICK,
        ErrorCode::TickUpperOverflow
    );
    require_eq!(0, tick_index % i32::from(tick_spacing));
    let expect_start_index = TickArrayState::get_array_start_index(tick_index, tick_spacing);
    require_eq!(tick_array_start_index, expect_start_index);
    Ok(())
}
```

The error messages have been updated from `443636` in `raydium-clmm` to `221818` in `solar-clmm`. This indicates that the protocol team intended to change the tick limit from `443636` in raydium-clmm to `221818` in solar-clmm.

Impact: Incorrect implementation of the tick limit check.

Proof Of Concept:

Recommended Mitigation: Adjust the tick limit.

```
    /// The minimum tick
-   pub const MIN_TICK: i32 = -443636;
+   pub const MIN_TICK: i32 = -221818;
    /// The minimum tick
    pub const MAX_TICK: i32 = -MIN_TICK;
```

Client:

KupiaSec:

7.3.2 Incorrect comment in `create_pool.rs`.

Description: In `create_pool.rs`, the comment on line 31 states that `token_mint_0` must be greater than `token_mint_1`. However, the actual constraint on line 33 specifies that `token_mint_0` must be less than `token_mint_1`.

```
    #[derive(Accounts)]
    pub struct CreatePool<'info> {
        ...

31      /// Token_0 mint, the key must grater then token_1 mint.
        #[account(
33          constraint = token_mint_0.key() < token_mint_1.key(),
            mint::token_program = token_program_0
        )]
        pub token_mint_0: Box<InterfaceAccount<'info, Mint>>,

        /// Token_1 mint
        #[account(
            mint::token_program = token_program_1
        )]
        pub token_mint_1: Box<InterfaceAccount<'info, Mint>>,

        ...
    }
```

Impact: Proof Of Concept:

Recommended Mitigation: Change the comment.

```
#[derive(Accounts)]
pub struct CreatePool<'info> {
    ...

-   /// Token_0 mint, the key must grater then token_1 mint.
+   /// Token_0 mint, the key must smaller than token_1 mint.
    #[account(
        constraint = token_mint_0.key() < token_mint_1.key(),
        mint::token_program = token_program_0
    )]
    pub token_mint_0: Box<InterfaceAccount<'info, Mint>>,

    /// Token_1 mint
    #[account(
        mint::token_program = token_program_1
    )]
    pub token_mint_1: Box<InterfaceAccount<'info, Mint>>,

    ...
}
```

Client:

KupiaSec: