Raydium AMM Program Audit Report

10.6.2023

– Mad Shield

Introduction

Mad Shield conducted a full security audit and vulnerability analysis on the Raydium AMM Program. The audit took approximately ~8 weeks to complete starting from March 15th and ending on May 15th 2023. This report briefly covers the program's workflow along with a short description of our general findings.

Overall, we are glad to confirm that no critical vulnerabilities were found in the code. Apart from a number of low and medium level bugs that have since been communicated to the team, the program operates within the boundaries of its specification.

Overview

The Raydium AMM is a decentralized exchange (DEX) protocol built on the Solana blockchain. Leveraging the high-speed and low transaction fees of Solana, it provides a fast and cost-efficient terminal for market participants to access and provide liquidity to trade and swap digital assets in a decentralized manner. At the time of this publication, TVL/Volume

At its core, the program is an implementation of a constant function market maker (CFMM) for the Solana Virtual Machine (SVM). For the hypothetical trading pair X-Y, the Raydium AMM keeps track of

$$R_x R_y = K$$

Where R_X and R_y represent the amount of X and Y in associated market reserve vaults deposited by liquidity providers (LPs). A Δ_x amount of X can then be swapped for Δ_y amount of Y such that

$$(R_x + \Delta_x)(R_y - \Delta_y) = R_x R_y = K$$

thereby, preserving the constant product.

A novel feature of the Raydium AMM that differentiates it from other AMMs such as Uniswap is that it additionally composes with the ecosystem-wide liquidity layer. The program utilizes its pool of assets to place limit orders in a fibonacci sequence on Openbook, the primary central limit order book (CLOB) of Solana. This allows for deeper liquidity on the order book while simultaneously earning extra fees for the liquidity providers through 3rd party order flow.

On the technical side, this on-chain market making strategy is a state machine prompted by an off-chain crank client to plan, place or cancel limit orders on the order book. State transitions occur in response to changes in market conditions like price or liquidity to accordingly adjust the AMM positions.

The other notable component of the Raydium AMM is the liquidity mining program. Upon depositing liquidity, the AMM mints a proportional amount of a LP token to the user representing their share of the total liquidity in the pool. Apart from trading fees, the Raydium team offers a suite of programs for staking and farming these LP tokens to earn yet more rewards in order to incentivize providing liquidity.

Mad Shield has completed audits on both Raydium stake and farm programs as well. We will further elaborate on the reward mechanism in relation to the Raydium AMM in the upcoming audit documentation.

In the following, we first outline the program's design by dissecting its main components and look at the swapping process from the perspective of the data structures. Then we cover the AMM's market making provision on the orderbook. Finally, we wrap up by enumerating our findings, shortly explaining each threat and the recommended fix.

Account Structure

In this section, we take a closer look at the program's implementation in Solana's programming model. Here we briefly explain the adopted account structure.

- **AMM Info:**

  The Amm Info is the primary account of the design and stores data that is needed for the purposes of both swapping assets and market making on the CLOB. The essential fields of this account are highlighted in Fig 1.

  A key account in this struct is the underlying Openbook market. There is one unique AMM per Openbook market specified as a Solana PDA with the following seeds:

  ```
  [program_id, market_account, "amm_associated_seed"]
  ```

  During the initialization of this account, token accounts are created for both quote and base currencies and an initial amount of each token is transferred to the vaults. The initial liquidity is calculated as the geometric mean of the deposited amounts

and defined as the `lp_amount` parameter. To avoid the price of the smallest unit of the LP mint to be unfeasibly high and unattainable for small liquidity providers in the future, the AMM does not mint the whole amount of LP tokens to the first depositor, instead locking a portion of the assets in the pool permanently [1].

```
pub struct AmmInfo {
    pub status: u64,
    pub nonce: u64,
    pub order_num: u64,
    pub depth: u64,
    pub coin_decimals: u64,
    pub pc_decimals: u64,
    pub state: u64,
    pub sys_decimal_value: u64,
    pub fees: Fees,
    pub out_put: OutPutData,
    pub token_coin: Pubkey,
    pub token_pc: Pubkey,
    pub lp_mint: Pubkey,
    pub open_orders: Pubkey,
    pub market: Pubkey,
    pub serum_dex: Pubkey,
    pub target_orders: Pubkey,
    pub amm_owner: Pubkey,
    pub lp_amount: u64,
}
```

Fig 1. The layout of the AmmInfo account.

The max number of limit orders that the AMM will open on the orderbook at any time is bounded by `order_num`. The `depth` determines the range around current price within which the AMM can place orders. The other account related to the market making is the Open Orders account which is also a PDA but is owned by the Openbook program and record keeps information such as the order_id, side and relative price-time of the limit orders on Openbook.

As the quote and price mints can have different decimals, the AMM uses a global `sys_decimal_value` that gives the maximum precision to compute order sizes. Most of the time it is equal to the bigger decimal between the two but it also takes into account the underlying Openbook market's lot sizes that order amounts are measured in.

Some of the market statistics are stored on-chain in the `out_put` struct. This includes details such as market open time, the total volume ever swapped and the total fee earned with its breakdown in each asset. The most important data points in this structure are `need_take_pnl_coin` and `need_take_pnl_pc` which are the fees that the Raydium authority has earned but has yet to claim.

The AMM subtracts this amount from the amount of tokens in each reserve as for the fees to not count toward the available liquidity in the pool when determining the result of a swap or the amount of value that LP mint tokens are accredited for.

Target Orders:

This account stores dynamic data to facilitate the state transitions that the AMM undergoes while market making on the CLOB. We will discuss the market making mechanism in detail in the next subsection, but here we give a brief overview of the most important fields of this account. You can see its high-level structure in Fig 2.

The amount of reserves available to the AMM at the start of the market making process is stored in `target_x` and `target_y`. Any changes to the underlying reserves due to a swap, deposit or withdrawal between the market making transactions can be detected by comparing the actual reserves to these values.

Similarly, `placed_x` and `placed_y` represent the amount of reserves at the end of the market making process and are later checked to determine whether the current orders need to be replaced.

During planning buy orders, `plan_x_buy` and `plan_y_buy` are updated to reflect the reserves after each order's price and size is calculated as if the order was filled. This

is necessary to plan subsequent orders such that the constant product is increasing and the AMM doesn't end up with bad debt. `plan_x_sell` and `plan_y_sell` have the same role but for sell orders.

As the orders are scheduled and placed in separate steps (multiple transactions or multiple instructions in the same transaction), the price and size of the planned orders are stored in `buy_orders` and `sell_orders` arrays while `replace_buy_client_id` and `replace_sell_client_id` store the order_id of the valid orders currently on the orderbook upon placement.

`plan_orders_cur` and `place_orders_cur` are self-explanatory. When either of them equal the number of orders the AMM wants to place (`amm.order_num`), a state change is initiated to place orders (in case of `plan_orders_cur`) or assign `placed_x` and `placed_y` to the current reserves (in case of `place_orders_cur`).

```
pub struct TargetOrders {
    pub owner: [u64; 4],
    pub buy_orders: [TargetOrder; 50],
    pub sell_orders: [TargetOrder; 50],
    pub target_x: u128,
    pub target_y: u128,
    pub plan_x_buy: u128,
    pub plan_y_buy: u128,
    pub plan_x_sell: u128,
    pub plan_y_sell: u128,
    pub placed_x: u128,
    pub placed_y: u128,
    pub replace_buy_client_id: [u64; MAX_ORDER_LIMIT],
    pub replace_sell_client_id: [u64; MAX_ORDER_LIMIT],
    pub plan_orders_cur: u64,
    pub place_orders_cur: u64,
    pub valid_buy_order_num: u64,
    pub valid_sell_order_num: u64,
}
```

Fig 2. The layout of the TargetOrders account.

Market Making Mechanism

The market making mechanism can be best described as a state machine that the AMM iterates on to manage orders. This process is broken down to multiple steps as the entire flow cannot be executed in a single transaction due to Solana's compute unit (CU) limit. The state transition logic is implemented in an instruction called `monitor_step` that kicks off the appropriate subroutine to perform the required actions and decide the next step. As the program proceeds, the intermediary state is saved. This is what the TargetOrders account is used for.

There are 4 main states that the AMM can be in at any given moment:

1. **Idle:** This is the initial state of the state machine. The AMM stays in this state provided the appropriate number of orders has been placed on the orderbook and there has been no change to the amount of liquidity in the reserves. More specifically, as long as there are no swaps, no liquidity deposit or withdrawal and none of the current AMM limit orders have been filled, no actions will be taken by the AMM. These requirements, referred to as the resting state, can be formalized as follows:

   ```
   bids.len() == target.valid_buy_order_num
   asks.len() == target.valid_sell_order_num
   !bids.is_empty()
   !asks.is_empty()
   x == target.placed_x
   y == target.placed_y
   ```

   where bids and asks are arrays of the open AMM orders on the book, x and y are the amount of assets in each reserve and target is the deserialized TargetOrders account.

   On the other hand, if any of these conditions are violated, a state transition is triggered. In case current price is out of the accepted range or there isn't enough liquidity in the AMM to market make, the state is set to CancellAllOrders. Otherwise, it is set to PlanOrders.

2. **PlanOrders:** In this step, the AMM determines the price and size of the limit orders without actually placing them on the orderbook. There are two notable core constraints in this subroutine. First, the prices at which the orders are placed are at multiples of a unit distance from the current price where the multipliers are derived from the fibonacci sequence. The unit distance itself is a function of the `depth` and

`order_num` parameters of the AMM. Second, the size of each order must be such that if the orders are filled, the resulting asset reserves still respect the constant product formula i.e. the AMM cannot lose money providing liquidity on the orderbook. To enforce this, let's say the current price of the AMM is p and that the AMM wants to place a bid at the price p_1. At this point, `target.plan_x_buy` and `target.plan_y_buy` have the current pool reserve amounts as their value. The size of the order is then computed as

$$delta\_y = \frac{target.plan\_x\_buy}{p_1} - target.plan\_y\_buy$$

To justify this equation, we need to calculate the asset reserves when this order is filled. The change in the asset X reserve will be

$$delta\_x = p_1 \times buy\_size = target.plan\_x\_buy - p_1 \times target.plan\_y\_buy$$

Now, the updated reserve amounts are

$$y + delta\_y = \frac{target.plan\_x\_buy}{p_1}$$

$$x - delta\_x = p_1 \times target.plan\_y\_buy$$

And their product is

$$(x - delta\_x)(y + delta\_y) = target.plan\_x\_buy \,.\, target.plan\_y\_buy$$

Which is the original constant product. Now upon placing each order the value of the `target.plan_x_buy` and `target.plan_y_buy` are updated to reflect the amount of reserve assets as if this order was filled so as to preserve the invariant for the next order. The same applies to the sell orders.

Please note that these computations are a simplified version of the actual implementation as we have skipped the fees for the purpose of illustration. You can see an example of all orders placed on the X-Y curve in Fig 3 for a hypothetical market with current price of 1 and equal reserves of 10 for both assets. To scatter the orders for better visual representation, we have significantly increased the `depth` parameter. In reality the orders are concentrated around the current price more densely.

After all the orders have been planned, the AMM transitions to the PlaceOrders state.

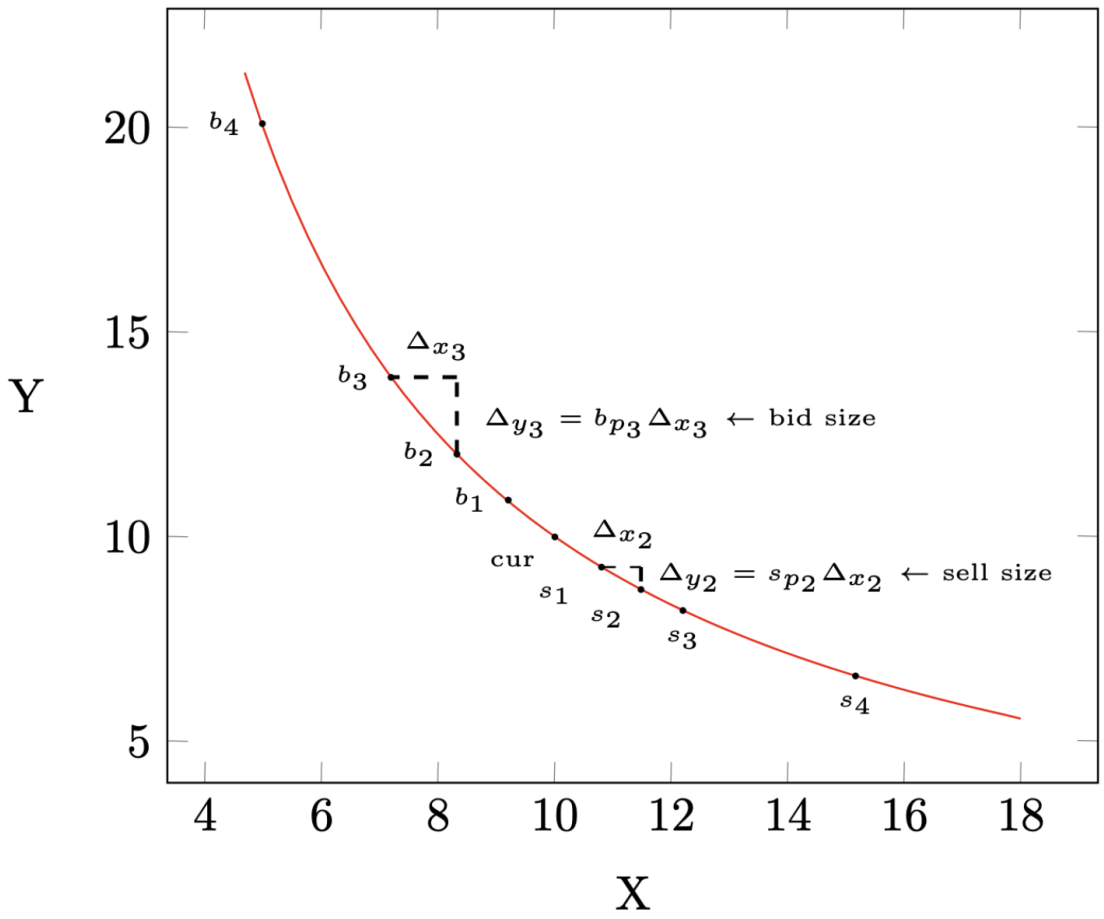

Fig 3. The AMM X-Y curve and the reserve balances at each price point. The vertical distance between each pair is the quote size of the limit order.

3. **PlaceOrders:** Here, the program finally places the orders on the Openbook market. It consumes the data from the PlanOrders state and either places a new order or replaces a current stale order on the orderbook.

   It is worth noting that if there has been any changes in the pool's liquidity reserves between PlanOrders and PlaceOrders transactions, the PlaceOrders subroutine will take the AMM back to the Idle state and won't place any orders as to not violate the constant product formula. This is also true for the PlanOrders subroutine as it can be called in multiple transactions until all the orders are planned and if any swap, deposit or withdrawal transactions happen in the middle, the AMM will go back to the Idle state to synchronize the `target_x` and `target_y` parameters with the new values.

After all orders are placed on the Openbook, the program sets the `target.placed_x` and `target.placed_y` to the current reserve amounts thus bringing the AMM to the resting state.

4. **CancelAllOrders:** As the name suggests, all orders on the orderbook are canceled and the funds are settled back to the program vaults. This can happen if there isn't enough liquidity in the AMM to account for all the deposits or the current price is out of the accepted bounds.

   Before returning, it will set the `plan_order_cur` and `place_order_cur` parameters to 0 and set the AMM state to Idle.

The flowchart in Fig 4 shows a visual summary of the state machine described above.

As a final note, the `monitor_step` instruction needs to be invoked periodically by a client (crank turner) as the on-chain program cannot automatically call itself. Currently, the Raydium team does the cranking for a few selected markets but it is a permission-less job that users can run for any market. The program is supposed to be open sourced soon alongside the SDK upon completion of this audit and we hope that this will help bootstrap interested parties to run their own cranks and further boost liquidity across the DeFi ecosystem.

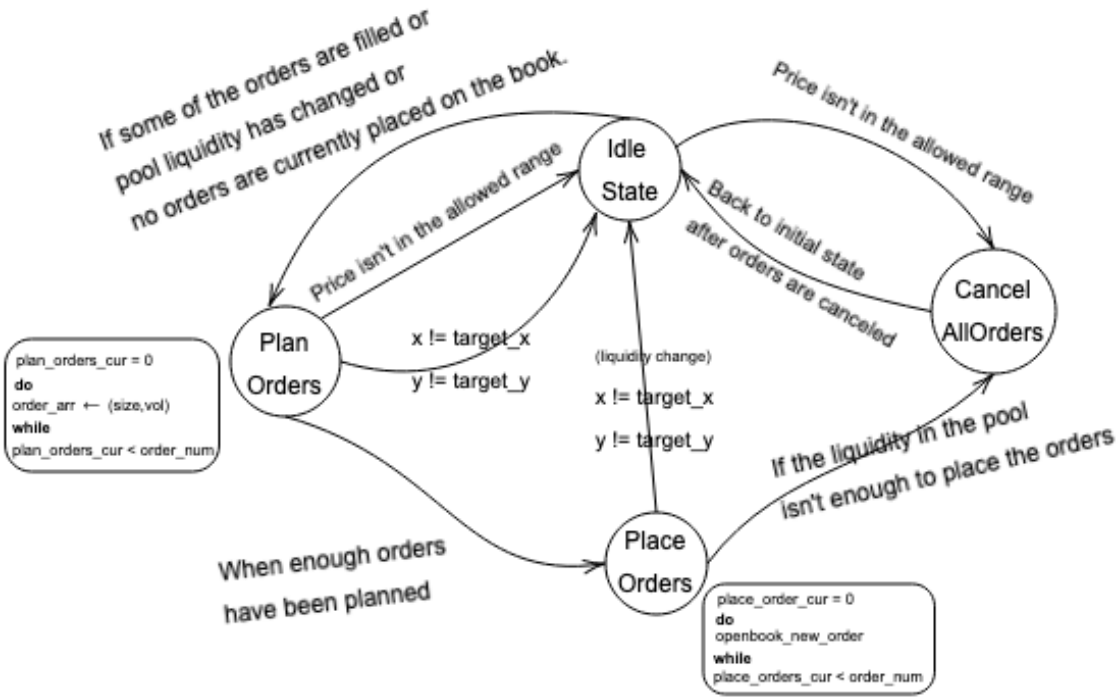

Fig 4. The state machine diagram of the orderbook Market Maker

Methodology

After the initial contact from the Raydium Protocol, we did a quick read through of the source code to evaluate the scope of the work and recognize early potential footguns. Due to the complex nature of the AMM program, we were in constant contact with core contributors to realize the specification in detail and that the program implements it accordingly.

After that, we started to do extensive code analysis. As the audit was requested after the exploit on the Raydium Staking and Farm programs, we took extra care to confirm the AMM program is resilient against authority/signer mischecks which was the root cause of that attack. As it stands, the program uses a Squads multisig wallet to upgrade the program, withdraw PnL and set different market parameters. We verify there are no more potential breaches in the code, however, we still recommend extra caution by the multisig parties to deliberately inspect the instruction input data before signing and executing transactions through the Squads program.

Although we only mentioned two accounts in the account structure section, the program uses various other PDAs for the reserve SPL token accounts, the LP mint account and the Openbook open orders account. We took extra care to confirm that the program is not susceptible to account re-initialization, substitution and token account confusions. One of the pain points in this process was the deprecation of Serum DEX as the CLOB of choice and migrating to Openbook for market making. As the market address is included in all of the PDA seeds, we carefully scrutinized the migration process with deep fuzzy testing to make sure that the AMM could not end up with doubly associated accounts from both Serum and Openbook markets.

Instances of potential vulnerabilities were discovered that we communicated through our channels to the Raydium developers. However, we are happy to announce that no critical or loss of funds bugs were discovered which bears a huge kudos to core contributors to Raydium. There were however some minor issues especially with regards to the aforementioned market migration and improvements that we will describe in the following section.

Findings & Recommendations

In this section, we enumerate some of the minor findings and issues we discovered and explain their implications and resolutions.

1. **Market migration inconsistencies**

   To store the current reserve amounts, the AMM uses an internal system decimal whose value depends on the quote and price lot size of the orderbook. The migration instruction assumed that these lot sizes would remain consistent across both the old Serum and the new Openbook markets. However, this is not always the case.

   To avoid calculations on numbers with different precisions, we recommended to restore the original values of the current parameters before reinitializing the market and then re-normalize them with the new system decimal. We found that the only affected parameters were `target.calc_pnl_x` and `target.calc_pnl_y` which are now correctly converted between the two systems.

```
if new_market_coin_lot_size != market_state.coin_lot_size
   || new_market_pc_lot_size != market_state.pc_lot_size
{
   // market lot size is different, calc_pnl_x & calc_pnl_x must update
   target.calc_pnl_x = Calculator::normalize_decimal_v2(
       pnl_pc_amount,
       amm.pc_decimals,
       amm.sys_decimal_value,
   )
   .as_u128();
   target.calc_pnl_y = Calculator::normalize_decimal_v2(
       pnl_coin_amount,
       amm.coin_decimals,
       amm.sys_decimal_value,
   )
   .as_u128();
}
```

   Another inconsistency in this process was that the migration would proceed without canceling the current orders that the AMM might still have on the Serum book. This could potentially cause deviation from the constant product in unexpected ways.

This issue is resolved by canceling all the orders and settling the funds from the Serum DEX back to the vaults before moving to the new market.

2. Initial LP mint supply

During the audit, a peculiar market was created by a user to trade a coin with 0 decimal. The program set the LP mint decimal equal to the coin decimal therefore for this particular market the LP mint had a decimal of 0. As we discussed in the account structure subsection for AmmInfo, to prevent pricing out smaller liquidity providers, a portion of the initial liquidity would be locked in the pool. This locked amount was set to 1,000,000 shares.

This works perfectly for regular markets where both assets have reasonable decimal values. However, in this case, it would cause almost all the initial liquidity to be locked in the pool resulting in the user being unable to withdraw the majority of their assets.

We emphasize that constant product market makers are not suitable for trading semi fungible assets or assets with low decimal values at all. But to circumvent this situation from happening again where a benign but curious user creates a market locking most of their funds, we suggested modifying the locked amount to be exactly one unit amount of the LP mint. This is backward compatible and has the same effect as the original implementation all while minimizing the locked liquidity for such atypical markets.

```
let liquidity = Calculator::to_u64(
   U128::from(token_pc.amount)
        .checked_mul(token_coin.amount.into())
        .unwrap()
        .integer_sqrt()
        .as_u128(),
)?;

let user_lp_amount = liquidity
   .checked_sub((10u64).checked_pow(lp_mint.decimals.into()).unwrap())
   .ok_or(AmmError::InitLpAmountTooLess)?;
```

3. Missing check in set_param instruction

The set_param instruction allows for many of the market parameters to be changed by the Raydium Squad authority. `order_num` which is the number of orders that the AMM will place on the orderbook is one of these parameters. This parameter is

implicitly bound to `MAX_ORDER_LIMIT` i.e. the length of the `replace_buy_client_id` and `replace_sell_client_id` arrays of the TargetOrders account that store all the order ids. There was no check to ensure that the order_num parameter couldn't be set bigger than `MAX_ORDER_LIMIT` which could temporarily brick the market making operations.

A conditional check is now implemented to fail the erroneous assignment of this parameter to out of bound values.

References

[1] Hayden Adams, Noah Zinsmeister, and Dan Robinson. 2020. Uniswap v2 Core. Retrieved Feb 24, 2021 from https://uniswap.org/whitepaper.pdf