Security Assessment Report

Fusion AMM

July 23, 2025

Summary

The Sec3 team was engaged to conduct a thorough security analysis of the Fusion AMM.

The artifact of the audit was the source code of the following programs, excluding tests, in a private repository.

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

Task	Type	Commit
Fusion AMM	Solana	59b3fd34c3e4a0167efa30281117d67c5a32e563

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

Table of Contents

Result Overview

Issue	Impact	Status
FUSION AMM		
[M-01] Missing update the tick initialized status during decrease limit order	Medium	Resolved
[L-01] Incorrect rounding direction in the `fill_limit_orders` function	Low	Resolved
[I-01] Inefficient tick initialization condition	Info	Resolved
[I-02] Ensure the decrease limit order token amount is non-zero	Info	Resolved
[I-03] Inconsistent handling of transfer hook accounts in `increase_limit_order`	Info	Resolved
[I-04] Incomplete rounding handling in `mul_by_sqrt_price_squared`	Info	Resolved
[Q-01] Edge case in swap logic when `sqrt_price_limit` equals `curr_sqrt_price`	Question	Resolved
[Q-02] Potential inaccuracy in OLP fee calculation	Question	Acknowledged

Findings in Detail

FUSION AMM

[M-01] Missing update the tick initialized status during decrease limit order

According to the implementation of the `_calculate_increase_limit_order` function, once a tick has a non-zero limit order, the tick's initialized status is set to true.

```
/* programs/fusionamm/src/instructions/increase_limit_order.rs */
053 | if amount == 0 {
054 |     return Err(ErrorCode::ZeroTradableAmount.into());
055 | }

/* programs/fusionamm/src/manager/limit_order_manager.rs */
039 | pub fn _calculate_increase_limit_order(
040 |     fusion_pool: &FusionPool,
041 |     limit_order: &LimitOrder,
042 |     tick: &Tick,
043 |     amount: u64,
044 | ) -> Result<IncreaseLimitOrderUpdate> {
045 |     if limit_order.amount == 0 {
046 |         let limit_sqrt_price = sqrt_price_from_tick_index(limit_order.tick_index);
047 |         let is_taker_order =
048 |             (limit_sqrt_price <= fusion_pool.sqrt_price && limit_order.a_to_b)
                      || (limit_sqrt_price >= fusion_pool.sqrt_price && !limit_order.a_to_b);
049 |         if is_taker_order {
050 |             return Err(ErrorCode::TakerOrderNotSupported.into());
051 |         }
052 |     } else if tick.age > limit_order.age {
053 |         return Err(ErrorCode::LimitOrderIsFilled.into());
054 |     }
055 |
056 |     let mut tick_update = TickUpdate::from(tick);
057 |     tick_update.initialized = true;
```

However, in the `decrease_limit_order` function, although it decreases the limit order token amount, it does not check whether the remaining limit order token amount has reached zero. As a result, it fails to update the tick's initialized status accordingly, which may leave the tick in an incorrect state.

```
/* programs/fusionamm/src/manager/limit_order_manager.rs */
079 | pub fn _calculate_decrease_limit_order(
080 |     fusion_pool: &FusionPool,
081 |     limit_order: &LimitOrder,
082 |     tick: &Tick,
083 |     amount: u64,
084 | ) -> Result<DecreaseLimitOrderUpdate> {
085 |     let mut tick_update = TickUpdate::from(tick);
086 |
087 |     if amount > limit_order.amount {
088 |         return Err(ErrorCode::LimitOrderAmountExceeded.into());
089 |     }
090 |
```

```
091 |     // Compute amounts to withdraw in the input and output token.
092 |     // Not filled
093 |     let (amount_in, amount_out) = if limit_order.age == tick.age {
094 |         tick_update.open_orders_input -= amount;
095 |         (amount, 0)
096 |     }
097 |     // Partially filled
098 |     else if limit_order.age + 1 == tick.age {
099 |         let remaining_input = checked_mul_div_u64(amount,
                  tick.part_filled_orders_remaining_input, tick.part_filled_orders_input)?;
100 |         let amount_out = limit_order.get_swap_out_amount(amount - remaining_input)?;
101 |
102 |         tick_update.part_filled_orders_input -= amount;
103 |         tick_update.part_filled_orders_remaining_input -= remaining_input;
104 |         (remaining_input, amount_out)
105 |     }
106 |     // Fulfilled
107 |     else if limit_order.age + 2 <= tick.age {
108 |         let amount_out = limit_order.get_swap_out_amount(amount)?;
109 |
110 |         if limit_order.a_to_b {
111 |             tick_update.fulfilled_a_to_b_orders_input -= amount;
112 |         } else {
113 |             tick_update.fulfilled_b_to_a_orders_input -= amount;
114 |         }
115 |
116 |         (0, amount_out)
117 |     } else {
118 |         panic!("The limit order can't be older than a tick")
119 |     };
```

A malicious user could repeatedly increase and decrease limit orders at usable ticks, forcing swap execution to cross multiple meaningless ticks. This could introduce a potential DoS risk.

It is recommended to add logic in the `decrease_limit_order` function to update the tick's initialized status.

Resolution

This issue has been fixed by `d2ae624`.

FUSION AMM

[L-01] Incorrect rounding direction in the `fill_limit_orders` function

In the `fill_limit_orders` function, when a user initiates a swap with a specified input amount (i.e., `amount_specified_is_input` is `true`), the program enters a branch that first calculates the total input tokens required to fill all available limit orders at the current price tick. This logic is intended to determine the maximum possible exchange at that price.

```
/* programs/fusionamm/src/math/swap_limit_math.rs */
026 | let part_filled_orders_remaining_input = tick.open_orders_input + tick.part_filled_orders_remaining_input;
027 |
028 | if amount_specified_is_input {
029 |     // Total possible swap input.
030 |     result.amount_in = get_limit_order_output_amount(part_filled_orders_remaining_input, !a_to_b, sqrt_price,
↪  false)?;
031 |     // The total amount of the limit order input token that can be swapped.
032 |     result.amount_out = part_filled_orders_remaining_input;
```

However, the implementation of this calculation contains an incorrect rounding direction. The code uses `part_filled_orders_remaining_input` (the total tokens available in limit orders, which represents the output for the current swap) to calculate the required number of input tokens the user must pay. On line 30, the call to `get_limit_order_output_amount` has its `round_up` parameter set to `false`, which forces the result to be rounded down.

By rounding down, calculates a required input amount that is fractionally less than what is actually required, creating a small but systemic value leakage that benefits the user at the expense of the protocol.

Resolution

This issue has been fixed by `6ec0fb2`.

FUSION AMM

[I-01] Inefficient tick initialization condition

According to the implementation of the `next_tick_modify_liquidity_update` function, a tick is considered uninitialized only when its `liquidity_gross` becomes zero and all associated limit order fields are also zero.

```
/* programs/fusionamm/src/manager/tick_manager.rs */
037 | pub fn next_tick_modify_liquidity_update(
038 |     tick: &Tick,
039 |     tick_index: i32,
040 |     tick_current_index: i32,
041 |     fee_growth_global_a: u128,
042 |     fee_growth_global_b: u128,
043 |     liquidity_delta: i128,
044 |     is_upper_tick: bool,
045 | ) -> Result<TickUpdate, ErrorCode> {
046 |     // noop if there is no change in liquidity
047 |     if liquidity_delta == 0 {
048 |         return Ok(TickUpdate::from(tick));
049 |     }
050 |
051 |     let liquidity_gross = add_liquidity_delta(tick.liquidity_gross, liquidity_delta)?;
052 |
053 |     if liquidity_gross == 0 {
054 |         return if tick.open_orders_input == 0
055 |             && tick.part_filled_orders_input == 0
056 |             && tick.part_filled_orders_remaining_input == 0
057 |             && tick.fulfilled_a_to_b_orders_input == 0
058 |             && tick.fulfilled_b_to_a_orders_input == 0
059 |         {
060 |             // Update to an uninitialized tick if remaining liquidity is being removed
061 |             Ok(TickUpdate::default())
062 |         } else {
063 |             Ok(TickUpdate {
064 |                 initialized: true,
... |                 // ...
075 |             })
076 |         };
077 |     }
```

However, a discrepancy exists between the fields required to keep a tick initialized versus those that provide actual, tradable liquidity. A swap exclusively draws upon `tick.open_orders_input` and `tick.part_filled_orders_remaining_input` for liquidity. The other fields, such as `tick.fulfilled_a_to_b_orders_input`, are historical counters for consumed orders and do not represent available liquidity.

This leads to an inefficiency where a tick with zero `liquidity_gross` and no open or partially-filled orders can remain `initialized` solely due to a non-zero value in a `fulfilled_*` field. These ticks contain no usable liquidity but are still loaded and processed during swap operations, causing unnecessary computational overhead and degrading performance. Consider refining the tick initialization condition, while ensuring that the related mechanisms for tick.age updates and the accounting of fulfilled orders continue

to function correctly.

Resolution

This issue has been fixed by `41548c4` and `1a446b2`.

FUSION AMM

[I-02] Ensure the decrease limit order token amount is non-zero

The `decrease_limit_order` instruction is designed to allow users to withdraw the liquidity from an existing limit order position.

```
/* programs/fusionamm/src/instructions/decrease_limit_order.rs */
054 | pub fn handler<'info>(
055 |     ctx: Context<'_, '_, '_, 'info, DecreaseLimitOrder<'info>>,
056 |     amount: u64,
057 |     remaining_accounts_info: Option<RemainingAccountsInfo>,
058 | ) -> Result<()> {
059 |     verify_position_authority_interface(&ctx.accounts.limit_order_token_account,
↪  &ctx.accounts.limit_order_authority)?;
060 |
061 |     let fusion_pool = &mut ctx.accounts.fusion_pool;
062 |     let limit_order = &mut ctx.accounts.limit_order;
063 |
064 |     // Process remaining accounts
065 |     let remaining_accounts =
066 |         parse_remaining_accounts(ctx.remaining_accounts, &remaining_accounts_info,
↪  &[AccountsType::TransferHookA, AccountsType::TransferHookB])?;
067 |
068 |     let update = calculate_decrease_limit_order(fusion_pool, limit_order, &ctx.accounts.tick_array, amount)?;
```

However, the current implementation lacks a validation check on the input `amount` to ensure it is a non-zero value. Regardless of the order's fill state, a zero-amount action is a meaningless operation. It is recommended to add a check to reject such transactions.

Resolution

This issue has been fixed by `1bc3613`.

FUSION AMM

[I-03] Inconsistent handling of transfer hook accounts in `increase_limit_order`

In Solana's Token-2022 standard, tokens can be created with a "transfer hook" extension. When a transfer of such a token occurs, the Token-2022 program invokes a specified custom program. Instructions that perform token transfers, such as `increase_limit_order`, must pass the hook program to the transfer instruction to successfully execute the transfer.

However, there is a discrepancy in the `increase_limit_order` instruction between its documented intent and the actual implementation for handling these transfer hook programs. The comment at lines 42-43 suggests that the instruction is designed to handle remaining accounts for *either* `token_mint_a` *or* `token_mint_b`, depending on which token is being transferred:

```
/* programs/fusionamm/src/instructions/increase_limit_order.rs */
042 | // remaining accounts
043 | // - accounts for transfer hook program of token_mint_a or token_mint_b
```

This indicates that only the accounts for the specific token being transferred are necessary. In practice, the implementation contradicts this. The `parse_remaining_accounts` function call on line 58 is configured to parse and require accounts for *both* `TransferHookA` and `TransferHookB` in all cases:

```
/* programs/fusionamm/src/instructions/increase_limit_order.rs */
057 | // Process remaining accounts
058 | let remaining_accounts =
059 |     parse_remaining_accounts(ctx.remaining_accounts, &remaining_accounts_info, &[AccountsType::TransferHookA,
↪  AccountsType::TransferHookB])?;
```

This implementation forces the caller to provide account information for both potential transfer hooks, even though only one set will ever be used for a single `increase_limit_order` call (determined by the `limit_order.a_to_b` flag). This creates unnecessary complexity, increases transaction size, and introduces a potential point of error for developers integrating with the instruction.

Resolution

This issue has been fixed by `268c67e`.

FUSION AMM

[I-04] Incomplete rounding handling in `mul_by_sqrt_price_squared`

The `mul_by_sqrt_price_squared` function is intended to multiply a given `amount` by the square of a `sqrt_price`, with a `round_up` parameter to control rounding behavior. However, the internal calculation of the price does not correctly implement this rounding.

The issue is on line 312:

```
/* programs/fusionamm/src/math/token_math.rs */
311 | pub fn mul_by_sqrt_price_squared(amount: u64, sqrt_price: u128, round_up: bool) -> Result<u64, ErrorCode> {
312 |     let price = mul_u256(sqrt_price, sqrt_price).shift_word_right();
313 |     let value_u256 = U256Muldiv::new(0, amount as u128).mul(price);
314 |     let value_u128 = if round_up && value_u256.get_word(0) > 0 {
315 |         value_u256.shift_word_right().try_into_u128()? + 1
316 |     } else {
317 |         value_u256.shift_word_right().try_into_u128()?
318 |     };
319 |     u64::try_from(value_u128).or(Err(ErrorCode::NumberDownCastError))
320 | }
```

The `shift_word_right()` operation performs a bitwise shift to scale the squared price, which effectively truncates the result. This is equivalent to always rounding down. The `round_up` parameter, which is intended to control the rounding direction, is not factored into this specific calculation.

However, all current usages of `mul_by_sqrt_price_squared` throughout the codebase involve a specific, fixed set of `sqrt_price` values. For these particular inputs, this tiny rounding error is not exploitable.

Resolution

This issue has been fixed by `0c9507e`.

FUSION AMM

[Q-01] Edge case in swap logic when `sqrt_price_limit` equals `curr_sqrt_price`

Fusion AMM adds limit order filling logic directly into the swap loop, and there exists a situation where this implementation can prevent the complete consumption of limit orders.

Consider the following scenario:

A pool has active limit orders at a tick whose square root of price is 80, while the current pool price is 90. A user executes an `a_to_b` swap with a `sqrt_price_limit` of 80. The swap consumes liquidity and moves the pool's `sqrt_price` to exactly 80. However, the swap amount is not large enough to fill all the limit orders at that price, so some limit orders remain. If the user then initiates a second `a_to_b` swap, again with a `sqrt_price_limit` of 80, this second swap does not execute and fails to consume the remaining limit orders.

This behavior appears to be caused by the swap loop's continuation condition on line 71:

```
/* programs/fusionamm/src/manager/swap_manager.rs */
071 | while amount_remaining > 0 && adjusted_sqrt_price_limit != curr_sqrt_price {
... |     // swap logic
208 | }
```

In this scenario, at the start of the second swap, `curr_sqrt_price` (80) is already equal to `adjusted_sqrt_price_limit` (80). This makes the loop condition immediately false, so no swapping occurs.

We'd like to confirm if this is an intentional design perhaps as a trade-off to maintain code simplicity. If it is a deliberate design choice, we would recommend follow Orca's PR #918 to reject these cases early.

Resolution

The team clarified that it's intended to reject swap with zero slippage. And in commit `1feb386`, the team adopted the logic from Orca's PR #918 to reject zero-slippage swaps upfront.

FUSION AMM

[Q-02] Potential inaccuracy in OLP fee calculation

When a user's limit order is filled during a swap, an "order liquidity providers fee" (`olp_fee`) is generated and accrued in the pool's state. The user can later claim their filled assets and the corresponding portion of these fees by calling the `decrease_limit_order` instruction.

```
/* programs/fusionamm/src/manager/limit_order_manager.rs */
134 | if delta_filled_amount_a > 0 {
135 |     delta_olp_fee_owed_b = checked_mul_div_u64(fusion_pool.olp_fee_owed_b, delta_filled_amount_a,
↪  fusion_pool.orders_filled_amount_a)?;
136 | }

144 | if delta_filled_amount_b > 0 {
145 |     delta_olp_fee_owed_a = checked_mul_div_u64(fusion_pool.olp_fee_owed_a, delta_filled_amount_b,
↪  fusion_pool.orders_filled_amount_b)?;
146 | }
```

However, the fee calculation relies on pool-level accumulators, which may lead to inaccuracies depending on when the user claims their funds.

The potential issue arises because a `decrease_limit_order` request may occur long after the swap that actually filled the order. During this intervening period, subsequent swaps may fill other limit orders at entirely different price ticks. These subsequent fills will modify the pool-level `olp_fee_owed` and `orders_filled_amount` accumulators.

As a result, the fee amount a user receives is dependent on unrelated trading activity that occurred *after* their own order was filled. This suggests the fee awarded is not a direct accounting of what their specific order generated, but rather an estimation based on the pool's state at the moment of withdrawal. Limit orders filled later at different prices can influence the fees earned by a previous limit order.

We would like to confirm if this is the intended behavior, perhaps as a form of estimation or simplification to avoid the complexity and storage overhead of tracking fees at a more granular tick level.

Resolution

The team acknowledged this issue and chose to retain the current implementation. They cited storage constraints as the primary reason: the current available tick space (10240 / 88 = 116 bytes) makes it impractical to track fees at a more granular level without significant refactoring.

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 CrypticDot dba DefiTuna (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.